Forth Language Review and Stack Based Tree Sort, for numbers and Strings

paulr

Author: Paul Richeson **Platform:** Raspberry Pi (Linux 6.18, aarch64) Â· pforth 2.2.0 (version code 34) **Date:** 2026-06-01 **License:** MIT â see `LICENSE`.

This combined report studies a binary search-tree (BST) sort implemented in Forth on the *pforth* engine, from three complementary angles. **Part I** introduces the Forth language itself â its two-stack execution model, postfix syntax, dictionary of words, and interactive compilation â and verifies the model with small programs. **Part II** documents how the application is turned into a runnable binary: in Forth, "compiling" means loading source into a live interpreter and snapshotting the dictionary to an image (`treesort.dic`) with `save-forth`, a process orchestrated by GNU Make. **Part III** analyses the architecture of the tree-sort from a computer-engineering standpoint: a single index-addressed node arena, iterative insertion carried on the data stack, recursive in-order traversal carried on the return stack, yielding a stable, swap-free, exactly-N-allocated sort with average O(N log N) comparisons. A character-sorting sibling (`treesortstring.fth`) demonstrates that the mechanism is independent of element representation.

Index Terms â Forth, stack machine, postfix notation, dictionary image, `save-forth`, GNU Make, binary search tree, tree sort, data stack, return stack, in-order traversal, stability, pforth.

This report introduces Forth, a stack-oriented, extensible, interactively compiled programming language, and demonstrates its core mechanics on the *pforth* engine running under Linux on a Raspberry Pi. We describe the two-stack execution model, postfix (Reverse Polish) syntax, the dictionary of *words*, and the readâevaluate loop that lets a programmer extend the language itself. Worked examples (`hello.fth`, `demo.fth`) confirm the expected behaviour of arithmetic, user-defined words, and counted loops. We conclude that Forth's minimal, self-hosted design makes it a useful pedagogical and embedded-systems tool that exposes the machine-level notion of a stack directly to the programmer.

Index Terms â Forth, stack machine, postfix notation, threaded code, interpreter, embedded systems, pforth.

Most mainstream languages hide the evaluation stack behind an expression parser. Forth, designed by Charles Moore in the late 1960s, does the opposite: the *data stack* is the central, visible object of

computation, and the program is written in the order the stack is manipulated. This makes Forth unusually small (a complete system fits in a few kilobytes) and unusually transparent â there is almost no distance between the source text and the operations the processor performs. Forth is consequently popular in firmware, bootloaders (Open Firmware), and resource-constrained embedded systems.

The objective of this lab is to (1) state Forth's execution model, (2) explain its syntax and extensibility, and (3) verify both with small programs on the pforth engine.

A Forth system maintains two LIFO stacks:

- the **data stack** (or *parameter stack*), which holds operands and results;
- the **return stack**, which holds return addresses for nested word calls and

  loop indices.

All arithmetic and logic consume their operands from the data stack and push their results back onto it. This is the same discipline a compiler's code generator uses internally, exposed here as the programming model itself.

Because operands must already be on the stack before an operator runs, Forth is written postfix:

```
3 4 +        push 3, push 4, add -> 7 on the stack
```

There is no operator precedence and there are no parentheses; evaluation order is simply left-to-right. The phrase `3 4 + .` pushes 3 and 4, replaces them with their sum, then `.` prints and pops the top of the stack.

Every token in Forth is a **word**. A word is an entry in the **dictionary**, a linked list of named, executable definitions. Built-in words such as `+`, `dup`, `.`, and `cr` are present at start-up; the programmer adds new words with the *colon definition*:

```
: square ( n -- n^2 )    dup * ;
```

`:` begins a definition, `square` names it, the body `dup *` duplicates the top of stack and multiplies, and `;` ends the definition. The comment `( n -- n^2 )` is a **stack effect diagram**: inputs left of `--`, outputs right. New words are indistinguishable from built-ins and may be used to build further words â Forth programs are deep towers of tiny definitions.

pforth runs a readâevaluate loop (the *outer interpreter*). In *interpret* state it executes each word as it is read; on encountering `:` it switches to *compile* state and appends the following words to a new dictionary entry until `;`. The same text stream therefore both defines and runs code, which is why Forth is described as *interactively compiled* rather than purely interpreted or purely compiled.

- **Hardware:** Raspberry Pi, Linux 6.18 (aarch64).
- **Engine:** `pforth_standalone`, a portable ANS-style Forth written in C

  (pforth 2.2.0). It is driven through the convenience wrapper `./forth`, which simply executes the engine with the `-q` (quiet) flag.
- **Programs under test:**

  * `hello.fth` â minimal output program. * `demo.fth` â a tour of arithmetic, colon definitions, composition, and a counted loop.

Each file was executed with `./forth <file.fth>`.

```
bye
```

`."` compiles a string literal that is printed when reached; `cr` emits a newline; `bye` exits the interpreter. **Observed output:** `Hello, world!` followed by a newline, then a clean exit.

Key fragments and their behaviour:

Source	Stack action	Output
`3 4 + .`	push 3, push 4, add, print	`7`
`: square ( n -- n^2 ) dup * ;` then `9 square .`	define, then 9Â·9	`81`
`: average ( a b -- avg ) + 2 / ;` then `10 20 average .`	(10+20)/2	`15`
`5 0 do i . loop`	counted loop, index `i`	`0 1 2 3 4`

The `do â¦ loop` construct uses the return stack to hold its limit and index; `i` copies the current index to the data stack. All results matched the predicted values, confirming the stack model.

Two observations are worth emphasising from an engineering standpoint:

1. **Composition without syntax.** `average` is built from `+` and `/` with no declaration overhead â the cost of an abstraction in Forth is one dictionary entry, not a function-call frame as in conventional ABIs. This is why Forth scales *down* so well onto small processors. 2. **The stack is the contract.** The stack-effect comment is not enforced by a type system; correctness depends on each word leaving the stack as advertised. This places discipline on the programmer but removes an entire layer of machinery (parser tables, type checker, register allocator) from the system.

Forth's identity is its visible two-stack machine, postfix evaluation, and a dictionary of composable words extended through interactive compilation. The test programs behaved exactly as the model predicts. Because the language is self-hosted and minimal, it doubles as a clear teaching model of how a stack machine evaluates expressions â the same foundation reused in the companion labs on building (Lab 2) and on the binary-tree sort (Lab 3).

This report documents how the `treesort` program is turned from Forth source into a runnable binary on the pforth engine. Unlike a C tool-chain, "compiling" in Forth means loading source into a live interpreter and then *snapshotting the resulting dictionary* to disk as a saved image (`treesort.dic`). We analyse the project `Makefile`, the dependency graph it encodes, the role of the C-compiled engine `pforth_standalone`, and the shell launcher that assembles a one-shot Forth driver at run time. We verify that `make` produces a 122,776-byte dictionary image and that `./treesort 5 3 8 1 5 9` runs against it. The exercise illustrates the *image-based* execution model that distinguishes Forth (and Smalltalk/Lisp) from conventional ahead-of-time compilation.

Index Terms â build system, GNU Make, Forth dictionary image, `save-forth`, incremental dependencies, embedded tool-chain.

A "binary" in a C project is machine code emitted by a compiler and linked against libraries. In a Forth project there are *two* artefacts:

1. the **engine** â `pforth_standalone`, native ARM machine code produced once from the pforth C sources; and 2. the **dictionary image** â `treesort.dic`, a saved snapshot of the Forth interpreter's memory *after* our source has been loaded into it.

The application is the image; the engine is the virtual machine that loads and runs it. This lab traces how the project `Makefile` orchestrates both, and how a thin shell script presents the result as an ordinary command-line tool.

- **Build tool:** GNU `make`.
- **Forth wrapper:** `./forth` â `pforth/platforms/unix/pforth_standalone -q`.
- **Sources / artefacts:**

  * `treesort.fth` â application source (the tree-sort words). * `pforth_standalone` â the native engine (423,928 bytes). * `treesort.dic` â the saved dictionary image (122,776 bytes). * `treesort` â shell launcher that feeds command-line numbers to the image.

The `Makefile` declares the following targets and prerequisites:

```
all  âââ° treesort.dic
treesort.dic : treesort.fth  pforth_standalone        (rebuild image)
pforth_standalone :                                   (build engine from C)
run  : treesort.dic                                   (build, then demo)
clean / distclean                                     (remove artefacts)
```

The arrows are Make's notion of *staleness*: `treesort.dic` is rebuilt whenever either the application source `treesort.fth` **or** the engine binary changes, because both are listed as prerequisites on line 27. This guarantees the image always reflects the current source and is never linked against a stale engine.

If `pforth_standalone` is missing, the rule

```
$(ENGINE):
    $(MAKE) -C pforth/platforms/unix pforth_standalone
```

recurses into the pforth platform directory and performs a *conventional* C build (the `.o`/`.eo` object files visible there are its intermediate output). This is the only step that produces native ARM machine code. On this system the engine was already present (423,928 bytes), so this rule was satisfied.

The central rule is:

```
$(DIC): $(SRC) $(ENGINE)
    @printf 'include $(SRC)0" $(DIC)" save-forth0ye0 | $(FORTH) >/dev/null
```

Three Forth commands are piped into the engine on standard input:

1. `include treesort.fth` â the outer interpreter reads the source and *compiles every colon definition into the live dictionary* (`init`, `feed`, `inorder`, `report`, etc.). After this line, the running image "knows" the tree-sort vocabulary. 2. `c" treesort.dic" save-forth` â `c"` pushes the counted-string filename; `save-forth` serialises the **entire current dictionary** (built-ins + our new words) to that file. *This is the act of "creating the binary."* No code generation or linking occurs; memory is written verbatim to disk. 3. `bye` â exit the engine.

The result, `treesort.dic`, is a self-contained image that, when loaded by the engine, restores the interpreter to exactly the state it had immediately after `include` â with all of our words defined and ready to call.

```
$ make
Compiling treesort.fth -> treesort.dic ...
Built treesort.dic
$ ls -l treesort.dic      # 122776 bytes
```

The image is ˜123 KB, larger than the 3.4 KB source because it contains the full base dictionary plus our additions â the snapshot is of the *whole interpreter*, not just our file.

The application is presented to the user by the `treesort` shell script, which performs three jobs:

1. **Validate input** â prints usage and exits if no numbers are supplied. 2. **Synthesise a driver** â writes a temporary `.fth` file (auto-deleted via a `trap â¦ EXIT`) containing: ‘‘‘ <N> init size the tree to N nodes <n1> feed <n2> feed â¦ insert each command-line number report draw the tree and print the sorted output bye ‘‘‘ where `N = $#` is the argument count. 3. **Execute against the image** â runs `pforth_standalone -q -d treesort.dic <driver>`; the `-d` flag tells the engine to **load our saved dictionary** instead of the default. A `grep` filter suppresses pforth's harmless include/level banner so only the program output is shown.

Thus `./treesort 5 3 8 1 5 9` boots the saved image, runs the generated driver, and prints the narrated tree and sorted result. (An alternate launcher, `treesortold`, skips the saved image and `includes` the source at run time instead â slower to start, but needing no `make` step.)

From a computer-engineering standpoint this build pipeline is an example of **image-based deployment**: program state, not just program text, is the shippable artefact. Advantages observed:

- **Fast start-up** â the application is already compiled inside the image, so
  no parsing happens at launch; the `-d` load is essentially an `mmap`-and-go.
- **Reproducibility** â the same source + engine always yields an equivalent
  image, and Make's prerequisite list enforces that invariant.
- **Separation of concerns** â the native engine (portable C, built rarely) is
  cleanly decoupled from the application image (built on every source edit).

A limitation is portability of the image: a `.dic` is tied to the engine's word layout and host word size, so it is not interchangeable across architectures â the source and a matching engine are the true portable units.

"Compiling" the Forth tree-sort means loading `treesort.fth` into the pforth interpreter and saving the resulting dictionary with `save-forth`, producing the 122,776-byte `treesort.dic` image. GNU Make sequences the optional engine build and the image build via an explicit dependency graph, and a shell launcher turns the image into a normal CLI tool by generating a per-invocation Forth driver. The process cleanly demonstrates Forth's image-based execution model.

This report analyses, from a computer-engineering perspective, the architecture of a binary search-tree (BST) sort implemented in Forth (`treesort.fth`) and its character-sorting sibling (`treesort-string.fth`). We examine the in-memory data structure (a single contiguous arena of fixed-width nodes addressed by integer indices), the mapping of tree operations onto Forth's data and return stacks, and the cost of each primitive in terms of stack traffic and memory accesses. We show that the algorithm performs **zero element swaps**, is **stable**, and allocates **exactly N nodes** with no resizing, and we relate these properties to the underlying stack-machine execution model. Empirical runs confirm O(N log N) average comparison counts (e.g. 9 comparisons for the 6-element input `sortme`) and worst-case O(NÂ²) on sorted input. The study illustrates how a pointer-based algorithm is expressed on a machine whose only addressable working storage is two LIFO stacks plus a heap allocated explicitly by the program.

Index Terms â binary search tree, tree sort, stack machine, data stack, return stack, recursion, in-order traversal, stability, memory arena, pforth.

Sorting on a stack machine raises an architectural question: the natural working store is a LIFO stack, yet a tree is a pointer structure with arbitrary, non-LIFO access. This lab dissects how `treesort.fth` resolves that tension. The algorithm builds a binary search tree by *insertion* and then performs an *in-order traversal* to emit a sorted sequence. We treat the implementation as a small machine architecture: a memory subsystem (the node arena), an addressing scheme (integer indices), a control unit (the colon-defined words), and a working-register set (the two Forth stacks).

All nodes live in **one contiguous block** obtained from the Forth heap with `ALLOCATE`, sized once for exactly *N* inputs:

```
: init ( n -- )   3 * cells allocate ... base !  0 #nodes !  -1 root !  0 #cmp ! ;
```

Each node is **three cells wide**:

```
index i â  [ value | left-child-index | right-child-index ]
              +0        +1                  +2        (cells)
```

A child field holding **-1** denotes *nil* (no child). The root index is held in the variable `root` (also -1 when the tree is empty), and `#nodes` is both the count of nodes used and the **next free index** â allocation is a simple bump of this counter.

Children are referenced by **small integer indices**, not full machine addresses. The accessor words convert an index to the address of a field:

```
: n>val   ( i -- a )  3 *      cells base @ + ;
: n>left  ( i -- a )  3 * 1+   cells base @ + ;
: n>right ( i -- a )  3 * 2 +  cells base @ + ;
```

Architecturally this is **base-plus-scaled-index addressing**: ‘address = base + (3Â·i + field)Â·cell-size‘. Two engineering benefits follow:

1. **Compactness** â an index is typically smaller than a pointer and is independent of where the arena was mapped, which matters on memory-constrained embedded targets. 2. **Locality** â because all nodes share one block, traversal touches a dense region of memory, friendly to the data cache, rather than chasing heap-scattered pointers.

`new-node` writes the value, sets both child links to -1, and bump-allocates:

```
: new-node ( val -- i )  #nodes @  -1 over n>left !  -1 over n>right !  2dup n>val !  nip
```

There is **no free list and no resizing**: the arena is sized once to N, and exactly N nodes are consumed. Memory use is therefore deterministic and known before the first insert â a desirable property for real-time/embedded use.

`feed ( val -- )` inserts one key. Its control state â the value being inserted and the "current node" walking pointer (`cur`) â lives on the **data stack** and in a variable, not on a call stack:

- If `root = -1`, the value becomes the root (one `new-node`, store to `root`).
- Otherwise a `begin â¦ again` loop walks down the tree:

 * each iteration increments `#cmp` (a comparison counter) and compares the key with `cur`'s value; * `key < node` â go **LEFT**; otherwise go **RIGHT**. Equal keys go right, which is what makes the sort **stable** (an earlier-inserted equal key is encountered first by the in-order walk and therefore emitted first); * when the chosen child link is -1, a new node is attached there and the loop exits via `drop exit`.

Because insertion is written as an explicit loop, it uses **bounded stack depth** regardless of tree height â only a few cells of data-stack scratch are live at a time. This is the architecturally interesting choice: descent is expressed as *data-stack manipulation*, not recursion.

Emitting the sorted output is naturally recursive (left subtree, self, right subtree). Here the implementation **does** use recursion, and Forth's `recurse` word calls the current definition again:

```
: inorder ( i -- )
   dup -1 = if drop exit then
   dup n>left @ recurse          visit left subtree
   dup n>val @ .                 emit this node  (emit/.ch in the string version)
   n>right @ recurse ;           visit right subtree
```

Each `recurse` pushes a **return address onto the return stack**; the maximum return-stack depth equals the height *h* of the tree. This is the clearest illustration in the project of the **two-stack architecture at work**: operands and the node index ride the *data stack*, while the *return stack* records the suspended traversal frames. `show-tree` uses the same recursive pattern but visits right-before-left and indents by depth to draw the tree rotated 90Â°.

Classical comparison sorts (quicksort, heapsort) move array elements. This sort **never moves a value after it is placed** â ordering is encoded entirely in the `left`/`right` link fields written once at insertion. The reported metric "0 swaps" is therefore exact: total writes to value fields = N, and they are all initial writes.

Let *N* be the input size and *h* the resulting tree height.

Operation	Work	Stack traffic	Memory accesses
`'new-node'`	O(1)	a few data-stack ops	3 writes (value + 2 links)
one `'feed'`	O(depth of insert)	bounded data-stack scratch	~1 value read + 1 link re
full build	Î£ depths	â	average O(N log N), worst O(NÂ²)
`'inorder'`	O(N)	return-stack depth = h	1 value read + 2 link reads per node

- **Comparisons** are tracked live in `#cmp`. Measured example: input `sortme`

  (6 chars) built with **9 comparisons** â consistent with average-case O(N log N).

- **Worst case** is *already-sorted input*, which degenerates the BST into a

  linked list: insertion becomes O(NÂ²) comparisons and the return-stack depth of `inorder` reaches *N*. This is the standard architectural caveat of an unbalanced BST and would, on a deeply nested input, stress the return stack.

- **Space** is exactly `3Â·N` cells of arena plus a handful of variables â

  allocated once, never grown.

The string sorter is the *same architecture* with one substitution: a node's `value` cell holds a **character code** instead of an integer. Only the I/O boundary changes â values are read with `KEY`-style byte codes supplied by the launcher and printed with `EMIT` / a quoting helper `.ch` rather than the numeric `.`. The tree, the index addressing, the stability rule (equal characters go right), and the stack usage are byte-for-byte identical. This demonstrates a clean **separation between the sorting mechanism and the element's representation**: the data structure is type-agnostic because a Forth cell is just a machine word. Verified: `./treesortstring sortme â emorst` (6 characters, 9 comparisons, 0 swaps, stable).

The design choices map directly onto stack-machine strengths and weaknesses:

- **Strength â explicit memory.** Forth has no garbage collector; by sizing one

  arena up front and addressing it by index, the program achieves deterministic, cache-friendly memory behaviour appropriate for embedded targets.

- **Strength â minimal working set.** Iterative insertion keeps live data-stack

  depth tiny; recursion is confined to traversal, where its depth (tree height) is the genuinely required state.

- **Weakness â no self-balancing.** Height is input-dependent; adversarial

  (sorted) input makes both time and return-stack depth linear. A production design would add balancing (AVL/red-black), at the cost of rotation logic and more link writes â trading the "0 swaps / 0 rotations" simplicity for worst-case guarantees.

The Forth tree-sort is a compact, pointer-structured algorithm realised on a two-stack machine: a single index-addressed node arena provides the heap, the **data stack** carries iterative insertion state, and the **return stack** carries recursive-traversal frames. The architecture yields a stable, swap-free, exactly-N-allocated sort with average O(N log N) comparisons (empirically 9 for N=6) and the well-understood O(NÂ²) unbalanced-BST worst case. The string variant shows that the mechanism is independent of element representation. The result is an instructive bridge between abstract data-structure theory and the concrete behaviour of a stack-based processor.

[1] L. Brodie, *Starting Forth*, 2nd ed. FORTH, Inc. [2] L. Brodie, *Thinking Forth*. Punchy Publishing. [3] ANSI X3.215-1994, *Programming Language â Forth*. [4] P. Burk et al., *pforth â a Portable Forth in C*, v2.2.0. [Online]. Available: https://github.com/philburk/pforth [5] R. M. Stallman et al., *GNU Make Manual*. Free Software Foundation. [6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. MIT Press. [7] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed. Addison-Wesley. [8] P. J. Koopman, *Stack Computers: The New Wave*. Ellis Horwood.

The project is built with GNU Make. Note that the pforth engine itself was **compiled from C source** (`make engine`, which invokes the engine's own Makefile under `pforth/platforms/unix/`); the resulting `pforth_standalone` binary is then used to "compile" the Forth source by loading it and saving the dictionary image `treesort.dic`. In Forth, "compiling" means loading source into the interpreter and snapshotting the resulting dictionary as a binary image â that image is the compiled program.

```
# Makefile for the Forth tree-sort.
#
# "Compiling" in Forth means loading your source into the interpreter and
# saving the resulting dictionary as a binary image. That image (treesort.dic)
# is the compiled program; you then run it on the pforth engine.
#
# Targets:
#   make            build treesort.dic   (the compiled Forth image)
#   make run        build, then run the demo numbers
#   make engine     (re)build the pforth engine from C source
#   make clean      remove the compiled image
#   make distclean  also clean the pforth engine build

# Where things live. (Keep these clean: a trailing comment would leak
# whitespace into the value, so comments go on their own lines.)
FORTH  := ./forth
ENGINE := pforth/platforms/unix/pforth_standalone
SRC    := treesort.fth
DIC    := treesort.dic
DEMO   := 5 3 8 1 5 9

# Default: build the compiled image.
all: $(DIC)

# Compile the Forth source into a saved dictionary image.
# Feed three lines to the engine: include the source, save the image, quit.
$(DIC): $(SRC) $(ENGINE)
    @echo "Compiling $(SRC) -> $(DIC) ..."
    @printf 'include $(SRC)0" $(DIC)" save-forth0ye0 | $(FORTH) >/dev/null
    @echo "Built $(DIC)"

# Build the pforth engine from its C sources if it is missing.
$(ENGINE):
    @echo "Building pforth engine ..."
    $(MAKE) -C pforth/platforms/unix pforth_standalone

engine: $(ENGINE)

# Run the demo.
run: $(DIC)
    ./treesort $(DEMO)

# Cleanup.
clean:
    rm -f $(DIC)

distclean: clean
    -$(MAKE) -C pforth/platforms/unix clean
```

Note on the toolchain: pforth was not installed from a package â it was *built from sources shipped in the* `pforth/` subtree. The `$(ENGINE)` target *above delegates to pforth's own Unix-platform Makefile to produce the* `pforth_standalone` C binary, which serves as the host interpreter for every *subsequent Forth compile/run step in this report.*

```
 treesort.fth - stable binary-tree sort with a visible, narrated process.
 Why tree sort fits the request:
   * 0 swaps      - elements are never swapped; they are linked into a tree.
   * stable       - equal keys are sent RIGHT, so earlier inputs stay earlier
                    in the in-order traversal.
   * RAM-efficient- we ALLOCATE exactly N nodes (3 cells each), no slack,
                    no resizing. Pointers are small integer indices, not
                    full addresses.
 A node lives at index i in one contiguous block, 3 cells wide:
     [ value | left-child-index | right-child-index ]
 A child index of -1 means "no child" (nil).

variable base       address of the node block
variable #nodes     nodes used so far  (also the next free index)
variable root       index of the root node, -1 when tree is empty
variable cur        walking pointer used during insertion
variable #cmp       total key comparisons performed

 ---- field accessors: index -> address of that field ------------------
: n>val   ( i -- a )  3 *      cells base @ + ;
: n>left  ( i -- a )  3 * 1+   cells base @ + ;
: n>right ( i -- a )  3 * 2 +  cells base @ + ;

 ---- set up storage for exactly n inputs ------------------------------
: init ( n -- )
   3 * cells allocate            ( a-addr ior )
   if ." !! ALLOCATE failed" cr then
   base !
   0 #nodes !  -1 root !  0 #cmp !
;

 ---- create a fresh leaf node holding val, return its index -----------
: new-node ( val -- i )
   #nodes @                      ( val i )
   -1 over n>left  !
   -1 over n>right !
   2dup n>val !
   nip
   1 #nodes +!
;

 ---- insert one value, narrating every comparison ---------------------
: feed ( val -- )
   dup ." . add " . ." :" cr
   root @ -1 = if
      new-node root !
      ."     tree was empty -> becomes the root" cr
      exit
   then
   root @ cur !
   begin
      1 #cmp +!
      ."     compare " dup .
      ." with node " cur @ n>val @ .
```

```
      dup cur @ n>val @ < if
         ."  -> smaller, go LEFT" cr
         cur @ n>left @ -1 = if
            dup new-node cur @ n>left !
            ."       attached as LEFT child of " cur @ n>val @ . cr
            drop exit
         else
            cur @ n>left @ cur !
         then
      else
         ."  -> not smaller, go RIGHT (equal keys stay stable)" cr
         cur @ n>right @ -1 = if
            dup new-node cur @ n>right !
            ."       attached as RIGHT child of " cur @ n>val @ . cr
            drop exit
         else
            cur @ n>right @ cur !
         then
      then
   again
;

 ---- draw the tree sideways (rotate 90 deg): root at far left, --------
 ---- larger values toward the top. Indentation = depth.        --------
: show-tree ( i depth -- )
   over -1 = if 2drop exit then
   2dup swap n>right @ swap 1+ recurse      right subtree above
   dup 4 * spaces  over n>val @ .  cr       this node
   over n>left @ over 1+ recurse            left subtree below
   2drop
;

 ---- in-order walk: left, self, right -> ascending, stable -----------
: inorder ( i -- )
   dup -1 = if drop exit then
   dup n>left @ recurse
   dup n>val @ .
   n>right @ recurse
;

: report
   cr ." ===== binary tree (rotated 90 deg: root at left, larger on top) =====" cr
   root @ 0 show-tree
   cr ." ===== in-order traversal = sorted output =====" cr
   ." sorted: " root @ inorder cr
   #nodes @ . ." numbers, " #cmp @ . ." comparisons, 0 swaps, stable" cr
;
```

```
 treesortstring.fth - stable binary-tree sort of the CHARACTERS of a string,
 with a visible, narrated process.  This is the sibling of treesort.fth: same
 tree, same algorithm, same guarantees - only the "value" in each node is now
 a character code instead of a number, and it is shown as the character.
 Why tree sort fits the request:
   * 0 swaps      - characters are never swapped; they are linked into a tree.
   * stable       - equal characters are sent RIGHT, so an earlier occurrence
                    stays earlier in the in-order traversal.
   * RAM-efficient- we ALLOCATE exactly N nodes (3 cells each), no slack,
                    no resizing. Pointers are small integer indices, not
                    full addresses.
 A node lives at index i in one contiguous block, 3 cells wide:
     [ char-code | left-child-index | right-child-index ]
 A child index of -1 means "no child" (nil).

variable base       address of the node block
variable #nodes     nodes used so far  (also the next free index)
variable root       index of the root node, -1 when tree is empty
variable cur        walking pointer used during insertion
variable #cmp       total key comparisons performed

 ---- print a character code as a quoted character, e.g.  'a' ----------
: .ch ( c -- )  [char] ' emit emit [char] ' emit space ;

 ---- field accessors: index -> address of that field ------------------
: n>val   ( i -- a )  3 *      cells base @ + ;
: n>left  ( i -- a )  3 * 1+   cells base @ + ;
: n>right ( i -- a )  3 * 2 +  cells base @ + ;

 ---- set up storage for exactly n inputs ------------------------------
: init ( n -- )
   3 * cells allocate            ( a-addr ior )
   if ." !! ALLOCATE failed" cr then
   base !
   0 #nodes !  -1 root !  0 #cmp !
;

 ---- create a fresh leaf node holding the char code, return its index --
: new-node ( c -- i )
   #nodes @                      ( c i )
   -1 over n>left  !
   -1 over n>right !
   2dup n>val !
   nip
   1 #nodes +!
;

 ---- insert one character, narrating every comparison -----------------
: feed ( c -- )
   dup ." . add " dup .ch ." :" cr
   root @ -1 = if
      new-node root !
      ."     tree was empty -> becomes the root" cr
      exit
```

```
   then
   root @ cur !
   begin
      1 #cmp +!
      ."     compare " dup .ch
      ." with node " cur @ n>val @ .ch
      dup cur @ n>val @ < if
         ."  -> smaller, go LEFT" cr
         cur @ n>left @ -1 = if
            dup new-node cur @ n>left !
            ."       attached as LEFT child of " cur @ n>val @ .ch cr
            drop exit
         else
            cur @ n>left @ cur !
         then
      else
         ."  -> not smaller, go RIGHT (equal chars stay stable)" cr
         cur @ n>right @ -1 = if
            dup new-node cur @ n>right !
            ."       attached as RIGHT child of " cur @ n>val @ .ch cr
            drop exit
         else
            cur @ n>right @ cur !
         then
      then
   again
;

 ---- draw the tree sideways (rotate 90 deg): root at far left, --------
 ---- larger characters toward the top. Indentation = depth.    --------
: show-tree ( i depth -- )
   over -1 = if 2drop exit then
   2dup swap n>right @ swap 1+ recurse      right subtree above
   dup 4 * spaces  over n>val @ .ch  cr      this node
   over n>left @ over 1+ recurse            left subtree below
   2drop
;

 ---- in-order walk: left, self, right -> ascending, stable -----------
: inorder ( i -- )
   dup -1 = if drop exit then
   dup n>left @ recurse
   dup n>val @ emit
   n>right @ recurse
;

: report
   cr ." ===== binary tree (rotated 90 deg: root at left, larger on top) =====" cr
   root @ 0 show-tree
   cr ." ===== in-order traversal = sorted output =====" cr
   ." sorted: " root @ inorder cr
   #nodes @ . ." characters, " #cmp @ . ." comparisons, 0 swaps, stable" cr
;
```