1. Singly Linked List implementation using built-in class

Program:

```
import java.util.LinkedList;
class Main1 {
  public static void main(String[] args) {
    LinkedList<String> languages = new LinkedList<>();
    // add elements in LinkedList
    languages.add("Java");
    languages.add("Python");
    languages.add("JavaScript");
    languages.add("Kotlin");
    languages.add("1");
    System.out.println("LinkedList: " + languages);
    // remove elements from index 1
    String str = languages.remove(1);
    System.out.println("Removed Element: " + str);
    System.out.println("Updated LinkedList: " + languages);
  }
}
```

2. Implement Stack using Stack class

Program:

```
import java.io.*;
import java.util.Stack;

class Stack1
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }
// popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop Operation:");
```

```
        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }

    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top: " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer) stack.search(element);
          if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position: " + pos);
    }
  public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();
        stack_push (stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
```

3. Queue Implementation Queue using Queue class

Program:

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {

    public static void main(String[] args)
```

```
    {
        Queue<Integer> q  = new LinkedList<>();
         // Adds elements {0, 1, 2, 3, 4} to  the queue
        for (int i = 0; i < 5; i++)
            q.add(i);
         // Display contents of the queue.
        System.out.println("Elements of queue "  + q);
         // To remove the head of queue.
        int removedele = q.remove();
        System.out.println("removed element-"  + removedele);
         System.out.println(q);
         // To view the head of queue
        int head = q.peek();
        System.out.println("head of queue-"   + head);
        int size = q.size();
        System.out.println("Size of queue-"  + size);
    }
}
```

4. Implement Set using Set class

Program

```
import java.util.*;
public class SetExample {
    public static void main(String[] args)  {
        // Set demo with HashSet
        Set<String> Colors_Set = new HashSet<String>();
        Colors_Set.add("Red");
        Colors_Set.add("Green");
        Colors_Set.add("Blue");
        Colors_Set.add("Cyan");
        Colors_Set.add("Magenta");
        //print set contents
        System.out.print("Set contents:");
        System.out.println(Colors_Set);

        // Set demo with TreeSet
        System.out.print("\nSorted Set after converting to TreeSet:");
        Set<String> tree_Set = new TreeSet<String>(Colors_Set);
        System.out.println(tree_Set);
    }
}
```

5. Implementation of Map using LinkedHashMap class

Program

```
import java.util.LinkedHashMap;
public class MapExample {
   public static void main(String[] args)
   {
      // Creating an empty Linked Hash Map
      LinkedHashMap<Integer, String> students = new LinkedHashMap<>();
      // Adding data to Linked Hash Map in key-value pair
      students.put(101, "Aaliyah");
      students.put(102, "Taylor");
      students.put(103, "Zayn");
      students.put(104, "Sabrina");
      students.put(105, "Paul");
      // Showing size and data of the Linked Hash Map
      System.out.println("The size of the Linked Hash Map is:- "+ students.size());
      System.out.println(students);
      // Checking whether a certaint key is available or not
      if (students.containsKey(105)) {
         String name = students.get(105);
         System.out.println("The name of the student having Id 105 is:- " + name);
      }
   }
}
```

Hive Database

In Hive, the database is considered as a catalog or namespace of tables. So, we can maintain multiple tables within a database where a unique name is assigned to each table. Hive also provides a default database with a name **default**.

- **hive> show databases;** **// to check the existing databases**
- **hive> create database demo;** **// to create new database**
- **hive> create database if not exists demo;**
- **hive>create database demo**

  >WITH DBPROPERTIES ('creator' = 'sumit', 'date' = '2019-06-03');
- **hive> describe database extended demo;**
- **hive> drop database demo;**
- **hive> drop database if exists demo;**
- **hive> drop database if exists demo cascade;**

➢ **Hive Tables**

In Hive, we can create a table by using the conventions similar to the SQL. It supports a wide range of flexibility where the data files for tables are stored. It provides two types of table: -

- Internal table
- External table

Internal Table

The internal tables are also called managed tables as the lifecycle of their data is controlled by the Hive. By default, these tables are stored in a subdirectory under the directory defined by hive.metastore.warehouse.dir (i.e. /user/hive/warehouse). The internal tables are not flexible enough to share with other tools like Pig. If we try to drop the internal table, Hive deletes both table schema and data.

```
hive> create table demo.employee (Id int, Name string , Salary float)
row format delimited
fields terminated by ',' ;
```

External Table

The external table allows us to create and access a table and a data externally. The **external** keyword is used to specify the external table, whereas the **location** keyword is used to determine the location of loaded data.

As the table is external, the data is not present in the Hive directory. Therefore, if we try to drop the table, the metadata of the table will be deleted, but the data still exists.

$hdfs dfs -mkdir /HiveDirectory
$hdfs dfs -put hive/emp_details /HiveDirectory

hive> create external table emplist (Id int, Name string , Salary float)
row format delimited
fields terminated by ','
location '/HiveDirectory';

select * from emplist;

Hive - Load Data

Once the internal table has been created, the next step is to load the data into it. So, in Hive, we can easily load data from any file to the database.

- Let's load the data of the file into the database by using the following command: -

  hive> load data local inpath '/homehive/empdata' into table demo.emp;

  Here, *empdata* is the file name that contains the data.

  hive> select * from demo.emp;

If we want to add more data into the current database, execute the same query again by just updating the new file name.

hive>load data local inpath '/home/hive/empdata1' into table demo.emp;

hive>select * from demo.emp;

- In Hive, if we try to load unmatched data (i.e., one or more column data doesn't match the data type of specified table columns), it will not throw any exception. However, it stores the Null value at the position of unmatched tuple.

➢ **Hive - Drop Table**

Hive facilitates us to drop a table by using the SQL **drop table** command. Let's follow the below steps to drop the table from the database.

- Let's check the list of existing databases by using the following command:
  hive> show databases;
- Now select the database from which we want to delete the table by using the following command: -
  hive> use demo;
- Let's check the list of existing tables in the corresponding database.
  hive> show tables;
- Now, drop the table by using the following command: -
  hive> drop table demo.emp;

- Let's check whether the table is dropped or not.
  hive> show tables;

Hive - Alter Table

In Hive, we can perform modifications in the existing table like changing the table name, column name, comments, and table properties. It provides SQL like commands to alter the table.

Rename a Table

If we want to change the name of an existing table, we can rename that table by using the following signature: -

hive>ALTER TABLE old_table_name RENAME TO new_table_name;

Let's see the existing tables present in the current database.

hive>SHOW TABLES;

Adding column

In Hive, we can add one or more columns in an existing table by using the following syntax

hive>ALTER TABLE table_name ADD COLUMNS(column_name datatype);

Ex: hive>Alter table demo.emp add columns (age int);

hive>describe demo.emp;
hive>select * from demo.emp;

Change Column

In Hive, we can rename a column, change its type and position. Here, we are changing the name of the column by using the following signature: -

hive>ALTER TABLE table_name CHANGE old_column_name new_column_name datatyp;

Ex: hive>ALTER TABLE demo.emp CHANGE name fname string;
hive>DESCRIBE demo.emp;
hive>SELECT * FROM demo.emp;

Delete or Replace Column

Hive allows us to delete one or more columns by replacing them with the new columns. Thus, we cannot drop the column directly.

Let's see the existing schema of the table.

hive>DESCRIBE demo.emp;

```
OK
id                      int
name                    string
salary                  float
age                     int
Time taken: 0.308 seconds, Fetched: 4 row(s)
hive>
```

Now, drop a column from the table.

hive>ALTER TABLE demp.emp REPLACE COLUMNS(id string, first_name string, age int);

hive>DESCRIBE demo.emp;

```
OK
id                      string
first_name              string
age                     int
Time taken: 0.337 seconds, Fetched: 3 row(s)
hive>
```

Operations in HiveQL

Arithmetic Operations

hive> select id, name, salary + 50 from employee;
hive> select id, name, (salary * 10) /100 from employee; **//find 10% salary from each employee**

Relational Operators

hive> select * from employee where salary >= 25000;
hive> select * from employee where salary < 25000;

Aggregate Functions in Hive

In Hive, the aggregate function returns a single value resulting from computation over many rows. Let"s see some commonly used aggregate functions: -

Return Type	Operator	Description
BIGINT	count(*)	It returns the count of the number of rows present in the file.
DOUBLE	sum(col)	It returns the sum of values.
DOUBLE	sum(DISTINCT col)	It returns the sum of distinct values.
DOUBLE	avg(col)	It returns the average of values.
DOUBLE	avg(DISTINCT col)	It returns the average of distinct values.
DOUBLE	min(col)	It compares the values and returns the minimum one form it.
DOUBLE	max(col)	It compares the values and returns the maximum one form it.

Sorting and Aggregating

- Sorting data in Hive can be achieved by using a standard ORDER BY clause, but there is a catch. ORDER BY produces a result that is totally sorted, as expected, but to do so it sets the number of reducers to one, making it very inefficient for large datasets.
- When a globally sorted result is not required and in many cases it isn't, you can use Hive's nonstandard extension, SORT BY, instead. SORT BY produces a sorted file per reducer.
- In some cases, you want to control which reducer a particular row goes to, typically so you can perform some subsequent aggregation. This is what Hive's DISTRIBUTE BY clause does.

GROUP BY and HAVING Clause

The Hive Query Language provides GROUP BY and HAVING clauses that facilitate similar functionalities as in SQL. Here, we are going to execute these clauses on the records of the below table:

- *Create Table:*

hive> create table emp (Id int, Name string, Salary float, Desig string)
>row format delimited
>fields terminated by ',' ;

- *Load data into emp table:*

hive> load data local inpath '/home/hive/emp_data' into table emp;

ID	NAME	SALARY	DESIG
1	John	40,000	Sr. Manager
2	Harry	30,000	Developer
3	Clark	25,000	Developer
4	Ram	35,000	Manager
5	Imran	50,000	Sr. Manager
6	Lisa	45,000	Manager
7	Syam	35,000	Manager

GROUP BY Clause:

The HQL Group By clause is used to group the data from the multiple records based on one or more column. It is generally used in conjunction with the aggregate functions (like SUM, COUNT, MIN, MAX and AVG) to perform an aggregation over each group.

hive> SELECT desig, sum(salary) from emp GROUP BY desig;
Developer 55,000
Manager 1,15,000
Sr. Manager 90,000

HAVING Clause:

The HQL HAVING clause is used with GROUP BY clause. Its purpose is to apply constraints on the group of data produced by GROUP BY clause. Thus, it always returns the data where the condition is TRUE.

hive> SELECT desig, sum(salary) from emp
>GROUP BY desig HAVING salary>=30,000;
Developer 30,000
Manager 1,15,000
Sr. Manager 90,000

ORDER BY and SORT BY Clause

By using HiveQL ORDER BY and SORT BY clause, we can apply sort on the column. It returns the result set either in ascending or descending order. Here, we are going to execute these clauses on the records of the below table:

ID	NAME	SALARY	DESIG
1	John	40,000	Sr. Manager
2	Harry	30,000	Developer
3	Clark	25,000	Developer
4	Ram	35,000	Manager
5	Imran	50,000	Sr. Manager
6	Lisa	45,000	Manager
7	Syam	35,000	Manager

ORDER BY Clause

In HiveQL, ORDER BY clause performs a complete ordering of the query result set. Hence, the complete data is passed through a single reducer. This may take much time in the execution of large datasets. However, we can use LIMIT to minimize the sorting time.

hive> select * from emp order by salary desc;

ID	NAME	SALARY	DESIG
5	Imran	50,000	Sr. Manager
6	Lisa	45,000	Manager
1	John	40,000	Sr. Manager
4	Ram	35,000	Manager
7	Syam	35,000	Manager
2	Harry	30,000	Developer
3	Clark	25,000	Developer

SORT BY Clause

The HiveQL SORT BY clause is an alternative of ORDER BY clause. It orders the data within each reducer. Hence, it performs the local ordering, where each reducer's output is sorted separately. It may also give a partially ordered result.

hive> select * from emp sort by salary desc;

ID	NAME	SALARY	DESIG
5	Imran	50,000	Sr. Manager
6	Lisa	45,000	Manager
1	John	40,000	Sr. Manager
4	Ram	35,000	Manager
7	Syam	35,000	Manager
2	Harry	30,000	Developer
3	Clark	25,000	Developer

DISTRIBUTE BY:

Distribute BY clause used on tables present in Hive. Hive uses the columns in Distribute by to distribute the rows among reducers. All Distribute BY columns will go to the same reducer.

- It ensures each of N reducers gets non-overlapping ranges of column
- It doesn't sort the output of each reducer

hive>SELECT Id, Name from emp DISTRIBUTE BY Id;

Pig Relational Operators:

Category	Operator	Description
Loading and Storing	LOAD STORE DUMP	Loads data from the file system or other storage into a relation . Saves a relation to the file system or other storage. Prints a relation to the console.
Filtering	FILTER DISTINCT FOREACH...GENERATE STREAM	Removes unwanted rows from a relation. Removes duplicate rows from a relation. Adds or removes fields from a relation. Transforms a relation using an external program.
Grouping and Joining	JOIN COGROUP GROUP CROSS	Joins two or more relations. Groups the data in two or more relations. Groups the data in a single relation. Creates the cross product of two or more relations.
Sorting	ORDER LIMIT	Sorts a relation by one or more fields. Limits the size of a relation to a maximum number of tuples.
Combining and Splitting	UNION SPLIT	Combines two or more relations into one. Splits a relation into two or more relations.

Loading and Storing

LOAD

$LOAD 'info' [USING FUNCTION] [AS SCHEMA];

- o LOAD is a relational operator.
- o 'info' is a file that is required to load. It contains any type of data.
- o USING is a keyword.
- o FUNCTION is a load function.
- o AS is a keyword.
- o SCHEMA is a schema of passing file, enclosed in parentheses.

Example:

• File in local file system

```
$cat data.txt
1010,10,3
2020,20,4
3030,30,5
4040,40,2
```

• Loading data file into HDFS file system

```
$ hdfs dfs -put data.txt /pigtest
```

• Starting pig grunt shell

```
$pig -x mapreduce or $pig
```

• Loading data into pig by defining schema and fields are separated with comma.

```
grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',') AS (d1:int,d2:int,d3:int) ;
```

• Printing loaded data on console

grunt> DUMP A;
(1010,10,3)
(2020,20,4)
(3030,30,5)
(4040,40,2)

STORE

• Stores or saves results to the file system.

grunt>STORE A INTO 'myoutput' USING PigStorage ('*');

Filtering :

FILTER

• File in local file system

$cat data.txt
1,10,3
2,20,4
3,10,3
4,20,4

• Loading data file into HDFS file system

$ hdfs dfs -put data.txt /pigtest

• Starting pig grunt shell

$pig -x mapreduce or $pig

• Loading data into pig by defining schema and fields are separated with comma.

grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',') AS (d1:int,d2:int,d3:int) ;

• To remove duplicate data

grunt>B = FILTER A BY d2 == 10;

• Printing loaded data on console

grunt> DUMP B;
(1,10,3)
(3, 10,3)

FOREACH

• File in local file system

$cat data.txt
1,2,3,4
5,6,7,8
8,7,6,5
4,3,2,1

• Loading data file into HDFS file system

$ hdfs dfs -put data.txt /pigtest

• Starting pig grunt shell

$pig -x mapreduce or $pig

• Loading data into pig by defining schema and fields are separated with comma.

grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',') AS (d1:int,d2:int,d3:int, d4:int) ;

• To fetch second and fourth columns

grunt>B = FOREACH A GENERATE d2,d4;

• Printing loaded data on console

grunt> DUMP B;

(2,4)

(6,8)

(7,5)

(3,1)

DISTINCT

• File in local file system

$cat data.txt

1,10,3

2,20,4

1,10,3

2,20,4

• Loading data file into HDFS file system

$ hdfs dfs -put data.txt /pigtest

• Starting pig grunt shell

$pig -x mapreduce or $pig

• Loading data into pig by defining schema and fields are separated with comma.

grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',') AS (d1:int,d2:int,d3:int) ;

• To remove duplicate data

grunt>B = DISTINCT A

• Printing loaded data on console

grunt> DUMP B;

(1, 10, 3)

(2, 20, 4)

Grouping and Joining

CROSS

• File in local file system

$cat data1.txt

1,2

2,3

$cat data2.txt

3,4,5

4,5,6

• Loading data file into HDFS file system

$ hdfs dfs -put data1.txt /pigtest

$ hdfs dfs -put data2.txt /pigtest

• Starting pig grunt shell

$pig

• Loading data into pig by defining schema and fields are separated with comma.

grunt> A = LOAD '/pigtest/data1.txt' USING PigStorage(',') AS (d1:int,d2:int) ;

grunt> B = LOAD '/pigtest/data2.txt' USING PigStorage(',') AS (d1:int,d2:int,d3:int) ;

• Cross product of data1.txt and data2.txt

grunt> C=CROSS A,B;

• Printing final output

grunt> DUMP C;

(1,2,3,4,5)

(1,2,4,5,6)

(2,3,3,4,5)

(2,3,4,5,6)

GROUP BY

• File in local file system

$cat data.txt

John,Ram,3

Clark,John,2

Nike,Ram,5

Imran,John,6

• Loading data file into HDFS file system

$ hdfs dfs -put data.txt /pigtest

• Starting pig grunt shell

$pig -x mapreduce or $pig

• Loading data into pig by defining schema and fields are separated with comma.

grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',')

AS (d1:chararray,d2:chararray,d3:int) ;

• To group the data based on d2 column data

grunt>B = GROUP A BY d2;

• Printing loaded data on console

grunt> DUMP B;

(Ram, {(John,Ram,3), (Nike,Ram,5)})

(John, {(Clark,John,2), (Imran,John,6)}

JOIN

• File in local file system

$cat student.txt

1,Ram,9.8

2,John,7.8

3,Ram,6.7

4,John,6.6

$cat department.txt
1,101,IT
2,101,IT
3,101,IT
4,101,IT

• Loading data file into HDFS file system
$ hdfs dfs -put student.txt /pigtest
$ hdfs dfs -put department.txt /pigtest
• Starting pig grunt shell
$pig -x mapreduce or $pig
• Loading data into pig by defining schema and fields are separated with comma.
grunt> A = LOAD '/pigtest/student.txt' USING PigStorage(',')
AS (rollno:int, name:chararray ,gpa:float) ;
grunt> B = LOAD '/pigtest/department.txt' USING PigStorage(',')
AS (rollno:int, deptno:int ,deptname:chararray) ;
• To join the data based on rollno
grunt>C = JOIN A BY rollno, B BY rollno;
• Printing loaded data on console
grunt> DUMP C;
1,Ram,9.8,1,101,IT
2,John,7.8,1,101,IT
3,Ram,6.7,1,101,IT
4,John,6.6,1,101,IT

Sorting:

ORDER BY

• File in local file system
$cat data.txt
John,Ram,3
Clark,John,2
Nike,Ram,5
Imran,John,6
• Loading data file into HDFS file system
$ hdfs dfs -put data.txt /pigtest
• Starting pig grunt shell
$pig -x mapreduce or $pig
• Loading data into pig by defining schema and fields are separated with comma.
grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',')
AS (d1:chararray,d2:chararray,d3:int) ;
• To sort tuples in an Order
grunt>B = ORDER A BY d3 DESC;

• Printing loaded data on console

```
grunt> DUMP B;
        Imran,John,6
        Nike,Ram,5
        John,Ram,3
        Clark,John,2
```

LIMIT

• File in local file system

```
$cat data.txt
John,Ram,3
Clark,John,2
Nike,Ram,5
Imran,John,6
```

• Loading data file into HDFS file system

```
$ hdfs dfs -put data.txt /pigtest
```

• Starting pig grunt shell

```
$pig -x mapreduce or $pig
```

• Loading data into pig by defining schema and fields are separated with comma.

```
grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',')
       AS (d1:chararray,d2:chararray,d3:int) ;
```

• To print only first two tuples

```
grunt>B = LIMIT A 2;
```

• Printing loaded data on console

```
grunt> DUMP B;
        John,Ram,3
        Clark,John,2
```

<u>Combining and Splitting:</u>

UNION

• File in local file system

```
$cat data1.txt
John,Ram,3
Clark,John,2
$cat data2.txt
Nike,Ram,5
Imran,John,6
```

• Loading data file into HDFS file system

```
$ hdfs dfs -put data1.txt /pigtest
$ hdfs dfs -put data2.txt /pigtest
```

• Starting pig grunt shell

```
$pig -x mapreduce or $pig
```

• Loading data into pig by defining schema and fields are separated with comma.

```
grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',')
        AS (d1:chararray,d2:chararray,d3:int) ;
```

```
grunt> B = LOAD '/pigtest/data2.txt' USING PigStorage(',')
        AS (d1:chararray,d2:chararray,d3:int) ;
```

• To combine two bags as one bag

```
grunt>C = UNION A,B;
```

• Printing loaded data on console

```
grunt> DUMP C;
        John,Ram,3
        Clark,John,2
        Nike,Ram,5
        Imran,John,6
```

SPLIT

• File in local file system

```
$cat data.txt
1,2
2,4
3,6
4,8
5,7
6,5
7,3
8,1
```

• Loading data file into HDFS file system

```
$ hdfs dfs -put data.txt /pigtest
```

• Starting pig grunt shell

```
$pig -x mapreduce or $pig
```

• Loading data into pig by defining schema and fields are separated with comma.

```
grunt> A = LOAD '/pigtest/data.txt' USING PigStorage(',') AS (d1:int,d2:int) ;
```

• To Split the tuples based on field values

```
grunt> SPLIT A INTO X IF d1<=5, Y IF d1>=6;
```

• Printing loaded data on console

```
grunt> DUMP X;
(1,2)
(2,4)
(3,6)
(4,8)
(5,7)
grunt> DUMP Y;
(6,5)
(7,3)
(8,1)
```

WEEK-10

PIG PROGRAMS

OBJECTIVE:

1. Run the Pig Latin Scripts to find Word Count.
2. Run the Pig Latin Scripts to find a max temp for each and every year.

PROGRAM LOGIC:

Run the Pig Latin Scripts to find Word Count.

```
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

Run the Pig Latin Scripts to find a max temp for each and every year

```
records = LOAD 'input/ncdc/micro-tab/sample.txt' AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999
AND
(quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group, MAX(filtered_records.temperature);
DUMP max_temp;
```

(Execute above two programs and write the output) on the left page with input file data and the output