CSCI 421 Design and Analysis of Algorithms Spring 2019

Lecture 2 Activity 3

1. Randomized quicksort. Modify partition() so that it always chooses the partitioning item uniformly at random from the array (instead of shuffling the array initially). Compare the performance against the original version of quicksort. Attach your code in plaintext and screenshots of your output here.

///////Lec2Act3\_1.java

import java.io.\*;

import java.util.\*;

import java.util.Random;

public class Lec2Act3\_1 {

public void printArray(int arr[]) {

int n = arr.length;

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

System.out.print(arr[i] + " ");

System.out.println();

}

public static void main(String[] args) {

int size = 8000;

int arr[] = new int[size];

int arr2[] = new int[size];

int item = 0;

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

item = (int) (Math.random() \* 100);

arr[i] = item;

}

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

arr2[i] = arr[i];

}

quicksort qs = new quicksort();

quicksortImproved qsi = new quicksortImproved();

long startTime, endTime, elapsed;

System.out.println("For Array Size: " + size);

startTime = System.currentTimeMillis();

qs.sort(arr);

endTime = System.currentTimeMillis();

elapsed = endTime - startTime;

System.out.println("QuickSort Time: " + elapsed);

startTime = System.currentTimeMillis();

qsi.sort(arr2);

endTime = System.currentTimeMillis();

elapsed = endTime - startTime;

System.out.println("QuickSort Improved Time: " + elapsed);

}

}

/////quicksort.java

import java.util.Random;

public class quicksort {

int partition(int a[], int lo, int hi) {

int i = lo, j = hi + 1;

while (true) {

while (less(a[++i], a[lo]))

if (i == hi)

break;

while (less(a[lo], a[--j]))

if (j == lo)

break;

if (i >= j)

break;

exch(a, i, j);

}

exch(a, lo, j);

return j;

}

void sort(int a[], int lo, int hi) {

if (hi <= lo)

return;

int j = partition(a, lo, hi);

sort(a, lo, j - 1);

sort(a, j + 1, hi);

}

public void sort(int a[]) {

sort(a, 0, a.length - 1);

}

private static void exch(int[] a, int i, int j) {

int swap = a[i];

a[i] = a[j];

a[j] = swap;

}

private static boolean less(Comparable v, Comparable w) {

return v.compareTo(w) < 0;

}

}

//////quicksortImproved.java

import java.util.Random;

public class quicksortImproved {

Random rand = new Random();

int partition(int a[], int lo, int hi) {

int pivot = a[hi];

int i = lo - 1;

for (int j = lo; j < hi; j++) {

if (less(a[j], pivot)) {

i++;

exch(a, i, j);

}

}

i++;

exch(a, i, hi);

return i;

}

void sort(int[] arr, int lo, int hi) {

if (hi <= lo)

return;

int randPiv = partition(arr,lo + rand.nextInt(hi - lo + 1),hi);

sort(arr, lo, randPiv - 1);

sort(arr, randPiv + 1, hi);

}

public void sort(int[] a) {

int lo = 0;

int hi = a.length - 1;

sort(a, lo, hi);

}

private static void exch(int[] a, int i, int j) {

int swap = a[i];

a[i] = a[j];

a[j] = swap;

}

private static boolean less(Comparable v, Comparable w) {

return v.compareTo(w) < 0;

}

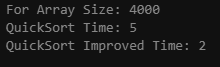
}

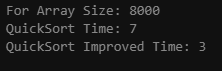
A close up of a sign

Description automatically generated

A close up of a sign

Description automatically generated





1. Write a performance driver client program that uses “insert” to fill a priority queue, then uses “remove the maximum” to remove half the keys, then uses insert to fill it up again, then uses remove the maximum to remove all the keys, doing so multiple times on random sequences of keys of various lengths ranging from small to large; measures the time taken for each run; and prints out or plots the average running times.

import java.util.Comparator;

import java.util.Iterator;

import java.util.NoSuchElementException;

import java.util.stream.IntStream;

import java.util.Random;

public class Lec2Act3\_2<Key> implements Iterable<Key> {

private Key[] pq;

private int n;

private Comparator<Key> comparator;

public Lec2Act3\_2(int initCapacity) {

pq = (Key[]) new Object[initCapacity + 1];

n = 0;

}

public Lec2Act3\_2() {

this(1);

}

public Lec2Act3\_2(int initCapacity, Comparator<Key> comparator) {

this.comparator = comparator;

pq = (Key[]) new Object[initCapacity + 1];

n = 0;

}

public Lec2Act3\_2(Comparator<Key> comparator) {

this(1, comparator);

}

public Lec2Act3\_2(Key[] keys) {

n = keys.length;

pq = (Key[]) new Object[keys.length + 1];

IntStream.range(0, n).forEach(i -> pq[i + 1] = keys[i]);

IntStream.iterate(n / 2, k -> k >= 1, k -> k - 1).forEach(this::sink);

assert isMaxHeap();

}

public boolean isEmpty() {

return n == 0;

}

public int size() {

return n;

}

public Key max() {

if (isEmpty())

throw new NoSuchElementException("Priority queue underflow");

return pq[1];

}

private void resize(int capacity) {

assert capacity > n;

Key[] temp = (Key[]) new Object[capacity];

for (int i = 1; i <= n; i++) {

temp[i] = pq[i];

}

pq = temp;

}

public void insert(Key x) {

if (n == pq.length - 1)

resize(2 \* pq.length);

pq[++n] = x;

swim(n);

assert isMaxHeap();

}

public Key delMax() {

if (isEmpty())

throw new NoSuchElementException("Priority queue underflow");

Key max = pq[1];

exch(1, n--);

sink(1);

pq[n + 1] = null;

if ((n > 0) && (n == (pq.length - 1) / 4))

resize(pq.length / 2);

assert isMaxHeap();

return max;

}

private void swim(int k) {

while (k > 1 && less(k / 2, k)) {

exch(k, k / 2);

k = k / 2;

}

}

private void sink(int k) {

while (2 \* k <= n) {

int j = 2 \* k;

if (j < n && less(j, j + 1))

j++;

if (!less(k, j))

break;

exch(k, j);

k = j;

}

}

private boolean less(int i, int j) {

if (comparator == null) {

return ((Comparable<Key>) pq[i]).compareTo(pq[j]) < 0;

} else {

return comparator.compare(pq[i], pq[j]) < 0;

}

}

private void exch(int i, int j) {

Key swap = pq[i];

pq[i] = pq[j];

pq[j] = swap;

}

private boolean isMaxHeap() {

return isMaxHeap(1);

}

private boolean isMaxHeap(int k) {

if (k > n)

return true;

int left = 2 \* k;

int right = 2 \* k + 1;

if (left <= n && less(k, left))

return false;

if (right <= n && less(k, right))

return false;

return isMaxHeap(left) && isMaxHeap(right);

}

public Iterator<Key> iterator() {

return (Iterator<Key>) new HeapIterator();

}

private class HeapIterator implements Iterator<int[]> {

private Lec2Act3\_2<int[]> copy;

public HeapIterator() {

if (comparator == null)

copy = new Lec2Act3\_2<>(size());

else {

copy = (Lec2Act3\_2<int[]>) new Lec2Act3\_2<Key>(size(), comparator);

}

IntStream.rangeClosed(1, n).forEach(i -> copy.insert((int[]) pq[i]));

}

public boolean hasNext() {

return !copy.isEmpty();

}

public void remove() {

throw new UnsupportedOperationException();

}

public int[] next() {

if (!hasNext())

throw new NoSuchElementException();

return copy.delMax();

}

}

public static void main(String[] args) {

Lec2Act3\_2<Integer> pq = new Lec2Act3\_2<Integer>();

int itCount = 4; // Iteration count...ie (4 = 8000), since it will be 1000 \* 2, 2000\*2

long startTime, endTime, elapsed;

int N = 1000;

long totalTime = 0;

while (itCount > 0) {

startTime = System.currentTimeMillis();

int i = 0;

Random rand = new Random();

while (i < N) {

int item = rand.nextInt(100);

pq.insert(item);

i++;

}

i = 0;

while (i < N / 2) {

pq.delMax();

i++;

}

endTime = System.currentTimeMillis();

elapsed = endTime - startTime;

totalTime = totalTime + elapsed;

System.out.println("\nArray Size: " + N);

System.out.println("Time = " + elapsed + " Milliseconds");

N = N \* 2;

itCount--;

}

System.out.println("===============================================");

System.out.println("Average TIme \t= " + totalTime / 4 + " Milliseconds");

}

}

A screenshot of a cell phone

Description automatically generated