atomic
compound
lists
trees
graphs

search
trees
graphs

Core Recipes	Template Origin		Abstraction
	Data Driven	Control Driven	
How to Design Functions (HtDF) Design any function.	Data Driven Templates Produce template for a data definition based on the form of the type comment.	Function Composition	From Examples Produce an abstract function given two similar functions.
How to Design Data (HtDD) Produce data definitions based on structure of the information to be represented.	2 One-of Data Functions where 2 arguments have a one-of in their type comments.	Failure Handling	From Type Comments Produce a fold function given type comments.
How to Design Worlds (HtDW) Produce interactive programs that use `big-bang`.		Backtracking Search	
		Generative Recursion	Using Abstract Functions
		Accumulators	
	Template Blending		

L20 L21 L22

L18 L19 L20

Accumulators

- accumulate information from prior recursive calls
- three categories
  - preserve context from prior recursive calls
  - result so far

these overlap some

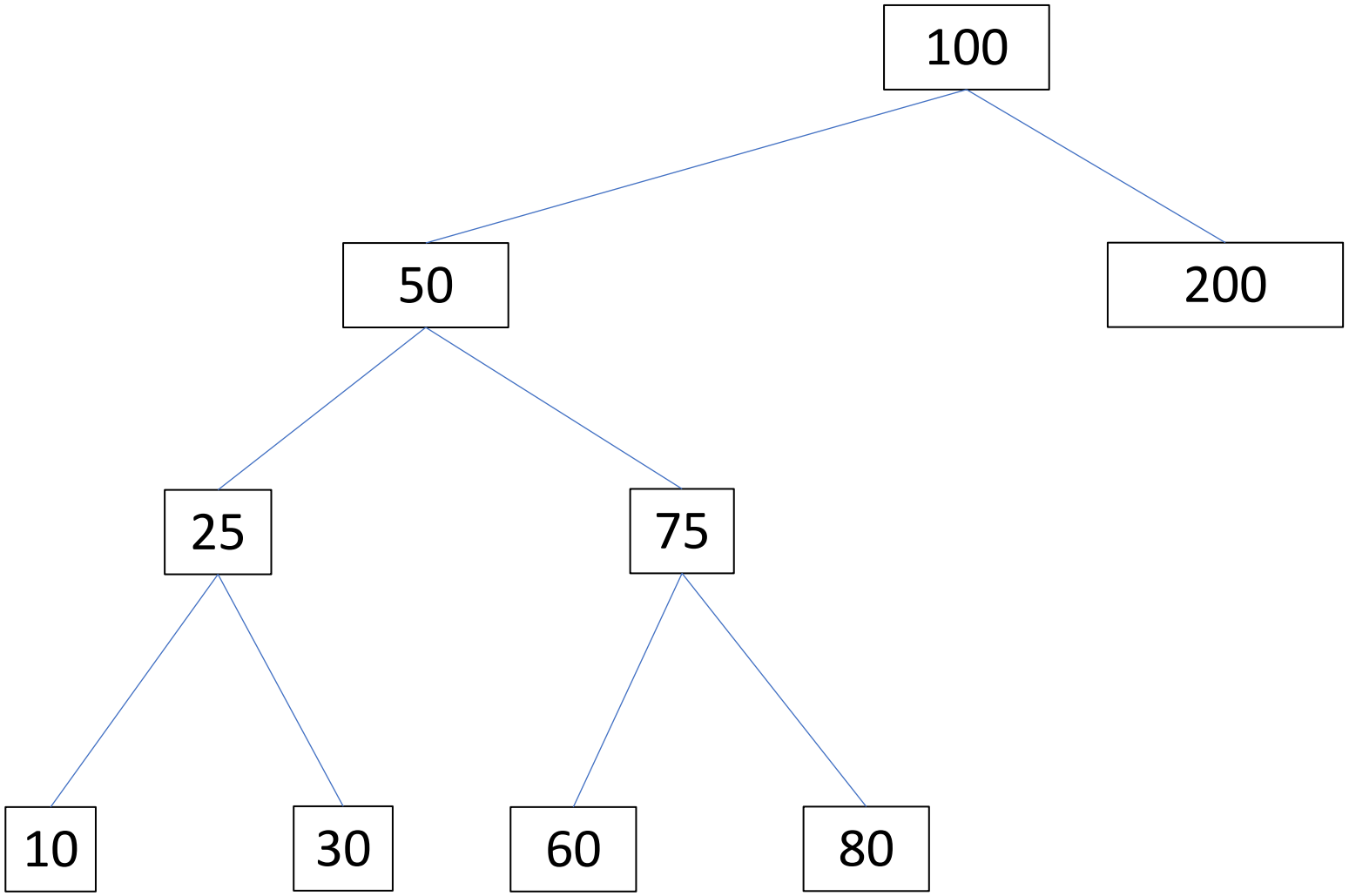
  - worklist

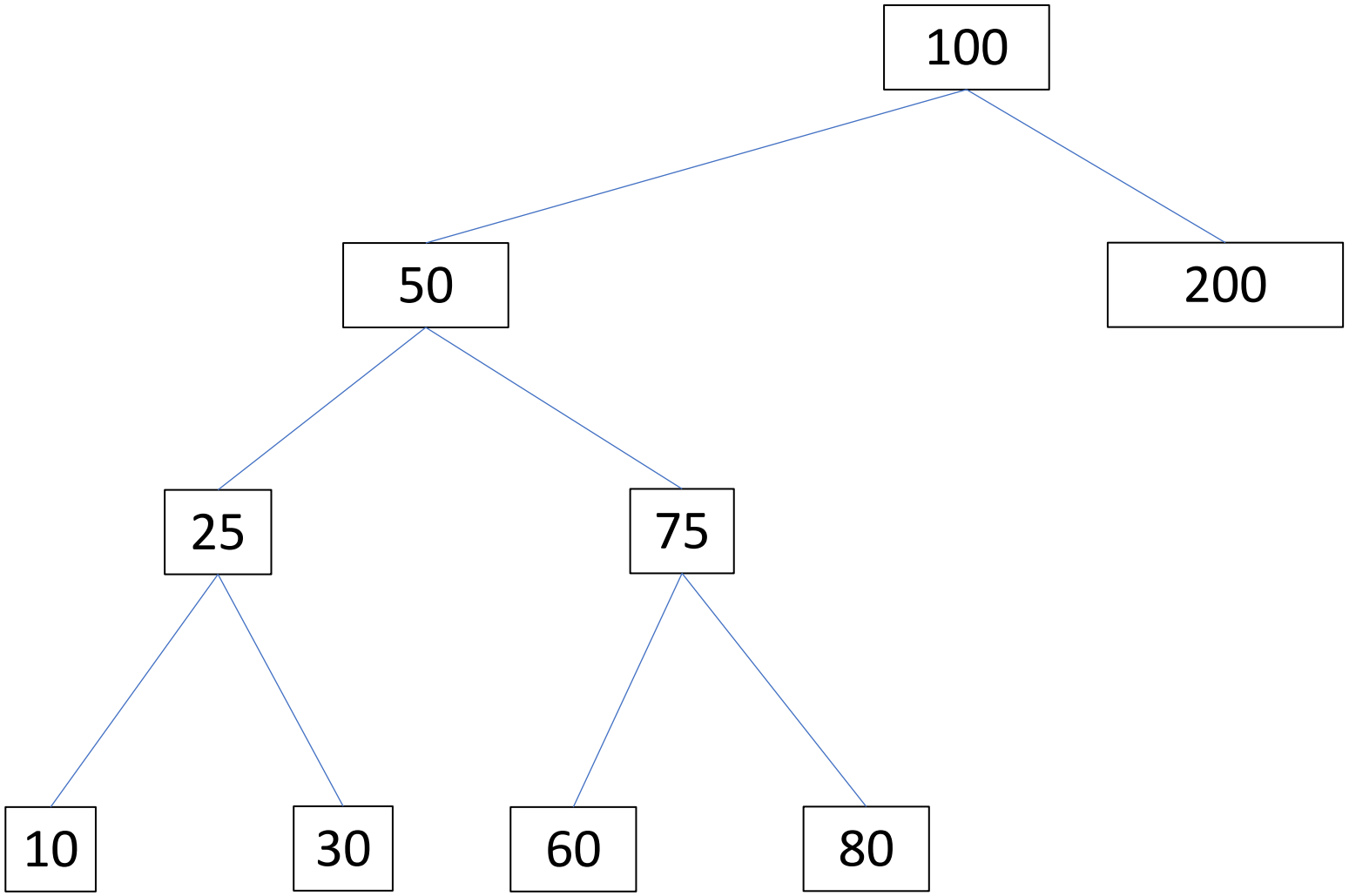
next lecture

accumulator design recipe (htdf + this)

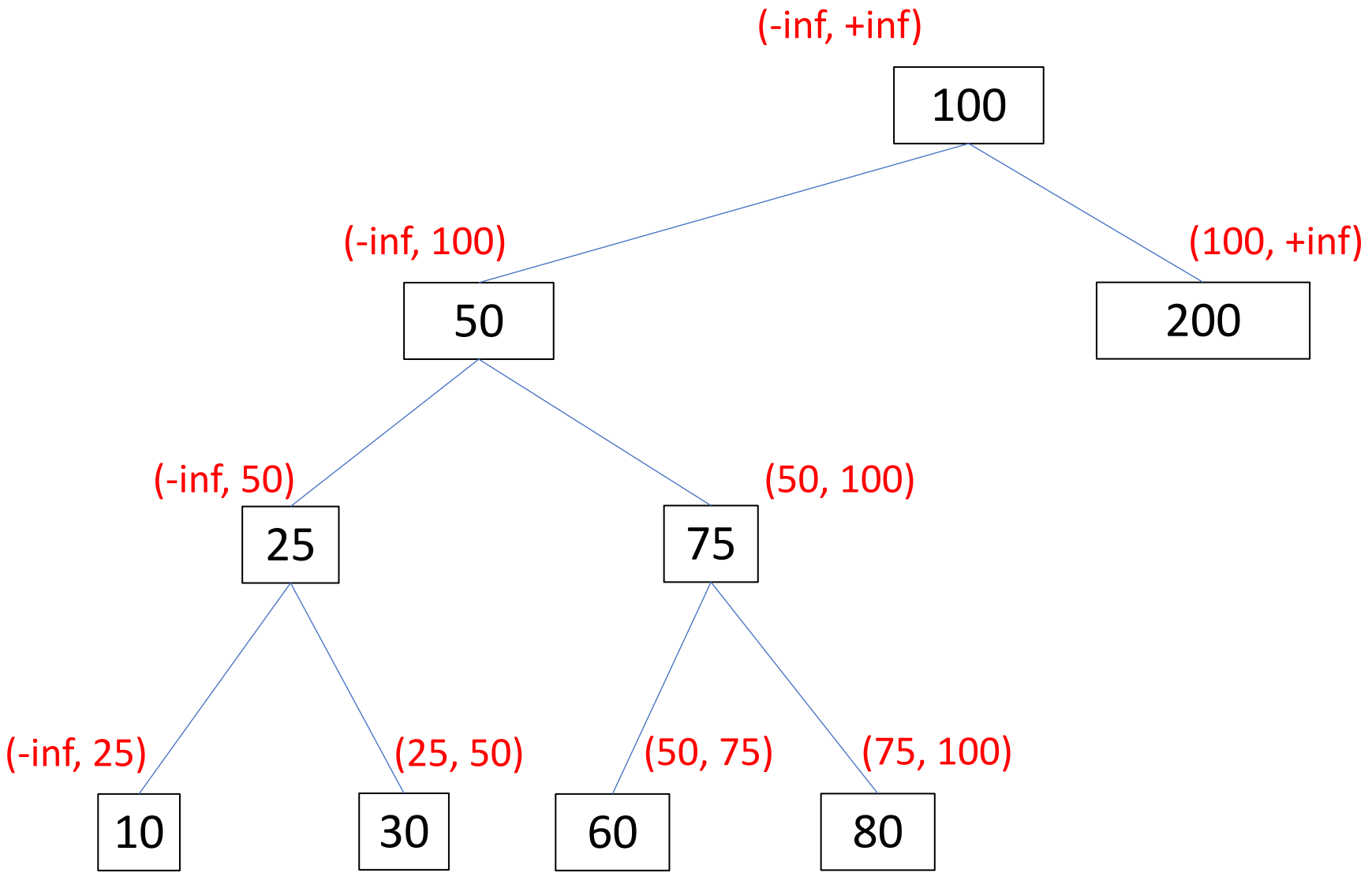
- templating:
  - recursive template wrapped in local and top-level function
  - add acc(umulator) parameter(s) to inner function
    - add acc after all … (can use better name)
    - add (… acc) in recursive call
    - add … in trampoline
- work out example progression of recursive calls
- wish for what the accumulator should be at the end
- work backward through progression to get accumulator at each step
- design type and invariant (may need a new data definition)
- initialize invariant, preserve invariant, exploit invariant
- test and debug

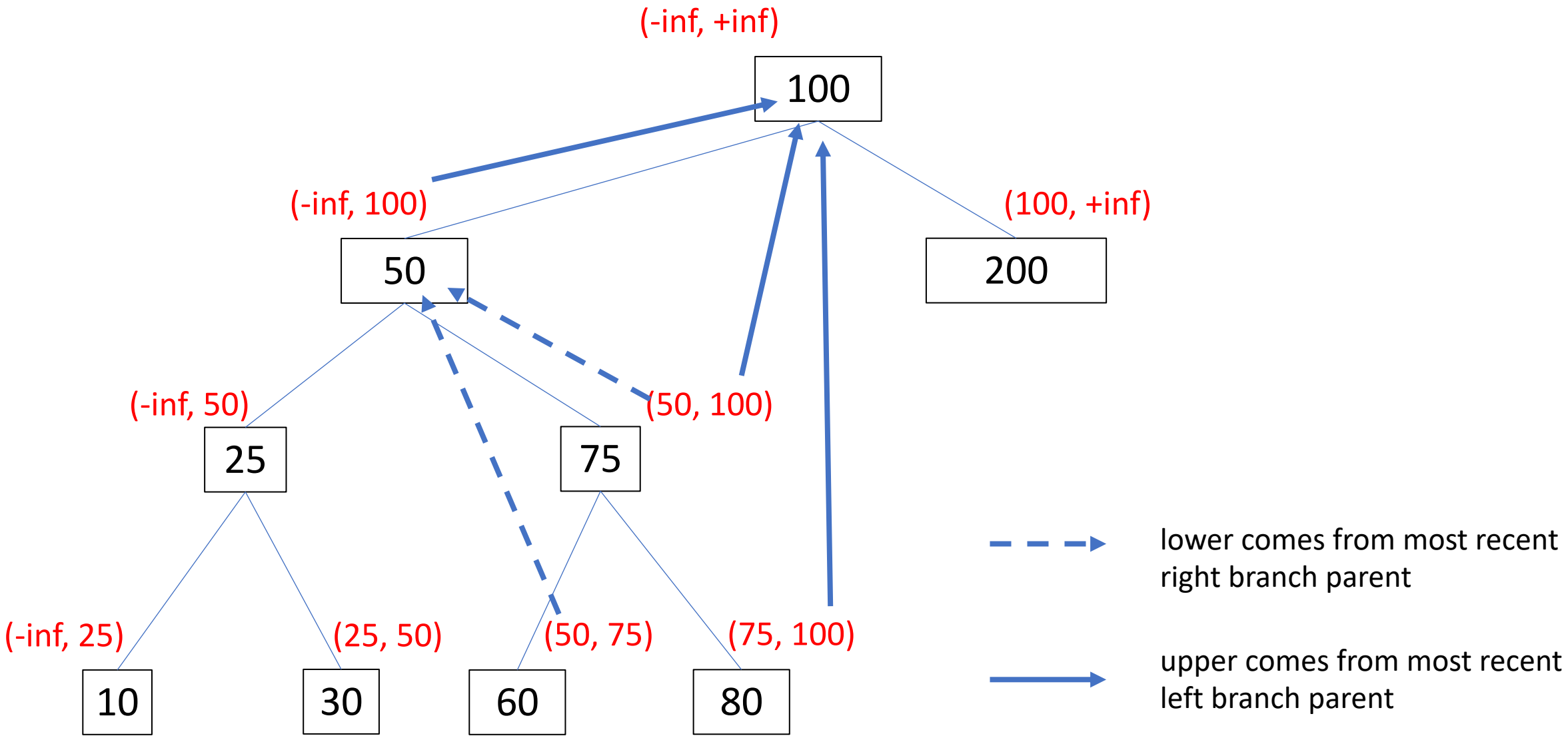
```
(@htdf sequence?)
(@signature (listof Natural) -> Boolean)
;; produces true if every element of lon is 1 greater than prior element
(check-expect (sequence? (list))       true)
(check-expect (sequence? (list 2))     true)
(check-expect (sequence? (list 2 3 4)) true)
(check-expect (sequence? (list 3 5 6)) false)
(check-expect (sequence? (list 2 3 4 7 5)) false)

(@template-origin (listof Natural) accumulator)

(define (sequence? lon0)
  ;; prev is  Natural
  ;; invariant: the element of lon0 immediately before (first lon)
  ;; (sequence? (list 2 3 4 7 5))

  ;; (sequence? (list   3 4 7 5) 2)
  ;; (sequence? (list     4 7 5) 3)
  ;; (sequence? (list       7 5) 4) ==> false
  (local [(define (sequence? lon prev)
            (cond [(empty? lon) true]
                  [else
                   (if (= (first lon) (+ 1 prev)) ;exploit (use)
                       (sequence? (rest lon)
                                  (first lon))    ;preserve
                       false)]))]

    (if (empty? lon0)                 ;if original list is empty, we can't
        true                          ;initialize accumulator, so special case
        (sequence? (rest lon0)
                   (first lon0)))))   ;initialize
```


100
50
200
25
75
10
30
60
80



Why is 80 OK here?
What range of numbers is OK here?


(-inf, +inf)
100
(-inf, 100)
50
(100, +inf)
200
(-inf, 50)
25
(50, 100)
75
(-inf, 25)
10
(25, 50)
30
(50, 75)
60
(75, 100)
80


(-inf, +inf)
100
(-inf, 100)
(100, +inf)
50
200
(-inf, 50)
(50, 100)
25
75
(-inf, 25)
(25, 50)
(50, 75)
(75, 100)
10
30
60
80
lower comes from most recent right branch parent
upper comes from most recent left branch parent

```
(@htdf bst?)
(@signature BinaryTree -> Boolean)
;; produce true if bt satisfies binary search tree invariants
(check-expect (bst? BT1) true)
(check-expect (bst? BT2) false)
(check-expect (bst? BT3) false)
(check-expect (bst? BT4) false)
(check-expect (bst? BT5) false)

(@template-origin BinaryTree accumulator)

(define (bst? bt0)
  ;; lower is Integer
  ;;   lower bound of key at current node
  ;;   initially -inf.0 is reset on recursions down a right branch
  ;; upper is Integer
  ;;   upper bound of key at current node
  ;;   initially +inf.0 is reset on recursions down a left branch
  ;;
  ;;
  (local [(define (bst? bt lower upper)
            (cond [(false? bt) true]
                  [else
                   (and (< lower (node-k bt) upper)                  ;exploit
                        (bst? (node-l bt)  lower        (node-k bt)) ;preserve
                        (bst? (node-r bt)  (node-k bt)  upper))]))]  ;preserve

    (bst? bt0 -inf.0 +inf.0)));                                      ;initialize
;;                            ;NOTE that we would never expect you to have
;;                            ;     already known about these two constants!
```