New Abstract List Types

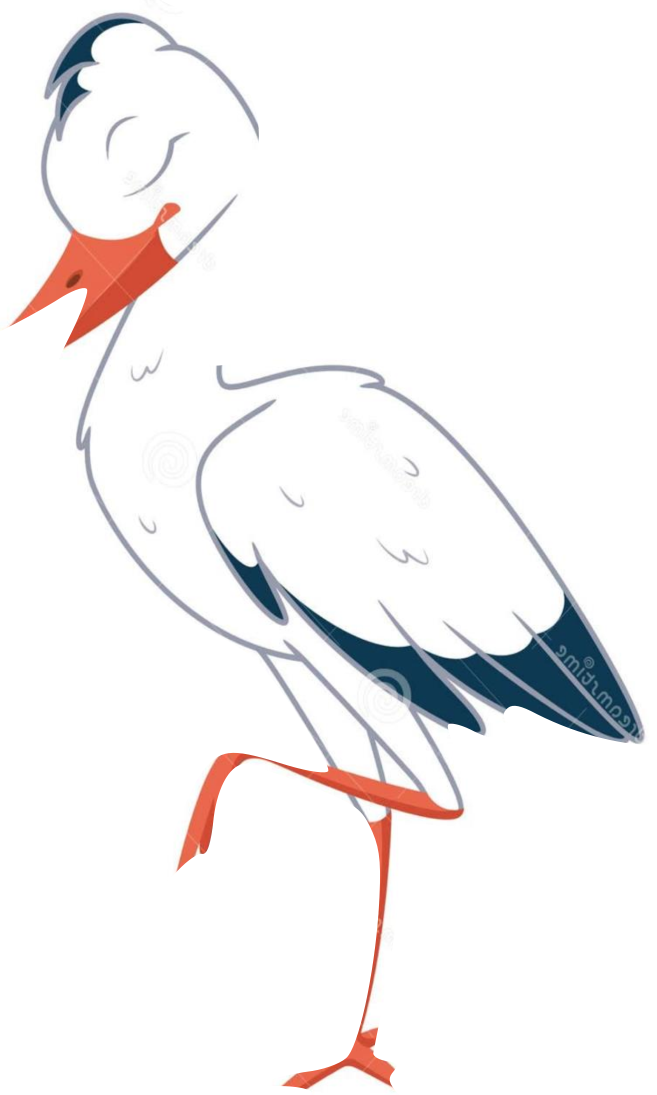
Improved efficiency for certain list patterns.

Ashok P. Nadkarni (author/implementor), Brian Griffin (presenter)

TCL List

```
0: ["One"   ]
1: ["Two"   ]
2: ["Three"]
3: ["Four" ]
```

- A collection of values stored in an C array.
- Access to values by indexing the array.
- Values stored in a Tcl_Obj struct
  - Tcl_Obj holds 2 representations, (called the "stork" model)
    - Canonical string
    - Internal representation, such as an integer or a List.
- Access via commands in a script, or C API calls.
- List commands: lindex, lsearch, lsort, lrange, foreach, lset, etc, are tied to this data model.



Abstract List

- An Abstract List
  - Separates the data management from the access operations.
  - Access is via a set of protocol functions.
- How list values are stored or managed depends on the protocol functions.
  - Examples: RB-Tree, Hash Table, Directed Graph, $f(x)$
- Values can even be computed on demand
  - The value for a given index must be consistent with the string representation of an equivalent List.

Abstract List

continued

- The `[lseq]` command is implemented as an Abstract List.
- Generates a sequence of numbers based on a start, end, and step values.

`[lseq 10 .. 15 by 3]` -> `{10 13 16 19 22 25}`

- Values computed using math: *f*(index) = ($start `+ ($index * $step))`
- This allows for very large lists with O(1) create time.
  Just don't ask for a string of the entire list. 😉

Why Abstract Lists

- Optimize
  - Value storage space
  - Value access
  - Computation
- Eliminate the need to mimic List commands
- Reduce or eliminate ”shimmering”
  - In TCL, defined scalar value types have a Length function that always returns 1
  - This avoids the conversion to a List

```
set k [expr {6 + 7}]
set point {3 15}
…
if {[llength $k] > 1} {
    # Process a point
    …
} else {
    set x [expr {$k / 2}]
}
```

New abstract list Tcl_Obj types

Can abstract lists improve existing list operations?

Yes!

New abstract list Tcl_Obj types

- repeatedList — holds repeated elements

  lrepeat *count ?element ...?*

- reversedList — holds elements of an existing list in reverse order

  lreverse *list*

- rangeList — holds elements of a range within an existing list

  lrange *list first last*

Benefits

- Memory efficient
  - Either through sharing memory with the source list or
  - A more efficient internal representation.
  - Allows certain list operations on large lists feasible that were previously impractical.
- Makes `lrepeat`, `lreverse`, and `lrange` very fast
  - orders of magnitude for even moderately long lists
- Possible other benefits for large lists:
  - Minimizing cache use
  - Minimizing swapping
  - Decrease cost of freeing additional memory

Drawbacks

- Indexing and iterating an abstract list can be less efficient
  - roughly 4-10%.
- Arithseries has higher overhead due to on demand memory allocation.
- With these new list types, the efficiency loss is due to not having direct support in the byte code execution engine.
- In all likelihood, this can be eliminated by modifying the BCE if so deemed necessary for acceptance.
- For simplicity, the current implementation does not modify the BCE but instead only uses the abstract list representations for lengths greater than a threshold (100 at the time of writing) where the memory savings is seen as substantial.

lrepeat Performance Comparison

Parameters	Command	Runtime 9.0.0	Runtime apn-tip636-appl
Memory growth		16612 KB	144 KB
--			
(len=10)	`{lrepeat $len x}`	0.131	0.135
(len=99)	`{lrepeat $len x}`	0.323	0.314
(len=101)	`{lrepeat $len x}`	0.322	0.113
(len=1000)	`{lrepeat $len x}`	1.894	0.113
(len=10000)	`{lrepeat $len x}`	18.327	0.113
(len=100000)	`{lrepeat $len x}`	183.535	0.113
(len=1000000)	`{lrepeat $len x}`	2,107.770	0.113

lindex Performance Comparison

Parameters	Command	Runtime	Runtime
		9.0.0	apn-tip636-appl
(len=10)	`{lindex $l $ix}`	0.100	0.099
(len=99)	`{lindex $l $ix}`	0.100	0.100
(len=101)	`{lindex $l $ix}`	0.100	0.109
(len=1000)	`{lindex $l $ix}`	0.099	0.108
(len=10000)	`{lindex $l $ix}`	0.101	0.108
(len=100000)	`{lindex $l $ix}`	0.099	0.109
(len=1000000)	`{lindex $l $ix}`	0.099	0.107

foreach Performance Comparison

Parameters	Command	Runtime	Runtime
		9.0.0	apn-tip636-appl
(len=10)	`{foreach v $l {}}`	1.974	1.935
(len=99)	`{foreach v $l {}}`	16.650	16.401
(len=101)	`{foreach v $l {}}`	17.000	17.705
(len=1000)	`{foreach v $l {}}`	165.182	170.577
(len=10000)	`{foreach v $l {}}`	1,645.150	1,698.710
(len=100000)	`{foreach v $l {}}`	16,556.300	17,017.000
(len=1000000)	`{foreach v $l {}}`	164,621.300	169,603.200

New List API's

Access List commands from C:

lrange	lrepeat	lreverse

`int Tcl_ListObjRange(`
`    Tcl_Interp *interp,`
`    Tcl_Obj    *srcListPtr,`
`    Tcl_Size    first,`
`    Tcl_Size    last,`
`    Tcl_Obj   **newListPtrPtr`
`);`
	`int Tcl_ListObjRepeat(`
	`    Tcl_Interp *interp,`
	`    Tcl_Size    repeatCount,`
	`    Tcl_Size    objc,`
	`    Tcl_Obj    *const objv[],`
	`    Tcl_Obj   **newListPtrPtr`
	`);`
		`int Tcl_ListObjReverse(`
		`    Tcl_Interp *interp,`
		`    Tcl_Obj *srcListPtr,`
		`    Tcl_Obj **newListPtrPtr`
		`);`

Acknowledgement

Thanks to Ashok for implementing and testing these new List optimizations.

Q & A