

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Scanner;

/\*\*

\* @author Samuel Swedberg

\* @version 12/8/22

\*

\* A client class demonstrating merge sort on large files

\*/

public class Client {

public static void main(String[] args) {

int size = 20000000;

System.out.printf("Size of data set: %d\n", size);

int blockTotal = 10;

int blockSize = size / blockTotal;

String path = "C:\\data\\data.txt";

String splitPath = "C:\\data\\data";

String sortedPath = "C:\\data\\sortedData.txt";

long start, stop, elapse, startTot, stopTot, elapseTot;

Data[] data = generateData(size);

File file = new File("C:\\data");

file.mkdir();

startTot = System.currentTimeMillis(); // Starts total timer

start = System.currentTimeMillis(); // Starts timer

fileCreate(data, path);

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

System.out.printf("Time to write %d integers : %d msec\n", size, elapse);

start = System.currentTimeMillis(); // Starts timer

split(path, splitPath, blockSize); // Splits the 200,000,000 integers into 10 bloocks of 20,000,000 integers

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

System.out.printf("Time to split into 10 blocks : %d msec\n", elapse);

start = System.currentTimeMillis(); // Starts timer

sortBlocks(splitPath, blockTotal);

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

System.out.printf("Time to sort blocks: %d msec\n", elapse);

start = System.currentTimeMillis(); // Starts timer

mergeBlocks(splitPath, blockTotal, size, blockSize);

stop = System.currentTimeMillis(); // Ends timer

elapse = stop - start; // Elasped time

System.out.printf("Time to merge blocks: %d msec\n", elapse);

stopTot = System.currentTimeMillis(); // Ends total timer

elapseTot = stopTot - startTot; // Elasped total time

System.out.printf("Total external merge sort time: %d msec\n", elapseTot);

System.out.printf("=====================================================\nisSorted checked %d items\n", size);

System.out.println("Verify sort, isSorted = " + isSorted(sortedPath));

}

/\*\*

\* @param arSize

\* @return

\*/

public static Data[] generateData(int arSize)

{

Data[] array;

array = new Data[arSize];

int size = 0;

for(int i=0; i<arSize; i++)

{

Data data = new Data();

array[i] = data;

size++;

}

return array;

}

/\*\*

\* @param data

\* @param path

\*/

public static void fileCreate(Data[] data, String path)

{

try

{

FileWriter fileWriter = new FileWriter(path);

BufferedWriter writer = new BufferedWriter(fileWriter);

int trueSize = data.length;

for(int i=0;i<trueSize;i++)

{

int r = data[i].getNum();

String s = Integer.toString(r);

writer.write(s);

writer.newLine();

}

writer.close();

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

/\*\*

\* @param path

\* @param blockTotal

\*/

public static void sortBlocks(String path, int blockTotal)

{

Comparator compId = new IdComparator();

int blockNum = 0;

for(int i=0;i<blockTotal;i++)

{

try

{

LinkedQueue queue = new LinkedQueue();

String truePath = path + blockNum + ".txt";

File file = new File(truePath);

Scanner scanLine = new Scanner(file);

int z = 0;

while(scanLine.hasNext())

{

queue.enqueue(scanLine.nextLine());

}

int trueSizeQueue = queue.size();

Data[] data = new Data[trueSizeQueue];

for(int x=0;x<trueSizeQueue; x++)

{

Data tempData = new Data();

String temp = ( String ) queue.dequeue();

int tempInt = Integer.parseInt(temp);

tempData.setNum(tempInt);

data[x] = tempData;

}

Sort.mergeSort(data, compId);

int trueSize = data.length;

FileWriter fileWriter = new FileWriter(truePath);

BufferedWriter writer = new BufferedWriter(fileWriter);

for(int j=0;j<trueSize; j++)

{

int x = data[j].getNum();

String s = Integer.toString(x);

writer.write(s);

writer.newLine();

}

blockNum++;

writer.close();

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

}

/\*\*

\* @param path

\* @param blocks

\* @param size

\* @param blockSize

\*/

public static void mergeBlocks(String path, int blocks, int size, int blockSize)

{

Data[] data = new Data[size];

Data[] dataA = new Data[blockSize];

Data[] dataB = new Data[blockSize];

Comparator compId = new IdComparator();

try

{

int x = 0;

for(int i=0; i<blocks-1; i++)

{

String truePathA = path + i + ".txt";

String truePathB = path + (i+1) + ".txt";

File fileA = new File(truePathA);

Scanner scanLineA = new Scanner(fileA);

File fileB = new File(truePathB);

Scanner scanLineB = new Scanner(fileB);

if(i == 0)

{

while(scanLineA.hasNext())

{

String s = scanLineA.nextLine();

Data temp = new Data();

dataA[x] = temp;

x++;

}

}

x = 0;

while(scanLineB.hasNext())

{

String s = scanLineB.nextLine();

Data temp = new Data();

dataB[x] = temp;

x++;

}

scanLineA.close();

scanLineB.close();

int mergeSize = dataA.length + dataB.length;

Data[] dataT = new Data[mergeSize];

int count1 = 0, count2 = 0;

if(i == 0)

{

for(int j=0; j<mergeSize; j++)

{

if(count1 < dataA.length)

{

dataT[j] = dataA[count1];

count1++;

}

else if(count2 < dataB.length)

{

dataT[j] = dataB[count2];

count2++;

}

}

Sort.mergeSort(dataT, compId); // Sorts on id using merge sort

dataA = dataT;

}

else

{

for(int z=0; z<mergeSize; z++)

{

if(count1 < dataA.length)

{

dataT[z] = dataA[count1];

count1++;

}

else if(count2 < dataB.length)

{

dataT[z] = dataB[count2];

count2++;

}

}

Sort.mergeSort(dataT, compId); // Sorts on id using merge sort

dataA = dataT;

}

if(i == 8)

{

fileCreate(dataT, "C:\\data\\sortedData.txt");

}

}

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

/\*\*

\* @param path

\* @param splitPath

\* @param blocks

\*/

public static void split(String path, String splitPath, int blocks)

{

try

{

FileReader fileReader = new FileReader(path);

BufferedReader reader = new BufferedReader(fileReader);

String line = "";

int lineCount = 0;

int blockNum = 0;

LinkedQueue lineQueue = new LinkedQueue();

while((line = reader.readLine()) != null)

{

lineQueue.enqueue(line);

lineCount++;

if(lineCount >= blocks)

{

int trueSize = lineQueue.size();

String truePath = splitPath + blockNum + ".txt";

FileWriter fileWriter = new FileWriter(truePath);

BufferedWriter writer = new BufferedWriter(fileWriter);

for(int i=0;i<trueSize; i++)

{

String s = ( String ) lineQueue.dequeue();

writer.write(s);

writer.newLine();

}

blockNum++;

lineCount = 0;

writer.close();

}

}

reader.close();

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

}

/\*\*

\* @param path

\* @return

\*/

public static boolean isSorted(String path)

{

try

{

File file = new File(path);

Scanner scanLine = new Scanner(file);

if(!scanLine.hasNext())

{

return true;

}

while(scanLine.hasNext())

{

String aS = scanLine.nextLine();

String bS = scanLine.nextLine();

int a = Integer.parseInt(aS);

int b = Integer.parseInt(bS);

if(a > b)

{

return false;

}

}

scanLine.close();

}

catch (IOException ioe)

{

ioe.printStackTrace();

}

return true;

}

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 12/8/22

\*

\* Comparable interface

\*/

public interface Comparable <K> {

//

//Used for a comparison based on the "natural ordering"

//

// return < 0 if this.a < b

// return 0 if this.a = b

// return > 0 if this.a > b

int compareTo( K b );

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 12/8/22

\*

\* Comparator interface

\*/

public interface Comparator <K> {

int compare( K a, K b );

}

import java.util.Random;

/\*\*

\* @author Samuel Swedberg

\* @version 12/8/22

\*

\* A class that contains information about Data

\*/

public class Data {

private int num;

/\*

@param id

\*/

Data()

{

this.num = genNum();

}

private int genNum()

{

int max = Integer.MAX\_VALUE;

Random rand = new Random();

int r1 = rand.nextInt(max - 0);

int r2 = rand.nextInt(max - 0);

int r = r2 - r1;

return r;

}

public void setNum(int num) { this.num = num; }

public int getNum() { return num; }

}

/\*\*

\*

\* @author Samuel Swedberg

\* @version 12/8/22

\*

\* Comparator for ID

\*/

public class IdComparator implements Comparator<Data> {

public int compare(Data a, Data b)

{

if ( a == null || b == null )

{

throw new NullPointerException("Type cannot be null");

}

int aId = a.getNum();

int bId = b.getNum();

return Double.compare(aId, bId);

}

}

/\*\*

\* LinkedQueue Class

\* Code Fragments 6.11

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 12/8/22

\*/

/\*\* Realization of a FIFO queue as an adaptation of a SinglyLinkedList. \*/

public class LinkedQueue<E> implements Queue<E> {

private SinglyLinkedList<E> list = new SinglyLinkedList<>( ); // an empty list

public LinkedQueue( ) { } // new queue relies on the initially empty list

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

public boolean isEmpty( ) { return list.isEmpty( ); }

public void enqueue(E element) { list.addLast(element); }

public E first( ) { return list.first( ); }

public E dequeue( ) { return list.removeFirst( ); }

}

/\*\*

\* Queue Class

\* Code Fragments 6.9

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 12/8/22

\*/

public interface Queue<E> {

/\*\* Returns the number of elements in the queue. \*/

int size( );

/\*\* Tests whether the queue is empty. \*/

boolean isEmpty( );

/\*\* Inserts an element at the rear of the queue. \*/

void enqueue(E e);

/\*\* Returns, but does not remove, the first element of the queue (null if empty). \*/

E first( );

/\*\* Removes and returns the first element of the queue (null if empty). \*/

E dequeue( );

}

/\*\*

\* SinglyLinkedList Class

\* Code Fragments 3.14, 3.15

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 12/8/22

\*/

public class SinglyLinkedList<E> {

//---- nested Node class -----

private static class Node<E> {

private E element; // reference to the element stored at this node

private Node<E> next; // reference to the subsequent node in the list

public Node(E e, Node<E> n) {

element = e;

next = n;

}

public E getElement( ) { return element; }

public Node<E> getNext( ) { return next; }

public void setNext(Node<E> n) { next = n; }

}

// instance variables of the SinglyLinkedList

private Node<E> head = null; // head node of the list (or null if empty)

private Node<E> tail = null; // last node of the list (or null if empty)

private int size = 0; // number of nodes in the lis

public SinglyLinkedList() {} // constructs an initially empty list

// access methods

public int size( ) { return size; }

public boolean isEmpty( ) { return size == 0; }

public E first( ) { // returns (but does not remove) the first element

if (isEmpty( )) return null;

return head.getElement( );

}

public E last( ) { // returns (but does not remove) the last element

if (isEmpty( )) return null;

return tail.getElement( );

}

// update methods

public void addFirst(E e) { // adds element e to the front of the list

head = new Node<>(e, head); // create and link a new node

if (size == 0)

tail = head; // special case: new node becomes tail also

size++;

}

public void addLast(E e) { // adds element e to the end of the list

Node<E> newest = new Node<>(e, null); // node will eventually be the tail

if (isEmpty( ))

head = newest; // special case: previously empty list

else

tail.setNext(newest); // new node after existing tail

tail = newest; // new node becomes the tail

size++;

}

public E removeFirst( ) { // removes and returns the first element

if (isEmpty( )) return null; // nothing to remove

E answer = head.getElement( );

head = head.getNext( ); // will become null if list had only one node

size--;

if (size == 0)

tail = null; // special case as list is now empty

return answer;

}

}

import java.util.Arrays;

/\*\*

\*

\* @author Samuel Swedberg

\* @version 12/8/22

\*

\* A class that holds different sorting algorithms

\*/

public class Sort {

/\*\*

\*

\* @param <K>

\* @param S1

\* @param S2

\* @param S

\* @param comp

\*/

public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {

int i = 0, j = 0;

while (i + j < S.length) {

if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))

S[i+j] = S1[i++]; // copy ith element of S1 and increment i

else

S[i+j] = S2[j++]; // copy jth element of S2 and increment j

}

}

/\*\*

\*

\* @param <K>

\* @param S

\* @param comp

\*/

public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {

int n = S.length;

if (n < 2) return; // array is trivially sorted

// divide

int mid = n/2;

K[ ] S1 = Arrays.copyOfRange(S, 0, mid); // copy of first half

K[ ] S2 = Arrays.copyOfRange(S, mid, n); // copy of second half

// conquer (with recursion)

mergeSort(S1, comp); // sort copy of first half

mergeSort(S2, comp); // sort copy of second half

// merge results

merge(S1, S2, S, comp); // merge sorted halves back into original

}

}