# Tölvunarfræði 2 – Verkefni 2

1. [Hlaðar] Í kennslubókinni eru sýndar tvær útfærslur á hlaða: ResizingArrayStack.java, sem notar fylki af breytilegri stærð, og Stack.java, sem notar tengdan lista. Báðir klasarnir útfæra aðferðina peek() sem skilar efsta staki hlaðans án þess að taka það af honum.
2. Bætið aðferðinni við [ResizingArrayStack.java](https://algs4.cs.princeton.edu/13stacks/ResizingArrayStack.java.html). Aðferðin tekur inn heiltöluna og skilar k-ta efsta stakinu á hlaðanum (ef k er 0 þá er skilað efsta stakinu, ef k er 1 þá næstefsta, o.s.frv.) án þess að breyta hlaðanum. Aðferðin á að hegða sér á svipaðan hátt og aðferðin gerir þegar það eru ekki nægilega mörg stök í hlaðanum, þ.e. kalla á frávikið .
3. Bætið aðferðinni líka við hina útfærsluna, [Stack.java.](https://algs4.cs.princeton.edu/13stacks/Stack.java.html) Hér er notaður tengdur listi, svo útfærslan verður töluvert ólík þeirri í a-lið.

Breytið svo main-fallinu í báðum útfærslum þannig að það lesi bara inn runu af

strengjum og prenti svo út öll stökin í hlaðanum með því að nota þar sem

gengur frá 0 uppí fjöldi staka í hlaðanum. Þið ættuð að fá frávik í síðasta peek-kallinu.

Skilið inn aðferðunum tveimur ásamt main-fallinu og einu skjáskoti af keyrslu. Vísbending: Þið þurfið væntanlega að bæta inn skipuninni "import edu.princeton.cs.algs4.\*;" í báðar útfærslurnar í upphafi.

**SVAR:**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac ResizingArrayStack.java

\* Execution: java ResizingArrayStack < input.txt

\* Dependencies: StdIn.java StdOut.java

\* Data files: https://algs4.cs.princeton.edu/13stacks/tobe.txt

\*

\* Stack implementation with a resizing array.

\*

\* % more tobe.txt

\* to be or not to - be - - that - - - is

\*

\* % java ResizingArrayStack < tobe.txt

\* to be not that or be (2 left on stack)

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Iterator;

import java.util.NoSuchElementException;

import edu.princeton.cs.algs4.\*;

/\*\*

\* The {@code ResizingArrayStack} class represents a last-in-first-out (LIFO)

\* stack

\* of generic items.

\* It supports the usual <em>push</em> and <em>pop</em> operations, along with

\* methods

\* for peeking at the top item, testing if the stack is empty, and iterating

\* through

\* the items in LIFO order.

\* <p>

\* This implementation uses a resizing array, which double the underlying array

\* when it is full and halves the underlying array when it is one-quarter full.

\* The <em>push</em> and <em>pop</em> operations take constant amortized time.

\* The <em>size</em>, <em>peek</em>, and <em>is-empty</em> operations takes

\* constant time in the worst case.

\* <p>

\* For additional documentation,

\* see <a href="https://algs4.cs.princeton.edu/13stacks">Section 1.3</a> of

\* <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.

\*

\* @author Robert Sedgewick

\* @author Kevin Wayne

\*/

public class ResizingArrayStack<Item> implements Iterable<Item> {

// initial capacity of underlying resizing array

private static final int INIT\_CAPACITY = 8;

private Item[] a; // array of items

private int n; // number of elements on stack

/\*\*

\* Initializes an empty stack.

\*/

public ResizingArrayStack() {

a = (Item[]) new Object[INIT\_CAPACITY];

n = 0;

}

/\*\*

\* Is this stack empty?

\*

\* @return true if this stack is empty; false otherwise

\*/

public boolean isEmpty() {

return n == 0;

}

/\*\*

\* Returns the number of items in the stack.

\*

\* @return the number of items in the stack

\*/

public int size() {

return n;

}

// resize the underlying array holding the elements

private void resize(int capacity) {

assert capacity >= n;

// textbook implementation

Item[] copy = (Item[]) new Object[capacity];

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

copy[i] = a[i];

}

a = copy;

// alternative implementation

// a = java.util.Arrays.copyOf(a, capacity);

}

/\*\*

\* Adds the item to this stack.

\*

\* @param item the item to add

\*/

public void push(Item item) {

if (n == a.length)

resize(2 \* a.length); // double size of array if necessary

a[n++] = item; // add item

}

/\*\*

\* Removes and returns the item most recently added to this stack.

\*

\* @return the item most recently added

\* @throws java.util.NoSuchElementException if this stack is empty

\*/

public Item pop() {

if (isEmpty())

throw new NoSuchElementException("Stack underflow");

Item item = a[n - 1];

a[n - 1] = null; // to avoid loitering

n--;

// shrink size of array if necessary

if (n > 0 && n == a.length / 4)

resize(a.length / 2);

return item;

}

/\*\*

\* Returns (but does not remove) the item most recently added to this stack.

\*

\* @return the item most recently added to this stack

\* @throws java.util.NoSuchElementException if this stack is empty

\*/

public Item peek(int k) {

if (k < 0 || k >= n)

throw new NoSuchElementException("Index out of bounds");

return a[n - k - 1];

}

/\*\*

\* Returns an iterator to this stack that iterates through the items in LIFO

\* order.

\*

\* @return an iterator to this stack that iterates through the items in LIFO

\* order.

\*/

public Iterator<Item> iterator() {

return new ReverseArrayIterator();

}

// a array iterator, in reverse order

private class ReverseArrayIterator implements Iterator<Item> {

private int i;

public ReverseArrayIterator() {

i = n - 1;

}

public boolean hasNext() {

return i >= 0;

}

public Item next() {

if (!hasNext())

throw new NoSuchElementException();

return a[i--];

}

}

/\*\*

\* Unit tests the {@code Stack} data type.

\*

\* @param args the command-line arguments

\*/

public static void main(String[] args) {

ResizingArrayStack<String> stack = new ResizingArrayStack<String>(); // Or Stack<String> for Stack.java

while (!StdIn.isEmpty()) {

String item = StdIn.readString();

stack.push(item);

}

try {

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

StdOut.println(stack.peek(i));

}

} catch (NoSuchElementException e) {

StdOut.println("NoSuchElementException caught");

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac Stack.java

\* Execution: java Stack < input.txt

\* Dependencies: StdIn.java StdOut.java

\* Data files: https://algs4.cs.princeton.edu/13stacks/tobe.txt

\*

\* A generic stack, implemented using a singly linked list.

\* Each stack element is of type Item.

\*

\* This version uses a static nested class Node (to save 8 bytes per

\* Node), whereas the version in the textbook uses a non-static nested

\* class (for simplicity).

\*

\* % more tobe.txt

\* to be or not to - be - - that - - - is

\*

\* % java Stack < tobe.txt

\* to be not that or be (2 left on stack)

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Iterator;

import java.util.NoSuchElementException;

import edu.princeton.cs.algs4.\*;

import java.util.Scanner;

/\*\*

\* The {@code Stack} class represents a last-in-first-out (LIFO) stack of

\* generic items.

\* It supports the usual <em>push</em> and <em>pop</em> operations, along with

\* methods

\* for peeking at the top item, testing if the stack is empty, and iterating

\* through

\* the items in LIFO order.

\* <p>

\* This implementation uses a singly linked list with a static nested class for

\* linked-list nodes. See {@link LinkedStack} for the version from the

\* textbook that uses a non-static nested class.

\* See {@link ResizingArrayStack} for a version that uses a resizing array.

\* The <em>push</em>, <em>pop</em>, <em>peek</em>, <em>size</em>, and

\* <em>is-empty</em>

\* operations all take constant time in the worst case.

\* <p>

\* For additional documentation,

\* see <a href="https://algs4.cs.princeton.edu/13stacks">Section 1.3</a> of

\* <i>Algorithms, 4th Edition</i> by Robert Sedgewick and Kevin Wayne.

\*

\* @author Robert Sedgewick

\* @author Kevin Wayne

\*

\* @param <Item> the generic type each item in this stack

\*/

public class Stack<Item> implements Iterable<Item> {

private Node<Item> first; // top of stack

private int n; // size of the stack

// helper linked list class

private static class Node<Item> {

private Item item;

private Node<Item> next;

}

/\*\*

\* Initializes an empty stack.

\*/

public Stack() {

first = null;

n = 0;

}

/\*\*

\* Returns true if this stack is empty.

\*

\* @return true if this stack is empty; false otherwise

\*/

public boolean isEmpty() {

return first == null;

}

/\*\*

\* Returns the number of items in this stack.

\*

\* @return the number of items in this stack

\*/

public int size() {

return n;

}

/\*\*

\* Adds the item to this stack.

\*

\* @param item the item to add

\*/

public void push(Item item) {

Node<Item> oldfirst = first;

first = new Node<Item>();

first.item = item;

first.next = oldfirst;

n++;

}

/\*\*

\* Removes and returns the item most recently added to this stack.

\*

\* @return the item most recently added

\* @throws NoSuchElementException if this stack is empty

\*/

public Item pop() {

if (isEmpty())

throw new NoSuchElementException("Stack underflow");

Item item = first.item; // save item to return

first = first.next; // delete first node

n--;

return item; // return the saved item

}

/\*\*

\* Returns (but does not remove) the item most recently added to this stack.

\*

\* @return the item most recently added to this stack

\* @throws NoSuchElementException if this stack is empty

\*/

public Item peek() {

if (isEmpty())

throw new NoSuchElementException("Stack underflow");

return first.item;

}

/\*\*

\* Returns a string representation of this stack.

\*

\* @return the sequence of items in this stack in LIFO order, separated by

\* spaces

\*/

public String toString() {

StringBuilder s = new StringBuilder();

for (Item item : this) {

s.append(item);

s.append(' ');

}

return s.toString();

}

/\*\*

\* Returns an iterator to this stack that iterates through the items in LIFO

\* order.

\*

\* @return an iterator to this stack that iterates through the items in LIFO

\* order

\*/

public Iterator<Item> iterator() {

return new LinkedIterator(first);

}

// the iterator

private class LinkedIterator implements Iterator<Item> {

private Node<Item> current;

public LinkedIterator(Node<Item> first) {

current = first;

}

// is there a next item?

public boolean hasNext() {

return current != null;

}

// returns the next item

public Item next() {

if (!hasNext())

throw new NoSuchElementException();

Item item = current.item;

current = current.next;

return item;

}

}

/\*\*

\* Unit tests the {@code Stack} data type.

\*

\* @param args the command-line arguments

\*/

public static void main(String[] args) {

Stack<String> stack = new Stack<String>();

Scanner scanner = new Scanner(System.in);

// Read strings from standard input until there is no more input

while (scanner.hasNext()) {

String item = scanner.next();

stack.push(item);

}

scanner.close();

// Print and remove each item from the stack

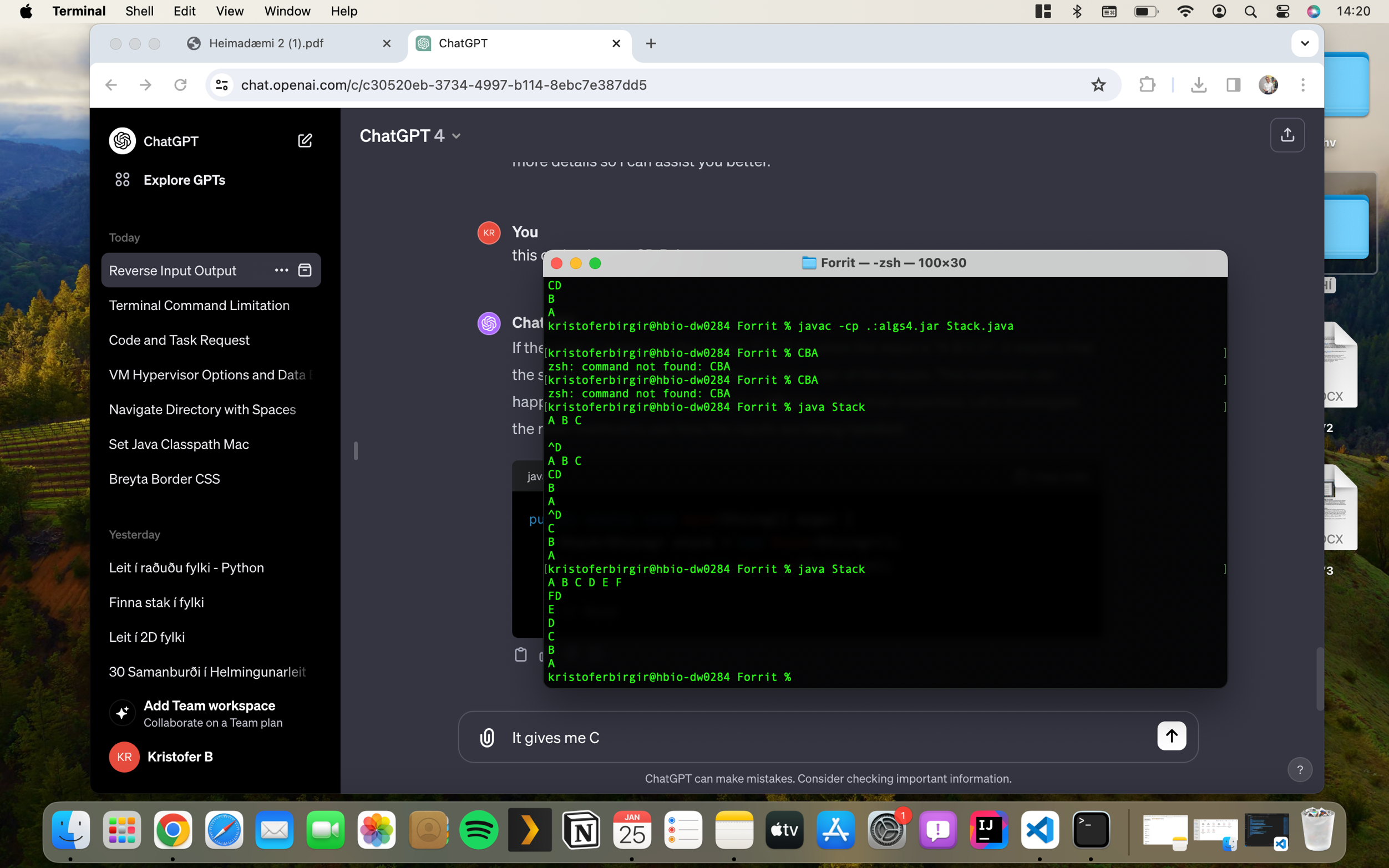
while (!stack.isEmpty()) {

System.out.println(stack.pop());

}

}

}



D er þarna inni útaf þetta er control D til að keyra.

A screenshot of a computer

Description automatically generated

a)