**1)**

public class File implements FSTree {

private int size; // invariant size >= 0

public File(int size) {

if(size<0)

throw new IllegalArgumentException("File size cannot be negative");

this.size = size;

}

@Override

public <T> T accept(Visitor<T> v) {

return v.visitFile(size);

}

}

public class Folder implements FSTree {

private final List<FSTree> children = new LinkedList<>();

public Folder(FSTree... children) {

for (var node : children)

this.children.add(requireNonNull(node));

}

@Override

public <T> T accept(Visitor<T> v) {

return v.visitFolder(children);

}

}

// conta tutti i file con size > minSize

public class FilesGreater implements Visitor<Integer> {

private final int minSize; // puo` essere negativo

public FilesGreater(int minSize) {

this.minSize = minSize;

}

@Override

public Integer visitFile(int size) {

return size > minSize ? 1 : 0;

}

@Override

public Integer visitFolder(List<FSTree> children) {

var res = 0;

for (var node : children)

res += node.accept(this);

return res;

}

}

**2)**

public class Leaf extends BSTree {

public Leaf(int value) {

super(value);

}

@Override

public <T> T accept(Visitor<T> v) {

return v.visitLeaf(value);

}

}

public class Node extends BSTree {

private final BSTree left, right;

public Node(int value, BSTree left, BSTree right) {

super(value);

if(left==null && right==null) throw new NullPointerException();

this.left = left;

this.right = right;

}

@Override

public <T> T accept(Visitor<T> v) {

return v.visitNode(value, left, right);

}

}

public class Search implements Visitor<Boolean> {

private final int searchedValue;

public Search(int searchedValue) {

this.searchedValue = searchedValue;

}

@Override

public Boolean visitLeaf(int value) {

return value == searchedValue;

}

@Override

public Boolean visitNode(int value, BSTree left, BSTree right) {

if (searchedValue < value)

return left != null && left.accept(this);

else

return value == searchedValue || right != null && right.accept(this);

OPPURE:

if (value > SearchedValue && left != null) return left.accept(this);

if (value < SearchedValue && right != null) return right.accept(this);

return value == searchedValue;

**3)**

class CharSeqIterator implements Iterator<Character>, Iterable<Character> {

private final CharSequence charSeq; // invariant charSeq != null

private final int length;

private int index;

public CharSeqIterator(CharSequence charSeq) {

this.charSeq = requireNonNull(charSeq);

length = charSeq.length();

}

@Override

public boolean hasNext() {

return index < length;

}

@Override

public Character next() {

if (!hasNext())

throw new NoSuchElementException();

return charSeq.charAt(index++);

}

@Override

public Iterator<Character> iterator() {

return this;

}

}

**4)**

class RangeIterator implements Iterator<Integer>, Iterable<Integer> {

private int next;

private final int end, step; // invariant step!=0

// ranges from start (inclusive) to end (exclusive) with step!=0

public RangeIterator(int start, int end, int step) {

if (step == 0)

throw new IllegalArgumentException("Step cannot be 0");

this.next = start;

this.end = end;

this.step = step;

}

// ranges from start (inclusive) to end (exclusive) with step 1

public RangeIterator(int start, int end) {

this(start, end, 1);

}

// ranges from 0 (inclusive) to end (exclusive) with step 1

public RangeIterator(int end) {

this(0, end);

}

@Override

public boolean hasNext() {

return step > 0 ? next < end : next > end;

}

@Override

public Integer next() {

if (!hasNext())

throw new NoSuchElementException();

var res = next;

next += step;

return res;

}

@Override

public Iterator<Integer> iterator() {

return this;

}

}

**5)**

public class PairImp<T1, T2> implements Pair<T1, T2> {

public final T1 first;

public final T2 second;

public PairImp(T1 first, T2 second) {

this.first = requireNonNull(first);

this.second = requireNonNull(second);

}

public T1 getFirst() {

return first;

}

public T2 getSecond() {

return second;

}

@Override

public String toString() {

return "(" + first + "," + second + ")";

}

}

public class Zipper<T1, T2> implements Iterator<Pair<T1, T2>> {

private final Iterator<T1> iterator1;

private final Iterator<T2> iterator2;

public Zipper(Iterable<T1> iterable1, Iterable<T2> iterable2) {

iterator1 = iterable1.iterator();

iterator2 = iterable2.iterator();

}

@Override

public boolean hasNext() {

return iterator1.hasNext() && iterator2.hasNext();

}

@Override

public Pair<T1, T2> next() {

if (!hasNext())

throw new NoSuchElementException();

return new PairImp<>(iterator1.next(), iterator2.next());

}

}

**6)**

// valori primitivi generici

public abstract class PrimVal<T> implements Value {

protected T val;

// invariante di classe: val!=null

protected PrimVal(T val) {

this.val = requireNonNull(val);

}

}

public class StringVal extends PrimVal<String> {

protected StringVal(String val) { super(val); }

@Override

public String asString() { return val; }

}

public class IntVal extends PrimVal<Integer> {

protected IntVal(Integer val) { super(val); }

@Override

public int asInt() { return val; }

}

public class Eval implements Visitor<Value> {

@Override

public Value visitIntLit(int val) {

return new IntVal(val);

}

@Override

public Value visitStringLit(String val) {

return new StringVal(val);

}

@Override

public Value visitTimes(Exp left, Exp right) {

// left si deve valutare in una stringa

// right si deve valutare in un intero

String s = left.accept(this).asString();

int i = right.accept(this).asInt();

return new StringVal(s.repeat(i));

}

}

**7)**

public class PosLit implements Exp { // nodi AST per literal interi positivi

private final int val;

// precondizione: val > 0

public PosLit(int val) {

if (val <= 0)

throw new IllegalArgumentException("Literal must be positive");

this.val = val;

}

@Override

public <T> T accept(Visitor<T> visitor) {

return visitor.visitNatLit(val);

}

}

public class Pow implements Exp { // nodi AST per elevamento a potenza

private final Exp left; // base

private final Exp right; // potenza

// precondizione: left!=null, right!=null

public Pow(Exp left, Exp right) {

this.left = requireNonNull(left);

this.right = requireNonNull(right);

}

@Override

public <T> T accept(Visitor<T> visitor) {

return visitor.visitPow(left, right);

}

}

public class Eval implements Visitor<Integer> {

@Override

public Integer visitNatLit(int val) {

return val;

}

@Override

public Integer visitPow(Exp left, Exp right) {

int base = left.accept(this);

int exp = right.accept(this);

int res;

for (res = 1; exp > 0; exp--)

res \*= base;

return res;

}

}

**8)**

public class Powers implements Iterator<Integer>, Iterable<Integer> {

private final int base; // base dell'esponente

private int items; // numero di elementi ancora da generare

private int next; // prossimo elemento da restituire

// precondizione: items >= 0

public Powers(int base, int items) {

if (items < 0)

throw new IllegalArgumentException();

this.base = this.next = base;

this.items = items;

}

@Override

public boolean hasNext() {

return items > 0;

}

@Override

public Integer next() {

if (!hasNext())

throw new NoSuchElementException();

int res = next;

next \*= base;

items--;

return res;

}

// restituisce se stesso

@Override

public Iterator<Integer> iterator() {

return this;

}

}

**9)**

public class Multiples implements Iterator<Integer>, Iterable<Integer> {

private final int step;

private final int max;

private int next;

public Multiples(int step, int items) {

if (step <= 0 || items < 0)

throw new IllegalArgumentException();

this.step = this.next = step;

this.max = step \* items;

}

@Override

public boolean hasNext() {

return next <= max;

}

@Override

public Integer next() {

if (!hasNext())

throw new NoSuchElementException();

int res = next;

next += step;

return res;

}

@Override

public Iterator<Integer> iterator() {

return this;

}

}

**10)**

public class BoolLit implements AST {

private final boolean value;

public BoolLit(boolean value) {

this.value = value;

}

@Override

public <T> T accept(Visitor<T> v) {

return v.visitBoolLit(value);

}

}

public class And implements AST {

private final AST left, right;

public And(AST left, AST right) {

this.left = requireNonNull(left);

this.right = requireNonNull(right);

}

@Override

public <T> T accept(Visitor<T> v) {

return v.visitAnd(left, right);

}

}

public class Eval implements Visitor<Boolean> {

@Override

public Boolean visitBoolLit(boolean b) {

return b;

}

@Override

public Boolean visitAnd(AST left, AST right) {

return left.accept(this) && right.accept(this);

}

}

public class ToString implements Visitor<String> {

@Override

public String visitBoolLit(boolean b) {

return String.valueOf(b);

}

@Override

public String visitAnd(AST left, AST right) {

return left.accept(this) + " " + right.accept(this) + " &&";

}

}

**11)**

public class FilteredIterator<E> implements Iterator<E> {

private final Predicate<E> pred;

private final ArrayList<E> list;

private int curr;

public FilteredIterator(Predicate<E> pred, ArrayList<E> list) {

this.pred = requireNonNull(pred);

this.list = requireNonNull(list);

}

@Override

public boolean hasNext() {

while (curr < list.size()) {

if (pred.test(list.get(curr)))

return true;

curr++;

}

return false;

}

@Override

public E next() {

if (!hasNext())

throw new NoSuchElementException();

return list.get(curr++);

}

}

public class Find {

public static <E> E find(Predicate<E> pred, ArrayList<E> list) {

return new FilteredIterator<>(pred, list).next();

}

}