Specification and Verification in Introductory Computer Science

Frank Pfenning
Carnegie Mellon University
MSR-CMU Center for Computational Thinking
May 14, 2012

www.cs.cmu.edu/~fp/courses/15122
c0.typesafety.net

Principal Contributors: Rob Arnold, Tom Cortina, Ian Gillis, Jason Koenig, William Lovas, Karl Naden, Rob Simmons, Jakob Uecker

MSR Collaborators: Rustan Leino, Nikolaj Bjørner
MS Guest Speaker: Jason Yang (Windows team)
MSR Inspiration: Manuvir Das, Peter Lee

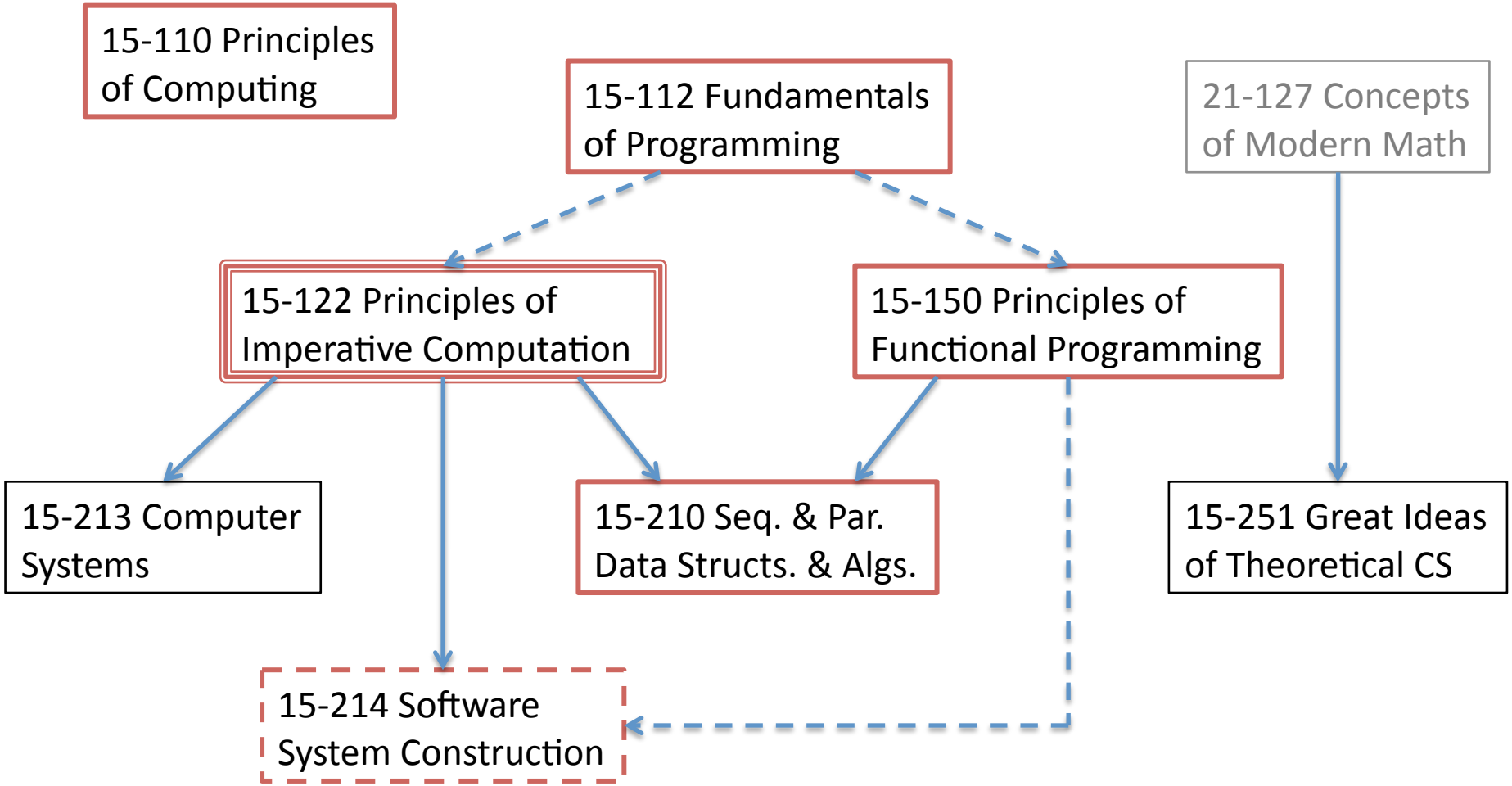
Outline

- Background and guiding principles
- Role in the curriculum
- Learning goals
- C0 Language
- (Lecture sample)
- Research plans

Background

- 15-122 Principles of Imperative Computation
- Part of a major core CS curriculum revision
  - 15-110/112 Principles of Computing/Programming
  - 15-150 Principles of Functional Programming
  - 15-210 Parallel & Sequential Data Structs. & Algs.
  - 15-214 Software System Construction
- Still under development
  - Pilot in Fall 2010, every semester since then
  - Now taught ~630 students, majors & non-majors
  - Adoption at Tübingen, Germany, Spring 2012

Core Curriculum Chart



Guiding Principles

- Computational thinking and programming must go hand-in-hand
- Algorithm and data structure design, analysis, and implementation is an intellectual activity
- We build abstractions from an understanding of the concrete
- Rigorous types, invariants, specifications, interfaces are crucial

Role in the New Curriculum

- Precondition: some programming experience
  - Self-taught or high-school programming or 15-112
  - Python and Java most common
  - Broad rather than deep; diverse
- Postcondition: preparation for 15-2xx courses
  - 15-210: Par. & Seq. Data Structs. & Algs.
  - 15-213: Computer Systems
  - 15-214: Software System Construction

Learning Goals

- Computational thinking
- Programming skills
- Specific data structures and algorithms
- Application contexts

Computational Thinking

- Algorithmic concepts vs. code
- Abstraction and interfaces
- Specification vs. implementation
- Pre- and post-conditions, loop invariants
- Data structure invariants
- Logical and operational reasoning
- Asymptotic complexity and practical efficiency
- Programs as data
- Exploiting randomness

Programming Skills

- Deliberate programming
- Understand static and dynamic semantics
- Develop, test, debug, rewrite, refine
- Invariants, specifications
- Using and designing APIs
- Use and implement data structures
  - Emphasis on mutable state ("RAM model")
- Render algorithms into correct code
  - Emphasis on imperative programming

Some Algorithmic Concepts

- Asymptotic analysis
  - Sequential computation
  - Time and space
  - Worst-case vs. average-case
  - Amortized analysis
  - Common classes: O(n), O(n*log(n)), $O(n^2)$, $O(2^n)$
- Divide and conquer
- Self-adjusting data structures
- Randomness
- Sharing

Specific Alg's and Data Struct's

- Binary search
- Sorting (selection sort, mergesort)
- Stacks and queues
- Hash tables
- Priority queues (heaps)
- Binary search trees (red/black, randomized)
- Tries
- Binary decision diagrams (SAT, validity)
- Graph traversal (depth-first, breadth-first)
- Minimum spanning trees (Prim's alg., Kruskal's alg.)
- Union-find

Application Contexts

- See algorithms in context of use
- Engage students' interest
- Assignments (all written+programming)
  - Image manipulation
  - Text processing (Shakespeare word freq's)
  - Grammars and parsing
  - Maze search
  - Huffman codes
  - Puzzle solving (Lights Out)
  - Implementing a virtual machine (C0VM)

Language

- Weeks 1-10: Use C0, a **small safe subset** of C, with a layer to express contracts
  - Garbage collection (malloc/free)
  - Fixed range modular integer arithmetic
  - Language unambiguously defined
  - Contracts as boolean expressions
- Weeks 11-14: Transition to C
  - Exploit positive habits, assertion macros
  - Pitfalls and idiosyncrasies of C

Type Structure

- t = int | bool | struct s | t* | t[]
- Distinguish pointers and arrays
- Distinguish ints and booleans
- Structs and arrays live in memory; ints, bools, and pointers in variables
- Strings and chars as abstract types

Control Structure

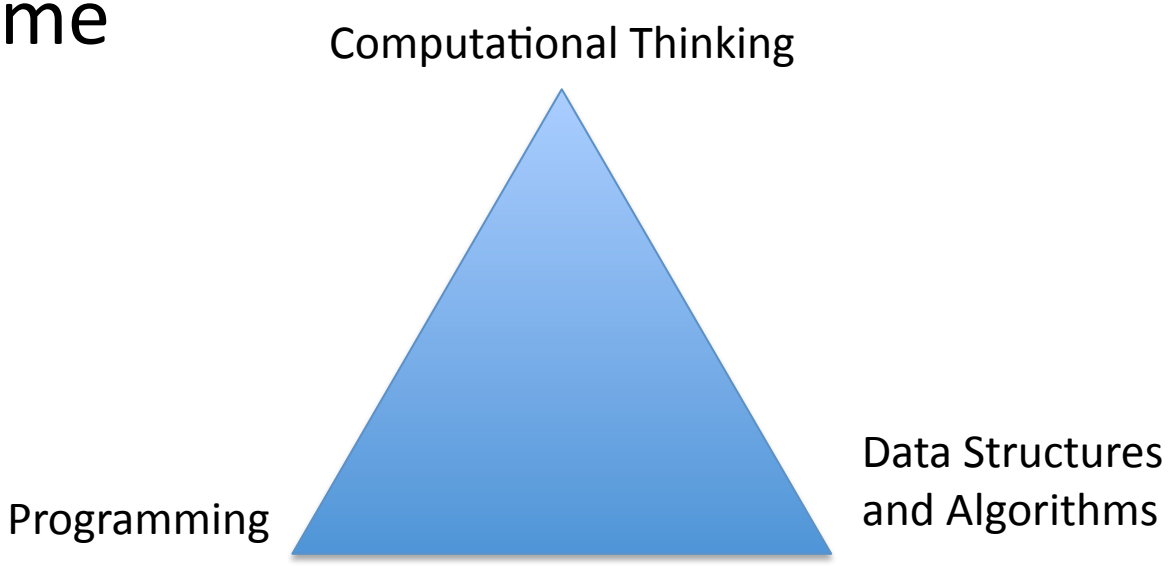
- Variables and assignment
- Separation of expressions and statements
- Conditionals, while, and for loops
- Functions
- Other considerations
  - Minimal operator overloading
  - No implicit type conversions
  - Initialization

Rationale for C0

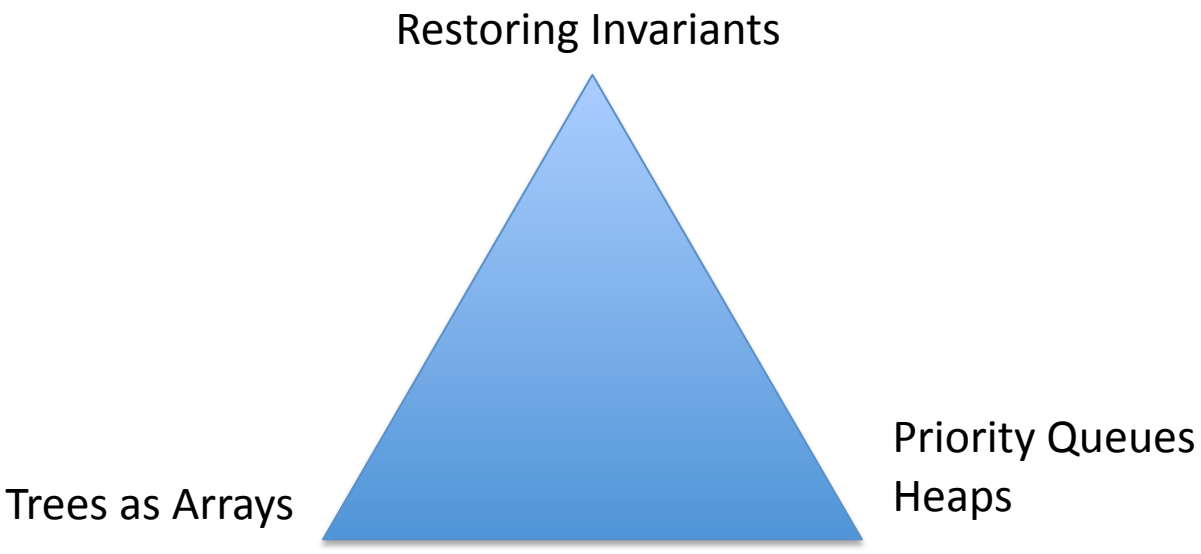
- Imperative implementations of simple algorithms are natural in this fragment
- Simplicity permits effective analysis
  - Proving invariants, sound reasoning
- Concentrate on principles first, C later
- Industrial use of assertions (SAL, Spec#)
  - Guest lecture by J. Yang from MS Windows team

Lecture Example

- In actual lecture use blackboard, plus laptop for writing code
- Recurring theme



Lecture 13: Priority Queues
Lecture 14: Restoring Invariants



Heap Interface

```
typedef struct heap* heap;
bool heap_empty(heap H);          /* is H empty? */
heap heap_new(int limit)          /* create new heap */
//@requires limit > 0;
//@ensures heap_empty(\result);
  ;
void heap_insert(heap H, int x); /* insert x into H */
int heap_min(heap H)              /* find minimum */
//@requires !heap_empty(H);
  ;
int heap_delmin(heap H)           /* delete minimum */
//@requires !heap_empty(H);
  ;
```

Checking Heap Invariants

```
struct heap {
  int limit;
  int next;
  int[] heap;
};

bool is_heap(heap H)
//@requires H != NULL && \length(H->heap) == H->limit;
{
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  for (int i = 2; i < H->next; i++)
    if (!(H->heap[i/2] <= H->heap[i])) return false;
  return true;
}
```

Heap Insertion

```
void heap_insert(heap H, int x)
//@requires is_heap(H);
//@requires !heap_full(H);
//@ensures is_heap(H);
{
  H->heap[H->next] = x;
  H->next++;
  sift_up(H, H->next-1);
}
```

Preliminary Course Assessment

- First four course instances successful
  - Covered envisioned material and more
  - Excellent exam and assignment performance
  - Lecture notes
  - Positive student feedback
- Interesting programming assignments
  - Using our C0 compiler weeks 1-10, gcc thereafter
  - Linked with C/C++ libraries

Some Course Tools

- C0 to C compiler (cc0), front end v3
- C0 interpreter and debugger (new Su 2012)
- C0 tutorial
- C0 language reference
- Binaries for Windows, Linux, Mac OS X
- C0VM for last assignment

Contracts

- Currently, contracts are checked dynamically if compiled with cc0 -d
- Contracts are enormously useful
  - Bridge the gap from algorithm to implementation
  - Express programmer intent precisely
  - Catch bugs during dynamic checking
  - Debugging aid (early failure, localization)
- Not easy to learn effective use of contracts

Exploiting Contracts Further

- Ongoing research
- Contracts could be even more useful
  - Static verification (MSR: Daphne, Boogie, Z3)
  - Counterexample, test generation (MSR: Pex)
  - Autograding of code and contracts
- The educational setting creates unique challenges and opportunities
  - Explainable static analysis (null ptrs, array bounds)
  - Pedagogically sound design
  - Diversity of student programmers

Summary

- 15-122 Principles of Imperative Computation
- Freshmen-level course emphasizing the interplay between computational thinking, algorithms, and programming in simple setting
- C0, a small safe subset of C, with contracts
- Contracts pervasive, checked only dynamically
- Research question: can we use static analysis and theorem proving to aid in achieving student learning goals?
- MSR collaborators: Rustan Leino, Nikolaj Bjørner
- Visits: F. Pfenning, J. Yang, R. Leino, K. Naden, J. Koenig,

Priority Queues

- Generalizes stacks and queues
- Abstract interface
  - Create a new priority queue
  - Insert an element
  - Remove a minimal element

Heaps

- Alternative 1: unsorted array
  - Insert O(1), delete min O(n)
- Alternative 2: sorted array
  - Insert O(n), delete min O(1)
- Alternative 3: heap
  - Partially sorted!

Heaps

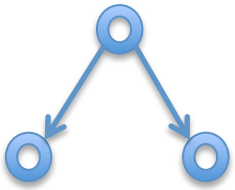
- A heap represents a priority queue as a binary tree with two invariants
- Shape:
  - Tree is complete (missing nodes bottom-right)
- Order:
  - Each interior node is greater-or-equal to its parent
  - OR: each node is less-or-equal to all its children
- Guarantee a minimal element is at root!

Shape Invariant

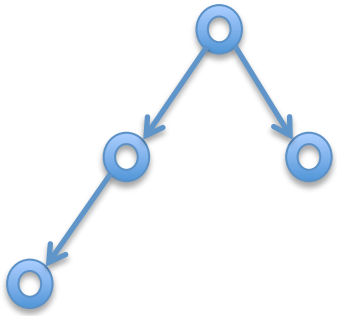
1 node

2 nodes

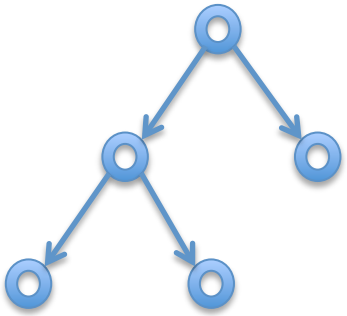
3 nodes



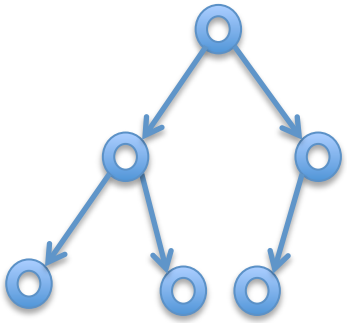
4 nodes



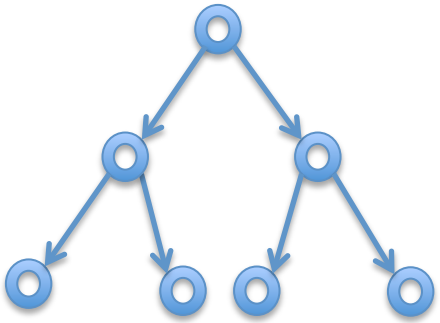
5 nodes



6 nodes

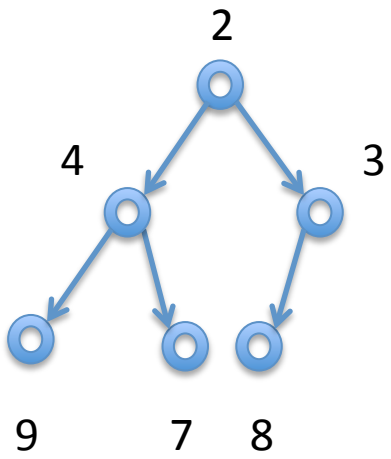


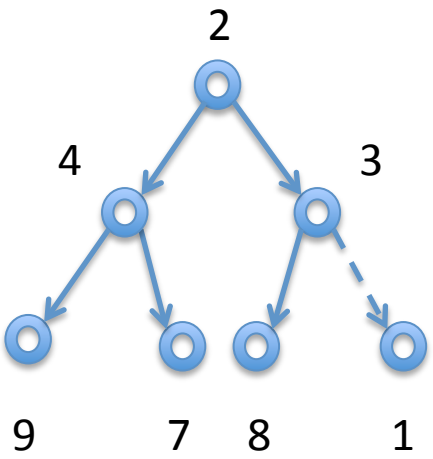
7 nodes



Inserting into Heap

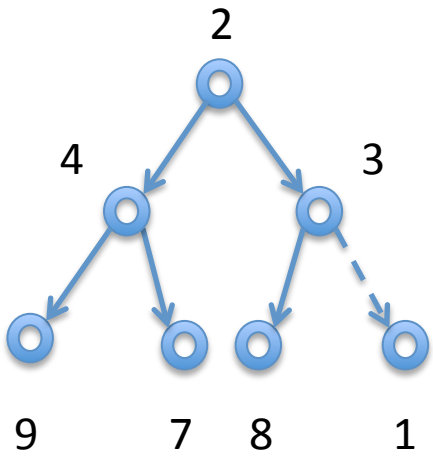
- By shape invariant, know where new element should go
- Now have to restore ordering invariant

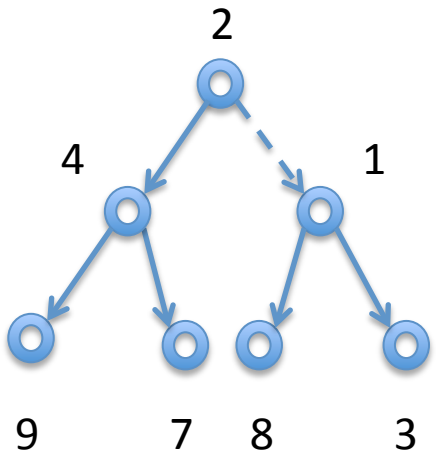




Sifting Up

- Order invariant satisfied, except between new node and its parent ("looking up")
- Swapping with parent will restore locally
- Parent may now violate invariant





Invariant Restored

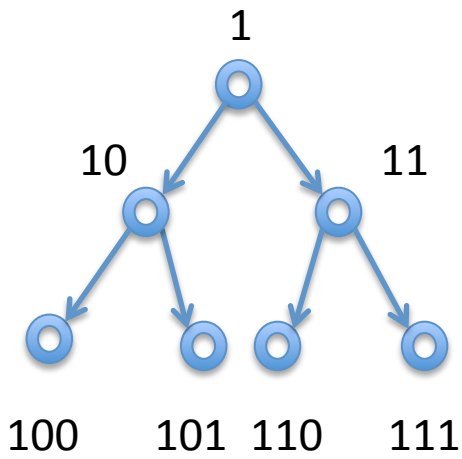
- When reaching root (no parent!), ordering invariant restored everywhere
- We have a valid heap!

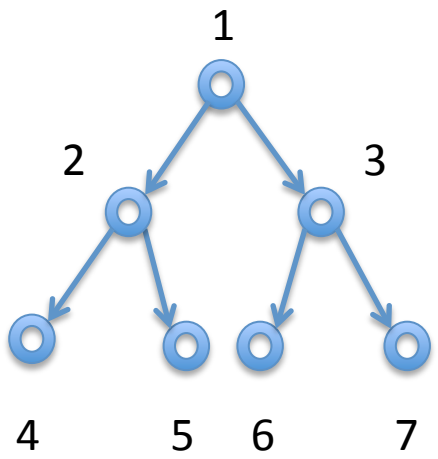
Analysis

- Insert requires O(log(n)) swaps in worst case
- Logarithmic thinking: complete binary tree with n elements has 1+log(n) levels
- Delete min also O(log(n)), omitted here

Heaps as Arrays

- Exploit shape invariant
- Binary numbering system for nodes, root is 1
- Left child is n0 = 2*n; right child is n1 = 2*n+1
- Parent is at n/2; 0 is unused





Creating a Heap

```
heap heap_new(int limit)
//@requires 1 <= limit;
//@ensures is_heap(\result) && heap_empty(\result);
{
  heap H = alloc(struct heap);
  H->limit = limit;
  H->next = 1;
  H->heap = alloc_array(int, limit);
  return H;
}
```

- Pre/postconditions are easy to specify and reason about
- How to automate?

Almost a Heap

```
bool is_heap_except_up(heap H, int n)
//@requires H != NULL && \length(H->heap) == H->limit;
{ int i;
  if (!(1 <= H->next && H->next <= H->limit)) return false;
  for (i = 2; i < H->next; i++)
    if (!(i == n || H->heap[i/2] <= H->heap[i]))
      return false;
  return true;
}
// is_heap_except_up(H, 1) == is_heap(H);
```

- Captures permitted violation precisely
- Observation at end for postcondition of `sift_up`

Sifting Up / Restoring Invariant

```
void sift_up(heap H, int n)
//@requires 1 <= n && n < H->limit;
//@requires is_heap_except_up(H, n);
//@ensures is_heap(H);
{ int i = n;
  while (i > 1)
    //@loop_invariant is_heap_except_up(H, i);
    {
      if (H->heap[i/2] <= H->heap[i]) return;
      swap(H->heap, i/2, i);
      i = i/2;
    }
  //@assert i == 1;
  //@assert is_heap_except_up(H, 1);
  return;
}
```