Programm- & Systemverifikation

Assignment 2

Georg Weissenbacher

184.741

Part 1 of Assignment 2

- Derive test cases for the insertion function of a **balanced (AVL) binary search tree**.

Part 1 of Assignment 2

- Derive test cases for the insertion function of a **balanced (AVL) binary search tree**.
- Use the following techniques:
  a) Equivalence class partitioning
  b) Boundary value testing

Signature

```
/* recursive tree structure                    */
typedef struct _tree
{
  struct _tree * left;
  struct _tree * right;
  int element;
  int height;
} Tree;
```

`insert(int e, Tree *t)`: Insert element `e` into the tree `t`

Note:

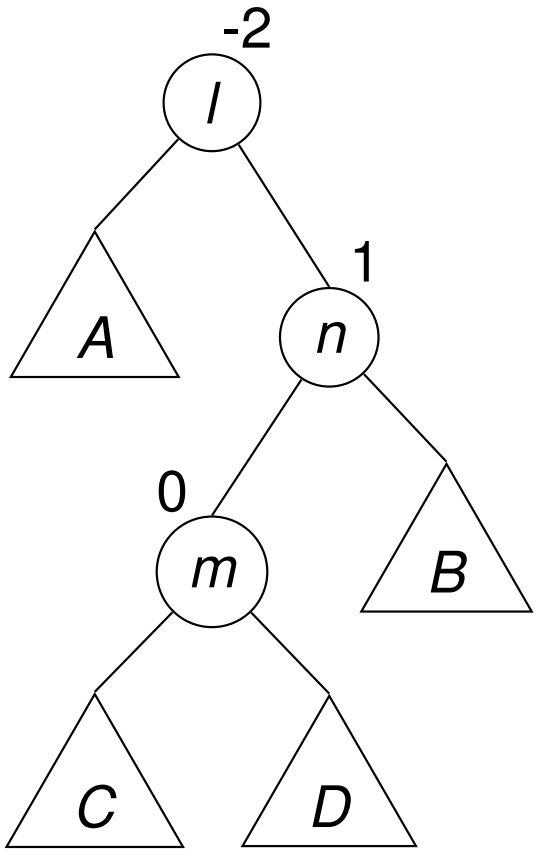
- You don't know the concrete implementation
- But you know how an AVL is supposed to work

Equivalence Classes for Inputs

Remember: Tree `t` is an input, too!

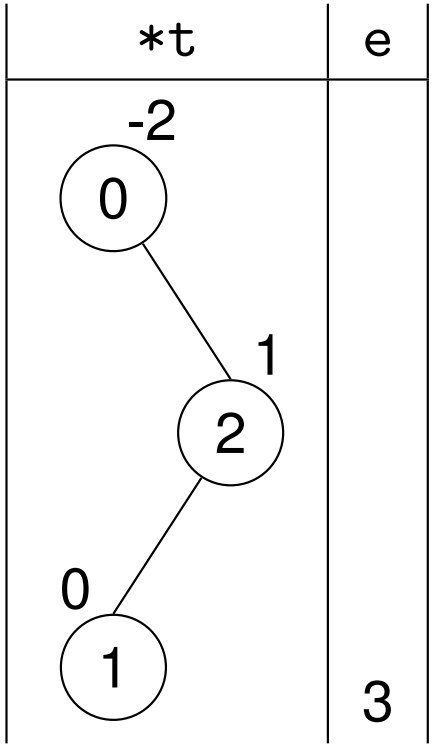
- Balanced: $|\text{left height} - \text{right height}| \leq 1$
- Elements in left sub-tree are smaller than elements in right sub-tree

1. Derive equivalence classes:
   - based on balance
   - number of elements
   - content
   - ...
2. Justify *and* illustrate your equivalence classes (see right).
3. Use table (as in lecture) to list your equivalence classes



Boundary Value Testing

1. Derive test cases using boundary value testing:
   - cover all equivalence classes (valid, invalid)
   - take outputs into account
2. Illustrate your test cases (see right)
3. Use table (as in lecture) to list your test cases



Submitting your solution

- Part 2 is a “pencil and paper” assignment, available separately
- Your solution must be handed in via TUWEL on April 30, 4pm
  - Use the file names `part1.pdf` and `part2.pdf`
    - There’s a scanner in the main library
  - Late submissions will not be accepted
- Make sure the solution shows your student ID and your name.

Name	Student ID

Coverage. Consider the following program fragment (a simple version of Kadane's algorithm for the maximums sub-array problem) and test suite:

```
bool subarr(int i, int j, int k)
  int maxsum = i;
  int lastsum = i;
  if (lastsum < 0)
    lastsum = j;
  else
    lastsum += j;
  if (lastsum > maxsum)
    maxsum = lastsum;
  if (lastsum < 0)
    lastsum = k;
  else
    lastsum += k;
  if (lastsum > maxsum)
    maxsum = lastsum;
  return maxsum;
}
```

Inputs			Output
i	j	k	result
-3	-1	2	2
3	-1	2	4

Control-Flow-Based Coverage Criteria. Indicate (✓) which of the following coverage criteria are satisfied by the test-suite above (assume that the term "decision" refers to all Boolean expressions in the program).

	satisfied	
Criterion	yes	no
path coverage		
statement coverage		
branch coverage		
decision coverage		
condition/decision coverage		

Data-Flow-Based Coverage Criteria. Indicate (✓) which of the following coverage criteria are satisfied by the test-suite above (here, the parameters of the function do not constitute definitions):

	satisfied	
Criterion	yes	no
all-defs		
all-c-uses		
all-p-uses		
all-c-uses/some-p-uses		
all-p-uses/some-c-uses		
all-uses		
all-du-paths		

If the test-suite from above does not satisfy the coverage criteria listed below, augment it with test-cases such that these criteria are satisfied. If full coverage cannot be achieved for one or more of these criteria, explain why.

decision coverage

Inputs			Output
`i`	`j`	`k`	result

all-p-uses/some-c-uses

Inputs			Output
`i`	`j`	`k`	result

MC/DC

Inputs			Output
`i`	`j`	`k`	result

Provide sufficiently many test-cases to guarantee modified condition/decision coverage for the following program fragment:

```
bool foo(int x, int y) {
  return ((x < y) || (y >> 2 == 0 ));
}
```

Input		Output
`x`	`y`	result