Exercise 5: Java, Memory Management, and Polymorphism with Interfaces

# Exercise 5.1: Java

public IntList reverse() {

IntList rList = new IntList(null); // new empty list

ConsCell i = start; // grab first index of list

while (i != null) { // iterate list to push on new list

rList = rList.cons(i.getHead()); // prepend old list onto new list

i = i.getTail(); // get next

}

return rList;

}

# Exercise 5.2: Memory Management

public class BFHM {

static private final int NULL = -1; // Null link

public int[] memory; // Managed memory

private int freeStart; // Start of free list

public BFHM(int[] initialMemory) {

memory = initialMemory;

memory[0] = memory.length; // One big free block

memory[1] = NULL; // End of free list

freeStart = 0; // Start of free list

}

public int allocate(int requestSize) {

int size = requestSize + 1; // Size with header

int p = freeStart; // Head of free list

int lag = NULL; // Previous block

int bestFit = NULL; // Best-fit block

int bestFitLag = NULL; // Previous best-fit block

int bestFitSize = Integer.MAX\_VALUE; // Size of best-fit block

// Best-fit search: find the smallest block that fits

while (p != NULL) {

if (memory[p] >= size && memory[p] < bestFitSize) {

bestFit = p; // New best-fit block

bestFitLag = lag; // Previous best-fit block

bestFitSize = memory[p]; // Current best-fit block size

}

lag = p; // Update previous block

p = memory[p + 1]; // Move to next block

}

if (bestFit == NULL) // No block large enough

throw new OutOfMemoryError();

int nextFree = memory[bestFit + 1]; // Block after best-fit block

int unused = memory[bestFit] - size; // Extra space

if (unused > 1) { // More than a header's worth

nextFree = bestFit + size; // Index of unused piece

memory[nextFree] = unused; // Set size of unused piece

memory[nextFree + 1] = memory[bestFit + 1]; // Link unused piece

memory[bestFit] = size; // Reduce size of best-fit block

}

if (bestFitLag == NULL)

freeStart = nextFree; // Update freeStart

else

memory[bestFitLag + 1] = nextFree; // Update link of previous best-fit

return bestFit + 1; // Usable word index

}

public void deallocate(int address) {

int addr = address - 1;

memory[addr + 1] = freeStart; // Link to free list

freeStart = addr; // Update freeStart

}

public static void main(String[] args) {

BFHM mm = new BFHM(new int[7]);

int a = mm.allocate(2);

int b = mm.allocate(1);

int c = mm.allocate(1);

mm.deallocate(a);

mm.deallocate(c);

int d = mm.allocate(1); // Allocates in the smallest free block

int e = mm.allocate(2); // Allocates in the remaining space

}

}

# Exercise 5.3: Polymorphism with Java Interfaces

Navigator:

public class Navigator {

public void navigate(Drivable d) {

d.start(); // invoke Drivable Method 1

d.stop(); // invoke Drivable Method 2

}

}

A:

public class A implements Drivable {

@Override

public void start() {

System.out.println("A is starting");

}

@Override

public void accelerate() {

System.out.println("A is accelerating");

}

@Override

public void stop() {

System.out.println("A is stopping");

}

@Override

public void turnLeft() {

System.out.println("A is turning left");

}

@Override

public void turnRight() {

System.out.println("A is turning right");

}

}

B:

public class B implements Drivable {

@Override

public void start() {

System.out.println("B is starting");

}

@Override

public void accelerate() {

System.out.println("B is accelerating");

}

@Override

public void stop() {

System.out.println("B is stopping");

}

@Override

public void turnLeft() {

System.out.println("B is turning left");

}

@Override

public void turnRight() {

System.out.println("B is turning right");

}

}

Main:

public class Main {

public static void main(String[] args) {

Drivable a = new A();

Drivable b = new B();

Navigator navigator = new Navigator();

navigator.navigate(a);

navigator.navigate(b);

}

}

Output:

A is starting

A is stopping

B is starting

B is stopping