Eduard Klimenko

TCSS 343 Algorithms

Martine De Cock

HW2: K-Way Merge Sort

­

Analysis:

The average time to sort was calculated from a total of 30 trials for each pair of the k and n values. Each line corresponds to a k value. We can see that as the input size increases the time to sort also increases for all values of k. In general, it seems that with higher values of k, the overall sort time decreases with the same input size. However, there difference in time seems to fall off after k at 20 because the sort time is almost identical to k at 50. We can see that with k at 2 it takes 2050 ms to sort an input of 3.2 million but only takes 1025 ms with k = 50. This could be due to the fact that the each of the subarrays are smaller which affects linkedlist efficiency. As this is only tested on my personal machine the results could be very different for other machines. The run time is still O (n log n) but with a multiplier in front (efficiency slows as k increases; any k > 20 will have little effect on run time compared to lower values of k).

Problem Approach:

To approach this problem I first began by reading section 5.1 in the text which went over the Merge Sort algorithm. I then fully read the homework instructions and then reread them again! I then wanted to conceptualize what I was supposed to do on paper before I tried to implement it in Java (see attached paper). After realizing that k-way is basically the same as the basic merge sort with the difference being that we split the array into k subarrays not just 2. We still grab the first element from each subarray so that stays the same. And since were are looking at the smallest value we can pull the first element from each subarray into a min heap and just grab the root (this is the same as comparing all first values of the subarrays). We then place the smallest value into the sorted list and replace the removed value from the heap with the next value of the subarray from which it was taken from (if it exists). With this thought proccess I decided to use LinkedLists as subarrays because I could avoid the possible out of bounds errors and the checks that would need to go with them by simply calling the built in function .pollFirst() and checking if it was not null. It was hard at first using all the given code because I had to look through each class and see what methods were available to me. I had to look online to see the general code to declare things such as array of LL, heaps, and using nodes as pointers. I was stuck for a long time until I remembered there was the method getWhichSubarray(). I also had some issue with generic types, casting, and throwing exceptions (I had to add a throws exception to every method since it was recursive). I modified the test methods to taking in k, n, and number of trials arguments to automate the proccess more easily. I then transferred the data to Excel and created a graph. Overall, the assignment was pretty difficult but a good challenge.

Modified Code (without comments for readability):

public static void merge(int[] data, int low, int high, int k) throws EmptyHeapException {

if (high < low + k) {

// the subarray has k or fewer elements

// just make one big heap and do deleteMins on it

Comparable[] subarray = new MergesortHeapNode[high - low + 1];

for (int i = 0, j = low; i < subarray.length; i++, j++) {

subarray[i] = new MergesortHeapNode(data[j], 0);

}

BinaryHeap heap = BinaryHeap.buildHeap(subarray);

for (int j = low; j <= high; j++) {

try {

data[j] = ((MergesortHeapNode) heap.deleteMin()).getKey();

} catch (EmptyHeapException e) {

System.out.println("Tried to delete from an empty heap.");

}

}

} else {

LinkedList<MergesortHeapNode>[] subarray = new LinkedList[k];

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

subarray[i] = new LinkedList<MergesortHeapNode>();

BinaryHeap heap = new BinaryHeap();

for (int i = 0; i < k; i++) {

for (int j = low + i \* (high - low + 1) / k;

j < low + (i + 1) \* (high - low + 1) / k; j++) {

subarray[i].add(new MergesortHeapNode(data[j], i));

}

heap.insert(subarray[i].pollFirst());

}

MergesortHeapNode curNode;

for (int i = low; i <= high; i++) {

curNode = (MergesortHeapNode) heap.deleteMin();

data[i] = (curNode).getKey();

curNode = subarray[curNode.getWhichSubarray()].pollFirst();

if (curNode != null) {

heap.insert(curNode);

}

}

}

}

private static void testCorrectness(int k, int n) throws EmptyHeapException {

int[] data = getRandomPermutationOfIntegers(n);

// for (int i = 0; i < data.length; i++) {

// System.out.println("data[" + i + "] = " + data[i]);

// }

// int k = 100;

kwayMergesort(data, k);

// verify that data[i] = i

for (int i = 0; i < data.length; i++) {

if (data[i] != i) {

System.out.println("Error! data[" + i + "] = " + data[i] + ".");

}

}

}

private static void testTiming(int k, int n, int trials) throws EmptyHeapException {

double avgTime = 0;

System.out.println("\*\* Results for k-way mergesort:");

System.out.println(" " + "n = " + n + " " + "k = " + k);

for (int i = 0; i < trials; i++) {

// timer variables

long totalTime = 0;

long startTime = 0;

long finishTime = 0;

// start the timer

Date startDate = new Date();

startTime = startDate.getTime();

int[] data = getRandomArrayOfIntegers(n);

kwayMergesort(data, k);

// stop the timer

Date finishDate = new Date();

finishTime = finishDate.getTime();

totalTime += (finishTime - startTime);

avgTime += totalTime;

}

System.out.println(" " + "Average Time: " + (avgTime / trials) + " ms.");

}

public static void main(String[] argv) throws EmptyHeapException {

int numTrials = 30;

int[] kValues = { 2, 3, 5, 10, 20, 50 };

for (int j = 0; j < kValues.length; j++) {

for (int i = 200000; i <= 3200000; i = i \* 2) {

testCorrectness(kValues[j], i);

testTiming(kValues[j], i, numTrials);

}

System.out.println("");

}

}

}