Scaling Tcl Data Structures

Don Porter
Ever seen this?

% lrepeat 600000000 a

max length of a Tcl list (536870909 elements) exceeded

% lindex a[string repeat { b} 540000000] 0

max length of a Tcl list (536870909 elements) exceeded
Tcl has Limits

- Most derive from UINT_MAX limit on memory.
  - calloc(unsigned int size);
- Many arguably not a bad idea.
- Still, shouldn't Tcl fail better?
  - Performing poorly at a task beats refusing it
  - Definitely beats bringing down the whole program.
- Tcl quits before memory does. Tcl 9 should not.
Dynamic Arrays – Tcl_DString

• The “Tcl String Growth Algorithm”
  – Have enough space? Use it!
  – No? Then reallocate at “double the size”.
  – Various refinements
    • Initial static allocation
    • Fallback attempts on failure.
    • Minimum growth rates.
It's not just for strings anymore!

- Same basic algorithm used over and over
  - Tcl_Tokens, OpNodes
  - Tcl_Obj * array in a list
  - Bytecode, ExceptionRanges, Literals
  - Etc. etc. etc.

- Some things are different
  - Hash Tables
  - OO stuff: methods, instances, chains, etc.
What's good about it?

• Delivers dynamic arrays.
  – No fixed limits other than memory allocation
• Lookup in constant time \([ O(1) ]\)
• Append in amortized constant time.
• At each moment, one pointer return to get all.
  – Tcl_DStringValue()
  – Tcl_ListObjGetElements(...&objv...)
What's not so good?

- Coded again and again and again and again.
- O(N) memory allocated but not used.
- Memory not released as needs shrink
  - “High water mark”
  - For many uses, this is correct.
- Returned pointers are not stable
Are there other dynamic arrays?

- More than I can even name in this talk.
- More than I know anything about.
Is there one you can tell us about?

- Yes!
- The “Brodnik Array”
Brodnik Array

- Store sequence of items as index array (store) of “data block” array of items (store[hi]).
- As index into index array grows, so does size of the data block array
  - store[0] holds one item at index 0.
  - store[2^n - 2] holds 2^n items at index 2^2n – 1
- Last data block is \( \sim \sqrt{\text{length of sequence}} \).
(Implicit) Index Map

set i 0; set hi 0; set lo 0; set ct 0; set size 1;
while 1 {puts "i $i → hi $hi, lo $lo"
    if { incr lo < $size } continue; set lo 0; incr hi
    if { $count == 0 } { set count $size
        set size [expr {2*$size}]; incr count $size
    }
    incr count -1
}
Explicit Index Map (sketch)

- Index in binary.
  - $77 = 1 (001) (101)$
  - $(101)$ is value of $lo = 5$.
  - Data block size is $2^3 = 8$.
  - From $(001)$ and $3$, compute $hi = 15$.
- Actual index value is offset by $1$.
  - Index $76 \rightarrow hi = 15$, $lo = 5$.
- Computation is $O(1)$
Growing and Shrinking

• As next store[i] is needed, allocate it.
• Keep no more than one unused store[i].
• Use String Growth Algorithm to grow store.
• When usage of store drops below $\frac{1}{4}$, shrink it by $\frac{1}{2}$. 
What's good about Brodnik?

- Only $O(\sqrt{N})$ unused allocation.
- Releases memory not used.
- Lookup in constant time.
- Append in amortized constant time.
- Data never moves. Pointers are stable!
- Coded just one time.
What's (maybe) not so good?

- **Unfamiliar**
  - Lack ability to “grab a pointer and go”
  - Either have to copy, or learn an interface.
- **Random access lookup needs 2 memory accesses instead of 1.**
  - Whether that matters is a complex question.
  - Random access lookup is not the most common.
- **Too much reallocation at small sizes**
  - Static foundation SMOP
Brodnik Pointers

• Relate to Brodnik Arrays as pointers relate to arrays.

• For sequential access, a BP remembers a location in a BA, so computing the next is cheap.

```c
T *t; BP_T p; BA_T *a;
for (t=BA_T_First(a, &p); t; t=BP_T_Next(&p))
```

• Contrast with

```c
T *t; T *p; T a[];
for (t=p=a; p < end; t = ++p)
```
Where is this off the mark?

- Brodnik Arrays are good for stack-like things.
- Not so good where we wish to insert and delete internal ranges from the sequence.
- So Tcl strings and lists really want something else.
- Got some ideas there, but even less fully realized than Brodnik arrays.
What's done?

• On dgp-refactor branch...
  – Set of BrodnikArray macros create BA of any type.
  – ...and BP to go with them.
  – Functional, with many interface routines working.
    • ...but still in flux.
  – Many simple dynamic array uses converted.
    • Reference, env Vars, CmdLocation, JumpFixup, etc. etc.
    • Tcl still runs, no obvious performance disasters
Next tasks

- Tcl_Token arrays
  - dgp-refactor already parses scripts, not commands
  - Stable pointers simplify code.
    wordIndex = parsePtr->numTokens;
    tokenPtr = &parsePtr->tokenPtr[wordIndex];
    … /* Grow the token array */ …
    tokenPtr = &parsePtr->tokenPtr[wordIndex];
Next tasks

• OpNode trees
  – Stored in dynamic array.
  – Child, parent links stored as indices into array.
  – If array grows to size_t size, and nothing changes, each OpNode doubles in size. (Each link, int → size_t)
  – Existing scheme is already wasteful. Index increments would be far more compact. (Each link, int → char !)
  – Stable pointers important to solution here too.
  – More detail at a WIP?
Concluding thoughts

- There are options beyond arrays and linked lists in C programming.
- Some are even useful!
- Coding once for re-use is good.
- Copy/paste/modify not so much.
- Fossil branches are good places to try things out and share them.
- There are more ideas than time.