A random-access pool of similar objects

NEIL A. EDELMAN

2022-03-01

Abstract

A pool is parameterized to one type, offering packed random-access insertion and deletion. Pointers to valid items in the pool are stable, but not generally in any order. When removal is ongoing and uniformly sampled while reaching a steady-state size, it will eventually settle in one contiguous region.

1 MOTIVATION

In many applications, we would like a stream of many similar object's addresses to be stable throughout each of their lifetimes, but we can not tell, *a priori* how many objects at one time will be needed. Dynamic arrays are not suited for this because, in order there to be a contiguity guarantee, the pointers are not guaranteed to be stable. C^{++}'s `std::deqeue` is close, but it only allows deletion from the ends.

The pool, therefore, must not be contiguous, but we want blocks of data to be in one section of memory for fast cached-access and low storage-overhead. This suggests an an array of chunks, each one exponentially bigger than the last full chunk.

To reach the ideal contiguous chunk of memory, we only allocate memory to a chunk from the largest capacity, active chunk_0. When data is deleted from a secondary chunk, it is unused until all the data is gone and we can free that chunk.

2 DETAILS

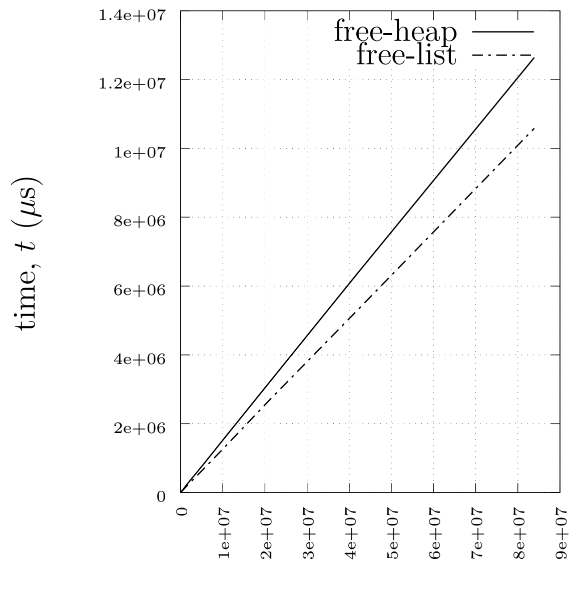
2.1 Marking Entries as Deleted

We face a similar problem as garbage collection: in the active chunk_0, we need some way to tell which are deleted. The first choice is a free-list. When adding an entry, check if the free-list is not empty, and if it is not, we recycle deleted entries by shifting the list. Alternately, popping the free-list works, too, but on average, shifting yields lower, more compact addresses.

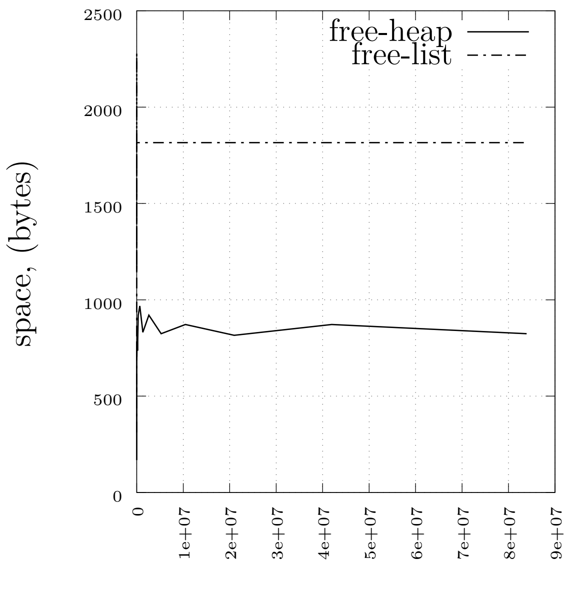
The free-list is $\mathcal{O}(1)$ amortized run-time, but hard $\Theta(\sum_n \text{capacity of chunk}_n)$[1] space requirement. Alternately, we could use an implicit complete binary-tree free-heap. This will negatively affect the run-time, $\mathcal{O}(\log \text{chunk}_0)$, but space requirement is much more reasonable, $\mathcal{O}(\text{chunk}_0)$. A reference count on the secondary chunks.

Figure 1 compares a hypothetical `struct keyval` with an `int` and a string of 12 `char` both in run-time, Figure 1a, and total space, Figure 1b. It was decided that the space requirement of using a free-list is too great for, what turned out to be, a very modest performance gain in this region.

With a free-list, all the items in a block that are non-deleted have a null pointer attached to them. A free-heap saves initializing and reading the list, and only needs the



(a) Run-time.



(b) Space.

Figure 1: Time and space to inserting and deleting n random, but linearly stable about 50 items.

space for what one has deleted. This is not without a downside: there is no guarantee that one will be able to expand the heap to account for a deletion; it could be that deleting an item fails.

2.2 Which chunk are they in?

Except primary chunk_0, we maintain the chunks sorted by memory location. Whenever we allocate a new chunk_0 in response to the old chunk_0 being full, it requires that we binary search the array of chunks and insert the old, $\mathcal{O}(\log \text{chunks}{+}\text{chunks}) = \mathcal{O}(\text{chunks})$. Similarly for deleting a chunk. Because the exponential growth of chunks, this happens $\mathcal{O}(\log \text{items})$. Thus, the worst-case time to delete is, $\mathcal{O}(\text{chunks}{+}\log \text{items in chunk}_0) = \mathcal{O}(\log \text{items})$.

Insertion is amortized $\mathcal{O}(1)$. In the case where chunk_0 is full, we allocate a new chunk for at least $(1+\epsilon)n$ new entries, and each entry bears a transfer of constant time. In the case where the free-heap is empty, just return $\text{chunk}_0[\text{size}_0]$. If there's any addresses in the maximum free-heap, just pop an address from the array of which the heap is made and return it; we know that it is lower in the binary-tree heap than those higher up. We prefer to have low-numbers, but any number would do.

Approximation to golden ratio for array.

$if(r > c.size/(growth = 50.0))$

Figure 2 shows a diagram of a pool, the same type that was tested in Figure **??**.

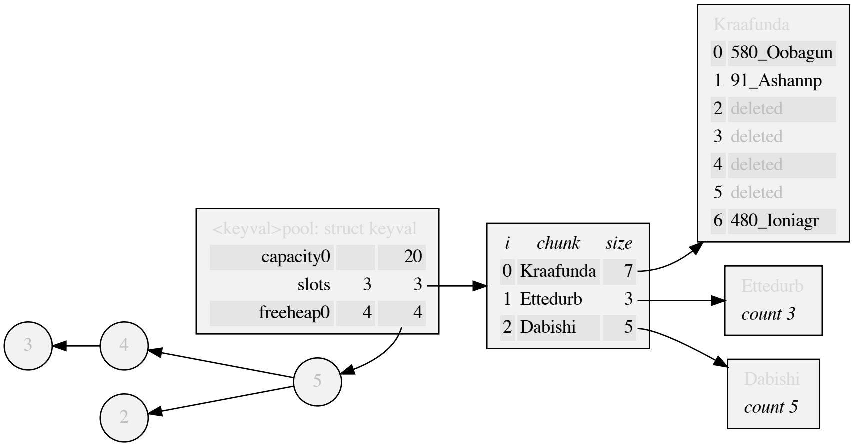


Figure 2: A pool consists of several chunks of packed data managed by an array, and a free-heap for chunk_0.

3 PERFORMANCE

4 IMPLEMENTATION

A max heap[2] with which we can compare the data in the maximum position in chunk_0 on deletion. If it matches, we decrement the max value and pop from the stack until top $<$ $\text{address}_{\text{end}}$. In this way, the end of the list is never deleted. The procedure is amortized. For example, in Figure 2, if we deleted the data at chunk_0, index 6, `480_Ioniagr`, then it would result in successive amortized deletions of all the heap; the *size* would be 2.

5 CONCLUSION

REFERENCES

[1] D. E. Knuth, "Big omicron and big omega and big theta," *SIGACT News*, vol. 8, p. 18–24, apr 1976.

[2] J. Williams, "Heapsort," *CACM*, vol. 7, no. 6, pp. 347–348, 1964.

[3] R. E. Tarjan, "Applications of path compression on balanced trees," *J. ACM*, vol. 26, p. 690–715, oct 1979.