Software Analysis and Verification

Lecture 1.

A: Introduction to the subject

B: Format of the class

Instructor: Viktor Kuncak, INR 318

viktor.kuncak@epfl.ch

Part A:
Introduction to the subject

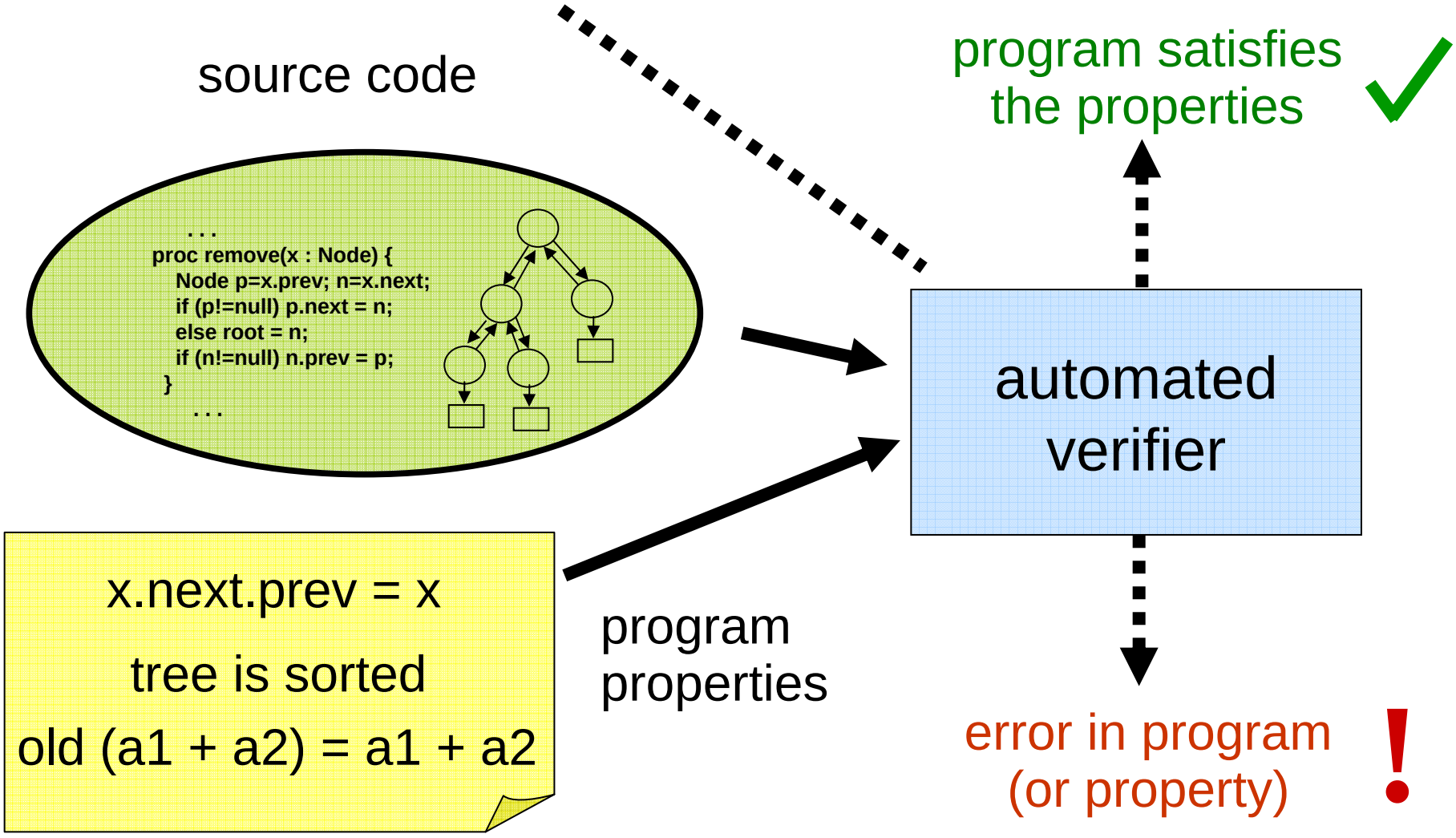
Big picture

Example: property, loop invariant, demo

Uses of software analysis and verification

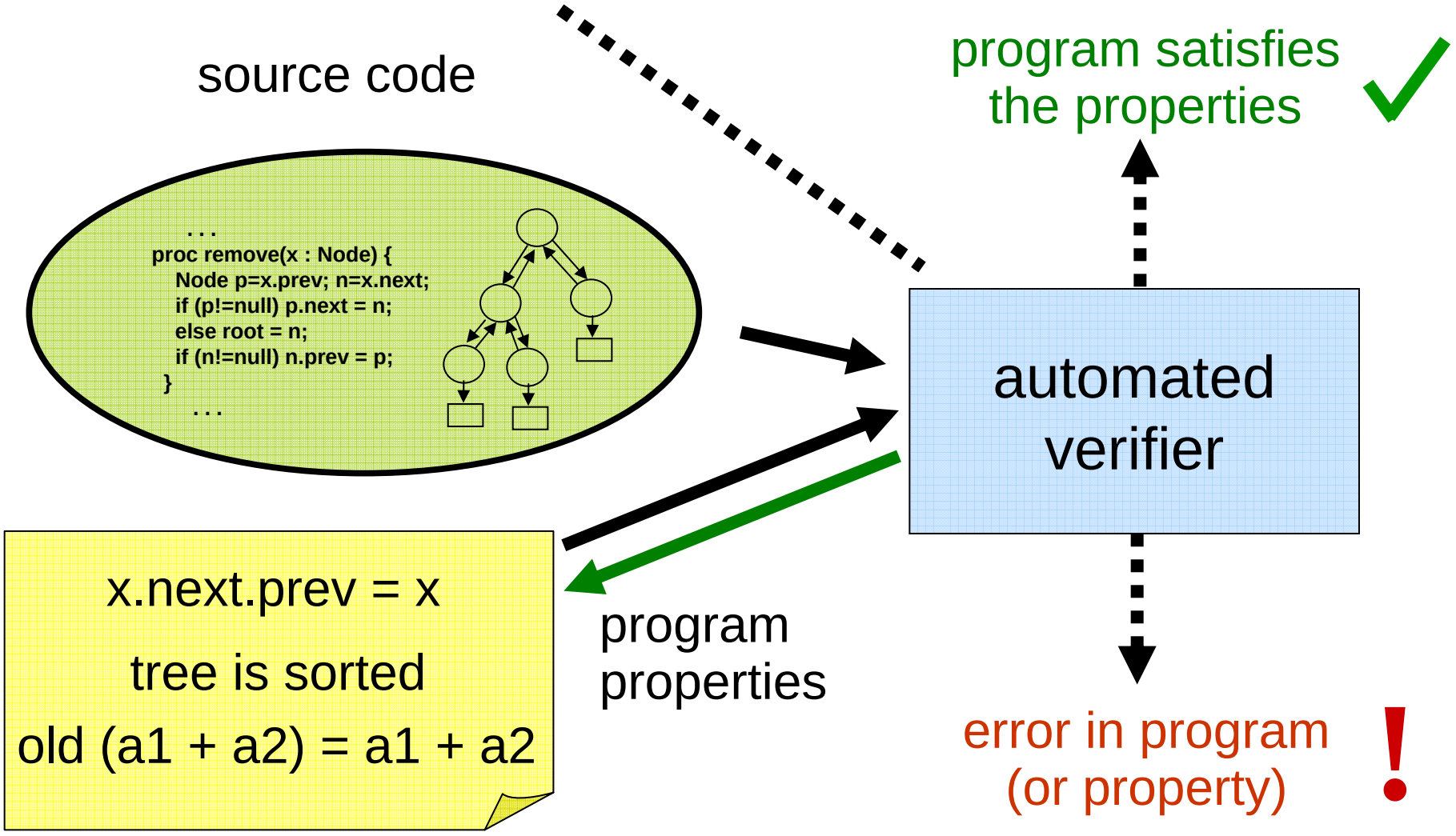
Software analysis and verification

how can we build it: theory and practice



Discovering properties

how can we build it: theory and practice



A simple Java method

```
public static int sum(int a0, int n0)
{
  int res = 0, a = a0, n = n0;
  while (n > 0) {
    a = 2*a;
    res = res + a;
    n = n – 1;
  }
  return res;
}
```

An example numerical property

```
public static int sum(int a0, int n0)
/*:
  requires “a0 ≥ 0 & n0 ≥ 0”
  ensures “result ≥ 0”
*/
{
  int res = 0, a = a0, n = n0;
  while (n > 0) {
    a = 2*a;
    res = res + a;
    n = n – 1;
  }
  return res;
}
```

method contract:
“if parameters are non-negative, then so is the returned value”

Does the property hold?

Why?

An example numerical property

```
public static int sum(int a0, int n0)
/*:
  requires "a0 ≥ 0 & n0 ≥ 0"
  ensures "result ≥ 0"
*/
{
  int res = 0, a = a0, n = n0;
  while /*: invariant "a ≥ 0 & res ≥ 0" */ (n > 0) {
    a = 2*a;
    res = res + a;
    n = n – 1;
  }
  return res;
}
```

method contract:
"if parameters are non-negative, then so is the returned value"

inductive loop invariant:

1. true at loop entry
2. if true at iteration k, true at k+1
3. implies the desired property.

Demo
using Jahob

Replace ≥ with **>** . Loop invariant?

```
public static int sum(int a0, int n0)
/*:
  requires “a0 > 0 & n0 > 0”
  ensures “result > 0”
*/
{
  int res = 0, a = a0, n = n0;
  while /*: invariant “                    ” */ (n > 0) {
    a = 2*a;
    res = res + a;
    n = n – 1;
  }
  return res;
}
```

method contract:
“if parameters are non-negative, then so is the returned value”

inductive loop invariant:

1. true at loop entry
2. if true at iteration k, true at k+1
3. implies the desired property.

What does a verification system do?

```
public static int sum(int a0, int n0)
/*:
  requires “a0 > 0 & n0 > 0”
  ensures “result > 0”
*/
{
  int res = 0, a = a0, n = n0;
  while /*: invariant “a>0 & res≥0 & (res>0 | n>0)”*/
  (n > 0) {
    a = 2*a;
    res = res + a;
    n = n – 1;
  }
  return res;
}
```

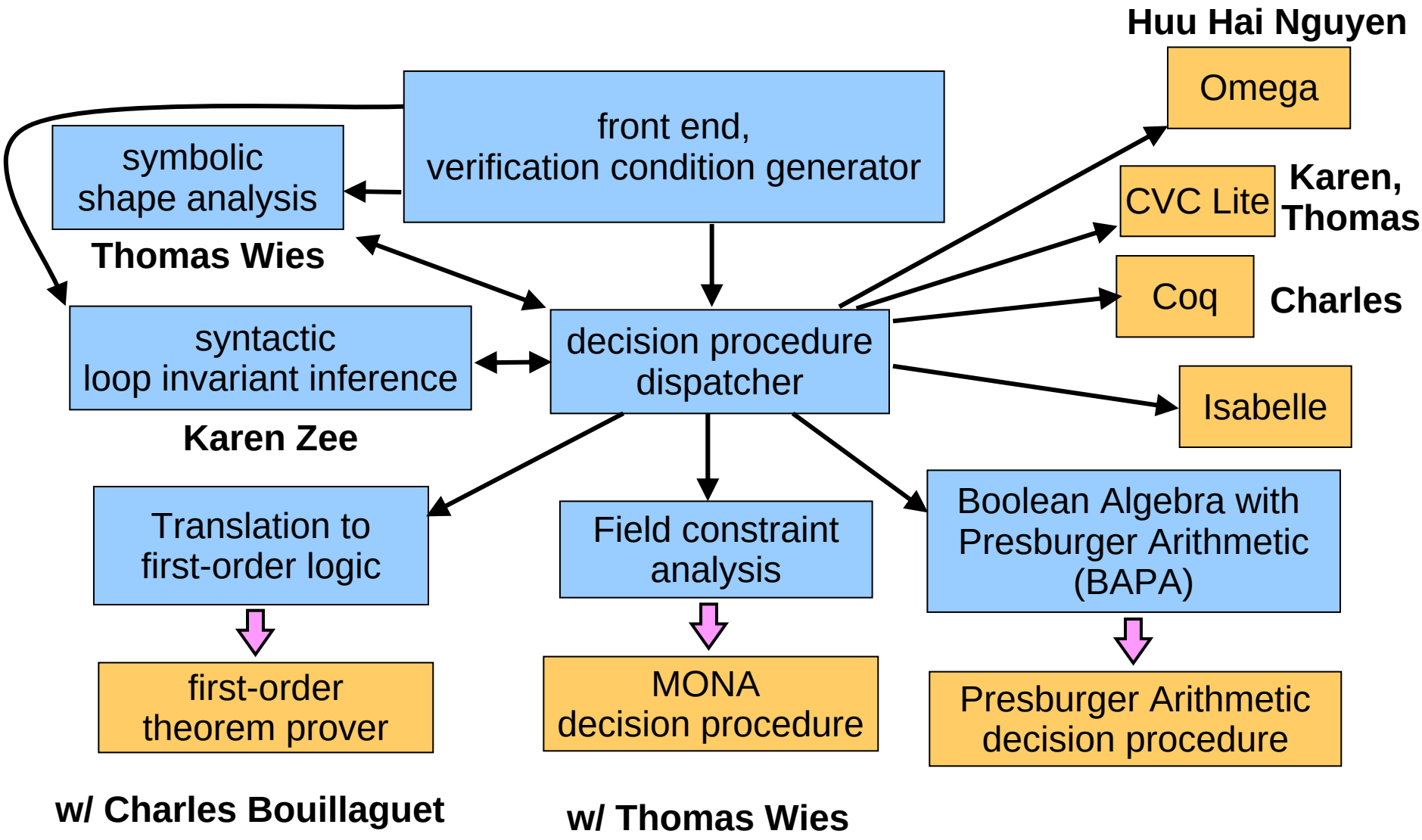
accepts program and desired properties

infers intermediate properties along the way

checks whether desired properties hold

Our automated verifier: Jahob

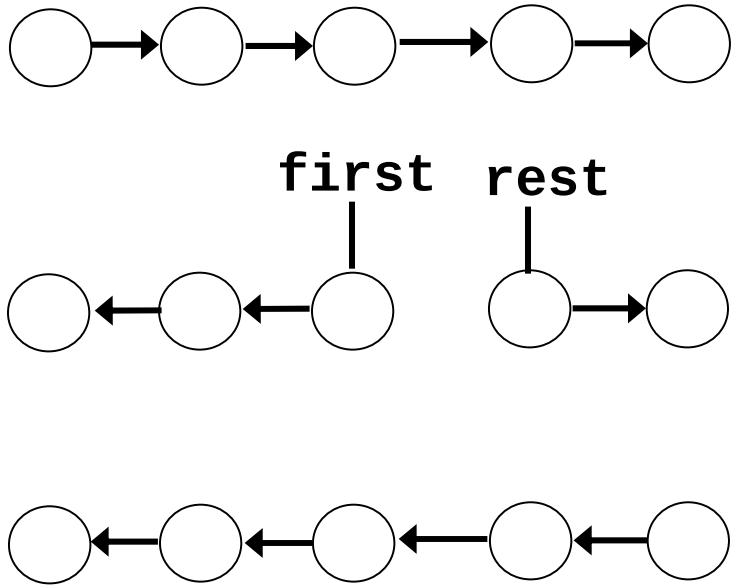
(you can use it in your final project if you wish)



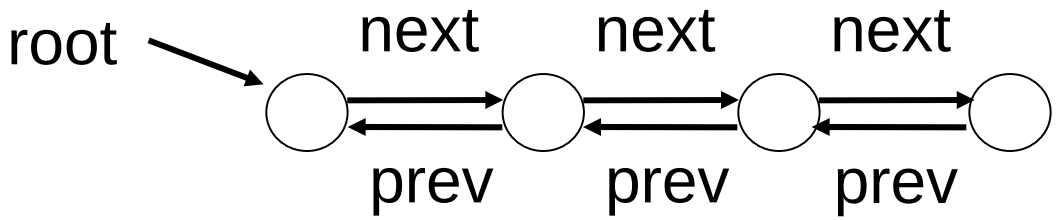
A loop invariant inferred by Jahob

```
public static void reverse(int a0, int n0)
/*: modifies content
    ensures "content = old content"
*/
{
   Node tmp, rest;
   rest = first; first = null;
   while (rest != null) {
     tmp = first; first = rest;
     rest = rest.next;
     first.next = tmp;
   }
}
```

"no new nodes introduced or existing nodes lost"



Shape invariants



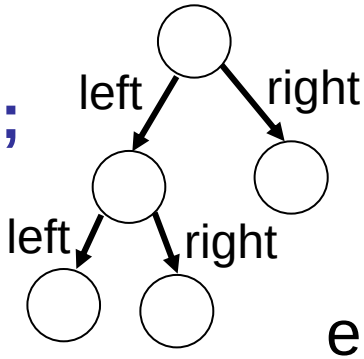
x.next.prev == x

acyclicity of next

shape not given by types,

but by structural properties;

may change over time



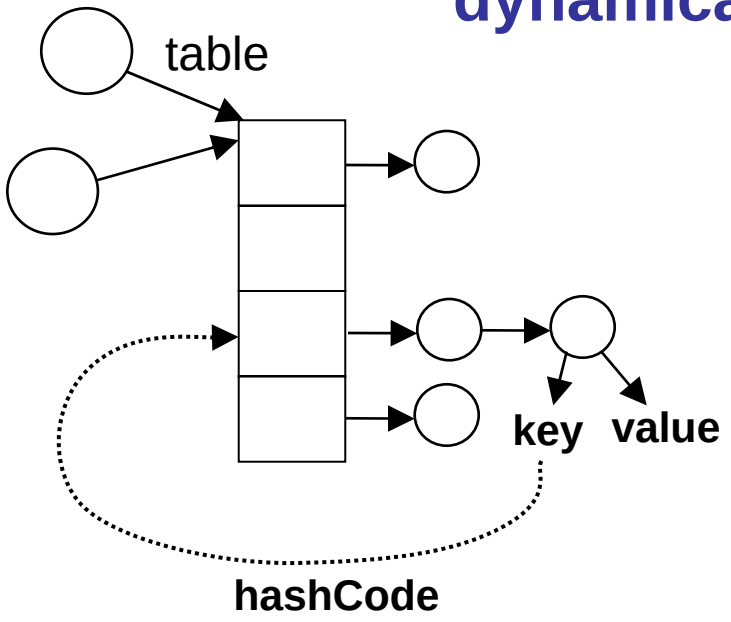
graph is a tree

elements are sorted

```
class Node {
    Node f1, f2;
}
```

Shape and numerical quanties

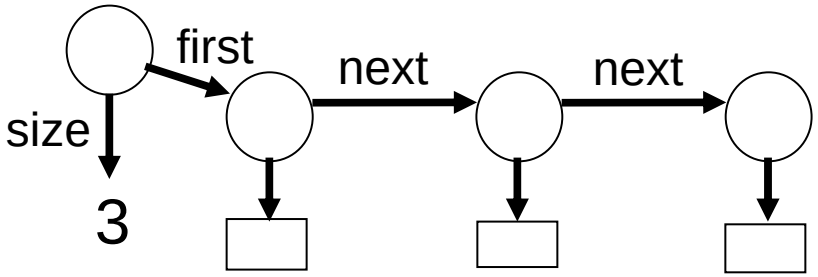
dynamically allocated arrays



node is stored in the bucket
given by the hash of node's key

instances do not share array

numerical quantities



value of size field is
the number of stored objects

Jahob summary

Verifies programs in Java subset

Specifications written in subset of Isabelle/HOL

Jahob proves

- data structure preconditions, postconditions
- data structure invariants
- absence of run-time errors

Current strength: data structures

Verified data structures

- lists, trees, priority queues, hash tables

Why are such tools interesting?

Reliability

- discovering and preventing existing errors
- making it easier to write correct programs

Performance

- prove properties that enable optimizations

Code maintenance and understanding

- reverse engineering, visualization, refactoring

Intellectual challenge

Application for reliability

Everyday software full of bugs – cost up to 60 billion/year

Critical software (and hardware) - not very different

- air-traffic control: are 2 planes going to collide
- Ariane 5
- Mars Rover
- Northeast black out in US
- software in your car: 10^5 lines of code (recalls)
- nuclear submarine
- heart pace maker

Reports of serious bugs in all of these

Developing a good tool is difficult, but can pay off!

Verification and analysis systems

Jahob,

Blast, HyTech

SLAM

Spec#

ESC/Java2

TVLA

FindBugs

Saturn

...

Full or partial correctness

Full correctness, full specifications:

- all we would like to be true about the program
- difficulty: specification as hard as program

Partial correctness, partial specifications:

- select most important properties
- specifications smaller
- more cost-effective

Ideally: tool can check everything

- programmer selects how much to verify

Full specification

```
public static int sum(int a0, int n0)
/*:
  requires “a0 > 0 & n0 > 0”
  ensures “result = 2*a0*(2^n0 - 1)”
*/
{
  int res = 0, a = a0, n = n0;
  while /*: invariant “                                    ”*/
  (n > 0) {
    a = 2*a;
    res = res + a;
    n = n – 1;
  }
  return res;
}
```

Tools applied in industry

ASTREE (ENS, Paris)

- In Nov. 2003, ASTRÉE was able to prove completely automatically the absence of any run-time error in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in 1^h20

Coverity

GrammaTech

AbsInt

SparkAda

Application for performance

Common sub-expression elimination

Moving code out of the loop: interference

Induction variable elimination (value change)

Parallelization: also interference

Static memory allocation

- escape analysis: do allocated objects live only within the procedure?
- static preallocation, garbage collection
- write barrier removal

Applications for maintenance

Reverse engineering: recover specifications

Refactoring: is a given transformation valid

Slicing, dependency analysis:
what parts of program should we look at?

Which components interact?

Call graph analysis

- What method could a given call site invoke?

Intellectual challenge

Verifying compiler grand challenge

- compared to human genome project
- compiler that says “this procedure is fishy”

Undecidability results: Rice’s theorem

Computational complexity

Key challenge: what is the class of programs

- programming discipline

Verifying “understandable programs”

- AI completeness

Part B:
Format of the class

Suggested prerequisites

Talk to me if you did not take them

- Theory of Computation
- Compilers

What will you learn?

What analysis and verification is good for

What we can do today

What we may be able to do tomorrow

Good if you are interested in

- pursuing research in analysis and verification
- using verification techniques in other areas
- using existing tools to find and prevent errors in your programs, and improve these tools
  - important trend: customizable analysis tools

Who is teaching

Instructor: myself

You: active participation (more on this later)

Some invited lectures

Attend selected talks in verification at EPFL

- I will point them out to the class when relevant

Perhaps a teaching assistant towards the end

Grading

Assumption: you are interested in material

- just do your best and do not worry
- if material unclear, ask immediately

Elements of your grade – do them all

- mini project: describe, implement, present
- homeworks: write and grade
- write lecture notes and put them online (wiki)
  - in pairs, also ask others and me if anything unclear
- paper summaries: write, lead discussions
- lecture participation: ask and answer questions

Writing lecture notes (scribing)

Everyone must do it

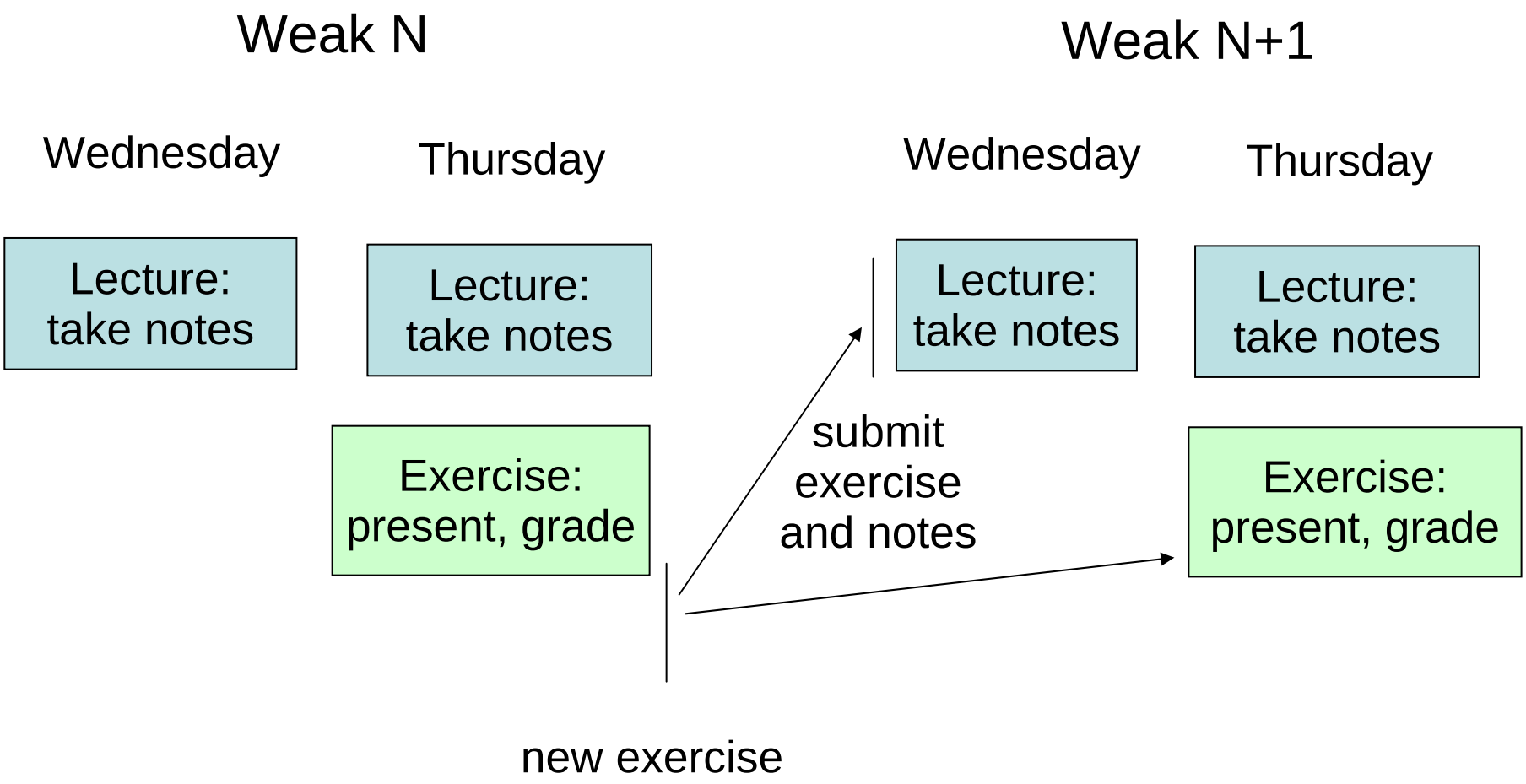
- after we go through everyone, start again
- put initial draft in wiki format (like Wikipedia)
- your class mates and myself can improve it
- no strict grading, but need to do it seriously

Volunteer before the class

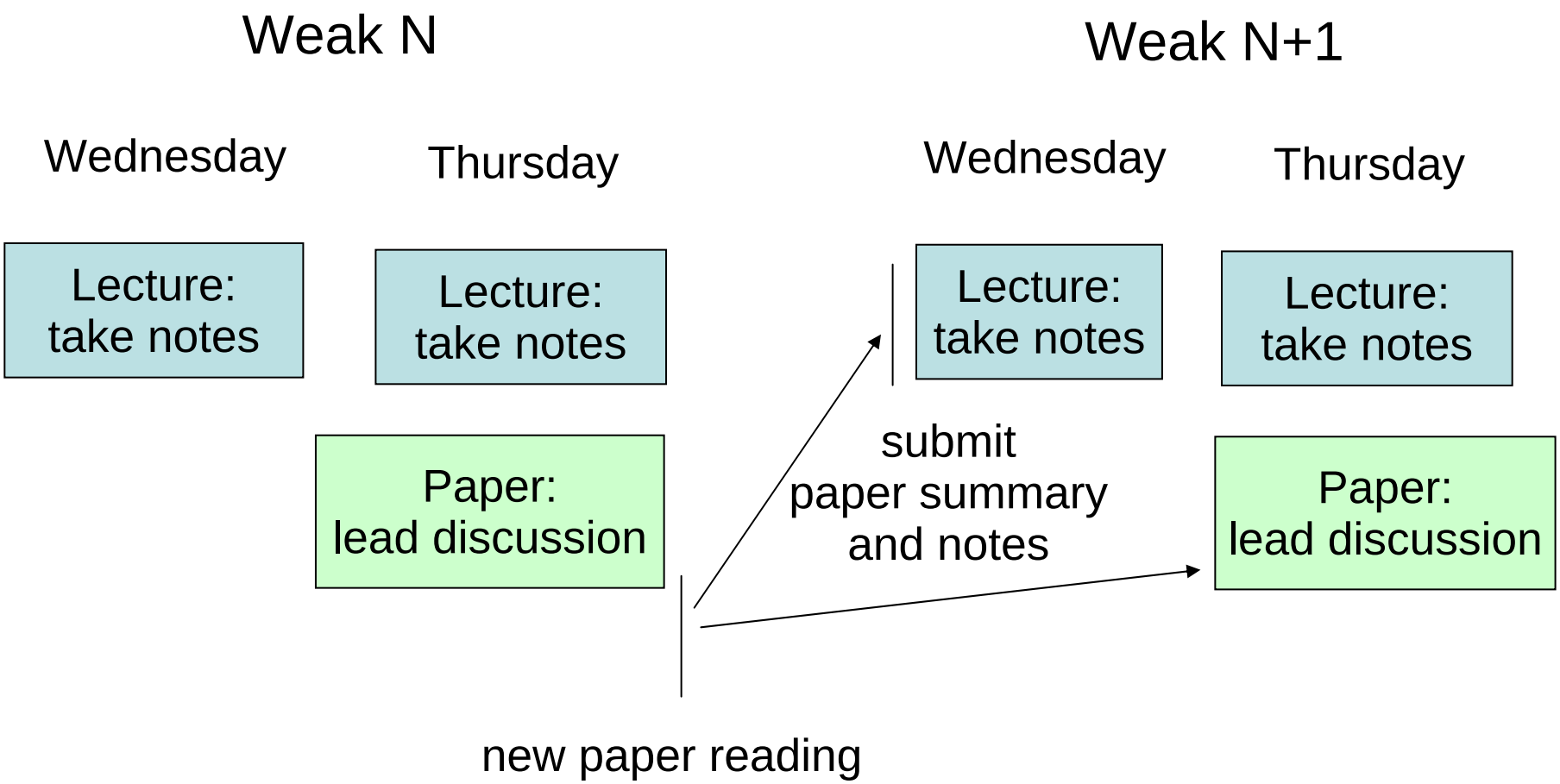
- anyone for tomorrow's class?

  programs → formulas
  proving certain formulas

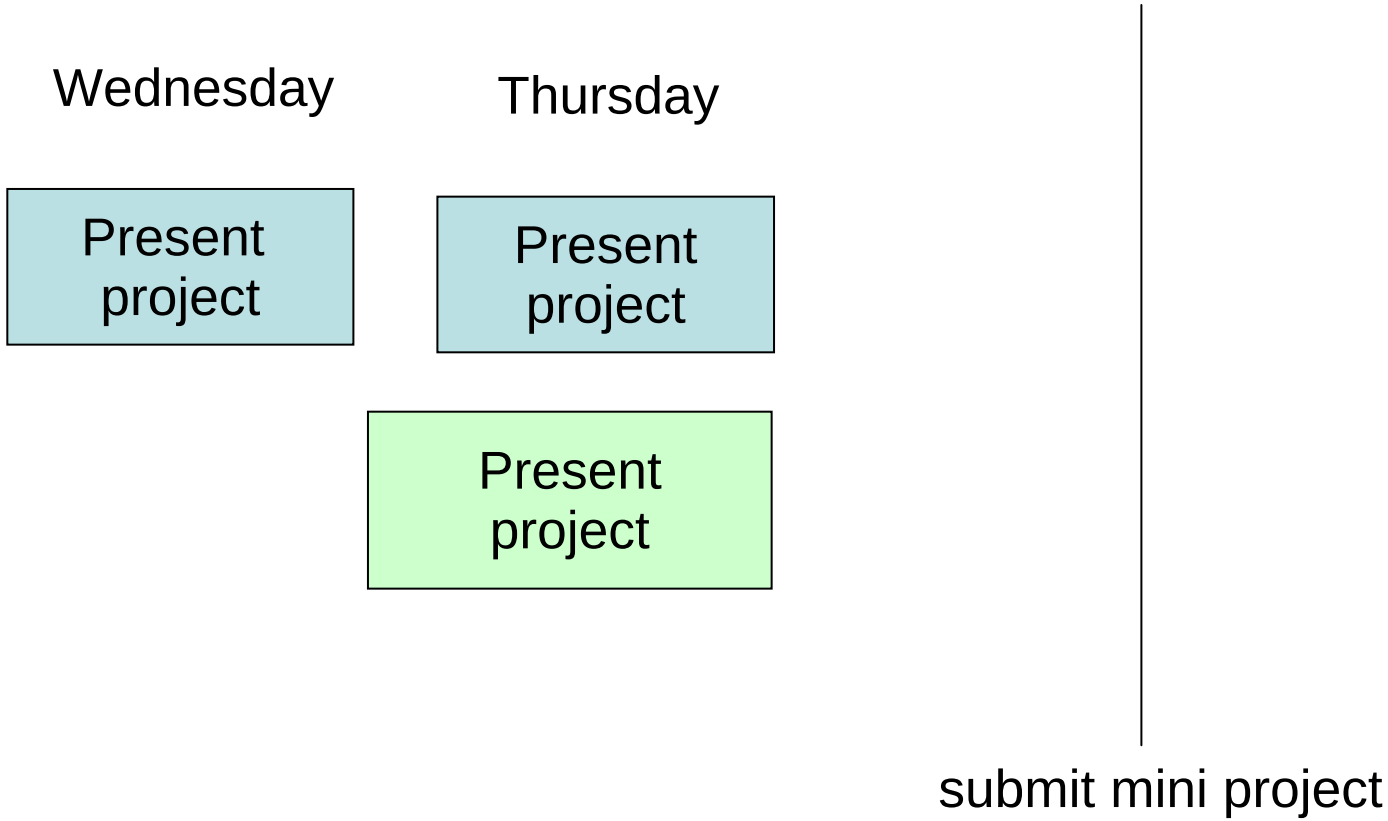
What happens when – phase 1



What happens when – phase 2



What happens when – phase 3



Final mini project

You can use Jahob if you wish

- take advantage of existing system
- may take some time to understand

You can start from scratch

- build a small language

Recommended impl. language: O'Caml

- can use ML, Haskell, Scala, Java, ...

Can have a project without implementation

- prove somewhat new theorems in it

Some topics we expect to cover

- Verification condition generation
- Theorem proving and decision procedures
- Abstract interpretation (dataflow analysis)
- Predicate abstraction and shape analysis
- Interprocedural analysis
- Concurrent, higher-order, object-oriented features
- Run-time checking, bug finding

Preliminary Quiz

Does NOT affect the grade

Helps me make lectures more interesting

Answer in electronic form and email back