A Garbage Collection System in Memory

Angus Cheung

December 2023

Contents

1 Approach Overview

1.1 Hash Tables

Three two-level hash tables are created. Each hash location contains a binary search tree. The hash tables are the following:

Hash Table Name	**Key**	**Value**
`free_addresses_hash_table`	Address of free blocks	Size of free blocks
`free_sizes_hash_table`	Size of free blocks	List of addresses of free blocks
`allocated_addresses_hash_table`	Address of allocated blocks	Size of allocated blocks

Table 1: Summary of Hash Tables

1.2 Details of Each Hash Table

1. `free_addresses_hash_table`: A hash table indexed by addresses of the free blocks, having the size of each free block as values.
2. `free_sizes_hash_table`: A hash table indexed by sizes of the free blocks, containing a list of all addresses of free blocks that have the same size as values.
3. `allocated_addresses_hash_table`: A hash table indexed by addresses of the allocated blocks, containing the size of each allocated block as values.

Note: The `free_addresses_hash_table` and `allocated_addresses_hash_table` have unique keys (addresses), while the `free_sizes_hash_table` may have non-unique keys (sizes).

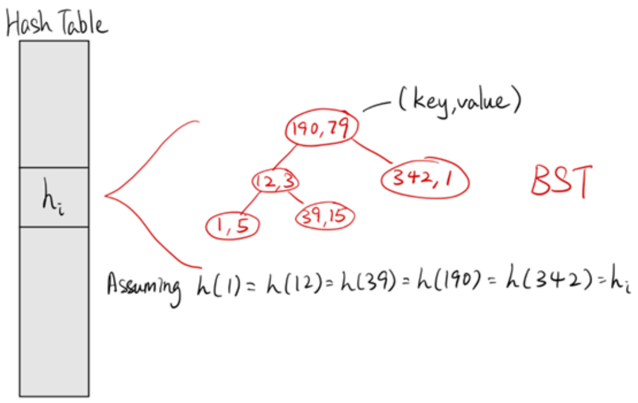

Figure 1: The Structure of `free_addresses_hash_table`

The diagram above shows the structure of `free_addresses_hash_table` as an example.

Hashing by multiplication is used as the hashing method. The following is a general form of the hash function (Mehta & Sahni, 2004):

$$h(x) = \lfloor mxA \rfloor \mod m \tag{1}$$

where A is an arbitrary real-valued constant.

Assuming that the key is b bits, meaning that it ranges from 0 to $2^b - 1$, the following hash function is used for `free_addresses_hash_table` and `allocated_addresses_hash_table`:

$$h(k) = \left\lfloor \frac{Ck}{2^b} \right\rfloor \mod C \tag{2}$$

Here, the capacity of the hash table is C. A good hash function must be universal, meaning that the keys will be distributed uniformly into the buckets. In this case, the buckets (indexed from 0 to $C-1$) will each contain a continuous range of $2^{b-\log_2 C}$ keys. This hash function is universal as long as the addresses are distributed uniformly. This choice of C is discussed in the "Trade-Offs in Data Structure" section.

For `free_sizes_hash_table`, we cannot assume a uniform distribution of block sizes. For every block allocation with starting address a and size b, there is a hard constraint of $a+b \leq M$, where M is the aggregated remaining memory space. When a follows a uniform distribution, $b \leq M-a$ follows a triangular distribution for each allocation request, with a shrinking M as the number of allocated blocks increases. The actual distribution of b would depend on the aggregated size of allocated spaces.

If we instead assume that b follows a $\log_2$-uniform distribution, then we can write:

$$h(k) = \left\lfloor \frac{C \log_2 k}{b} \right\rfloor \mod C \tag{3}$$

All hash tables support the following operations:

- Inserting a key-value pair
- Deleting a key
- Finding the corresponding value given a key input
- Finding the next larger/smaller key and the corresponding value given a key input
- Finding the maximum key and the corresponding value

In addition, `free_sizes_hash_table` supports deleting a key-value pair. For example, if it contains $(3, [1, 5])$ and `delete(3, 5)` is run, the node becomes $(3, [1])$.

Lastly, we maintain a bitmap, an array of zeros and ones, that shows which buckets are non-empty.

1.3 Request Operation

When a request operation (i.e., to allocate a block) is received, the algorithm first checks whether the operation is valid. The most basic criteria are:

- The size and address provided must be within the range from 0 to the total memory size, inclusively.
- If no starting address is provided, it checks the maximum free block size and sees whether it is large enough. The maximum free block size is obtained by calling `max_key()` on `free_sizes_hash_table`.
- If a starting address (`op.addr`) is provided, it checks whether that location is currently within a free block. This is done by calling `next_smaller_key(op.addr)` on `free_addresses_hash_table` to see if the starting address and the ending address of this free block sandwich `op.addr`. If so, it checks whether the free block is large enough to accommodate the allocation.

After the operation is deemed valid, the algorithm proceeds with the allocation:

- If `op.addr` is provided, the block is allocated directly.
- Otherwise, the algorithm must first find the appropriate location to allocate the block. There are three strategies to allocate the block:
    1. **First-fit:** It loops through the buckets in `free_addresses_hash_table` that are non-empty. This is achieved using the bitmap, as mentioned earlier. It finds the first free block large enough and allocates it into the location.

2. **Best-fit:** It calls `query(size)` on `free_sizes_hash_table` to find a free block that is exactly the same size as the allocated block. If it fails to find one, it calls `next_larger_key(size)` on `free_sizes_hash_table` to find the smallest free block larger than the allocated block.
3. **Worst-fit:** It calls `max_key()` on `free_sizes_hash_table` to find the largest free block to fit the allocated block.

The allocation process consists of updating the three hash tables and merging adjacent allocated blocks:

1. The algorithm begins by identifying the free block that will contain the allocated block.
2. The allocated block is inserted into `allocated_addresses_hash_table`.
3. The free block is deleted from `free_addresses_hash_table` and `free_sizes_hash_table`.
4. Extra free spaces at the start and end of the allocated block are likely left. These are inserted back into `free_addresses_hash_table` and `free_sizes_hash_table`.

Afterwards, adjacent allocated blocks are merged, if applicable:

- Adjacent allocated blocks are identified by calling `next_smaller_key` and `next_larger_key` on `allocated_addresses_ha`
- If 1: The end of the previous allocated block is equal to the start of the current allocated block, or 2: The end of the current allocated block is equal to the start of the next allocated block, the algorithm performs merging.
- The algorithm deletes the original blocks from `allocated_addresses_hash_table` and adds the merged block to the hash table.
- If both 1 and 2 co-occur, the algorithm deletes all three original blocks and forms one merged block.

1.4 Release Operation

When a release operation (i.e., to free up an allocated space) is received, the algorithm first checks whether the operation is valid. The most basic criteria are:

- The size and address provided must be within the range from 0 to the total memory size, inclusively.
- It finds the allocated block that exactly starts at the starting address supplied by calling a `query` on `allocated_addresses_hash_table`.
- If unsuccessful, it finds the allocated block that contains the starting address provided by calling `next_smaller_key` on `allocated_addresses_hash_table`.

Next, it determines:

- The end of the allocated block: `allocated_start` + `allocated_size`.
- The end of the space to be freed up: `start` + `size`.

If the end of the freed space exceeds the allocated block, it means that some space to be released is already free in the first place. This makes the operation invalid. Otherwise, the operation is considered valid.

The deallocation process includes updating the three hash tables and merging adjacent free blocks. This process is the exact opposite of the allocation process. The steps are highly similar.

2 Trade-Offs in Data Structure

2.1 Hash Function Selection

The current hash function produces buckets containing a continuous range of keys. A crucial assumption to maintain the universality of the hash function is that the keys are uniformly distributed. If all keys are concentrated in a specific narrow range, then some buckets would have many keys, while others would have none.

For **first-fit**, the hash function does not perform well when the number of blocks is tiny. This is because all addresses are concentrated at the lower end. However, as blocks are continuously released and allocated, a first-fit strategy results in a fragmented memory space. At this time, the addresses become more uniformly distributed; thus, the hash function performs better.

For **best-fit**, the hash function allows easy identification of the smallest free block larger than the required size. Since the hash function guarantees that the buckets contain a continuous range of keys, we can look at the bucket that the required size belongs to in `free_sizes_hash_table` and quickly find the free block slightly larger than the required size. If it is not found, we can look at the next non-empty bucket and get the minimum key inside that bucket.

For **worst-fit**, the hash function allows easy identification of the largest free block. Since the hash function guarantees that the buckets contain a continuous range of keys, we can swiftly find the largest block in `free_sizes_hash_table` by looking at the non-empty bucket with the highest index. This assumes that there is a considerable variation in free block sizes.

A series of 100,000 random request and release operations in a memory map of size 2^{32} is generated. The first 10 operations are requests, and the following operations are randomized uniformly between requests and releases. For requests:

- There is a uniform probability of having no address provided or having an address provided.
- The address is generated by a uniform distribution in $[0, 2^{32} - 1]$.
- The size is generated with a $\log_2$-uniform distribution in $[0, 32]$.

For releases, a previous request is randomly picked and reverted. Note that most of the operations are erroneous due to the random generation process.

Using **best-fit**, the following bucket size distribution is observed (from left to right: `free_addresses_hash_table`, `free_sizes_hash_table`, `allocated_addresses_hash_table`):

- Number of elements in the three hash tables: 2426, 1207, 2427.

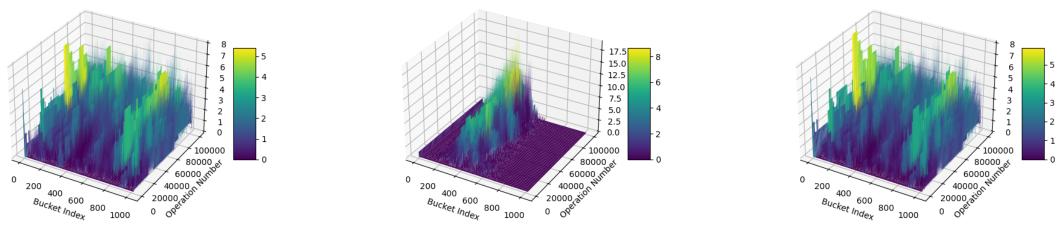

Figure 2: Best-fit bucket size distributions for the three hash tables.

Using **first-fit**, the following bucket size distribution is observed:

- Number of elements in the three hash tables: 4251, 3515, 4251.

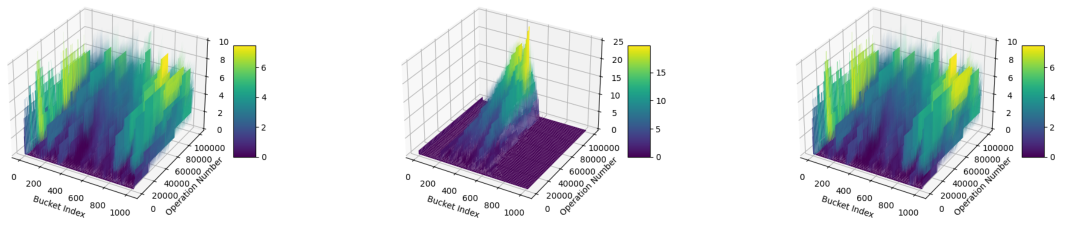

Figure 3: First-fit bucket size distributions for the three hash tables.

Using **worst-fit**, the following bucket size distribution is observed:

- Number of elements in the three hash tables: 4787, 4752, 4787.

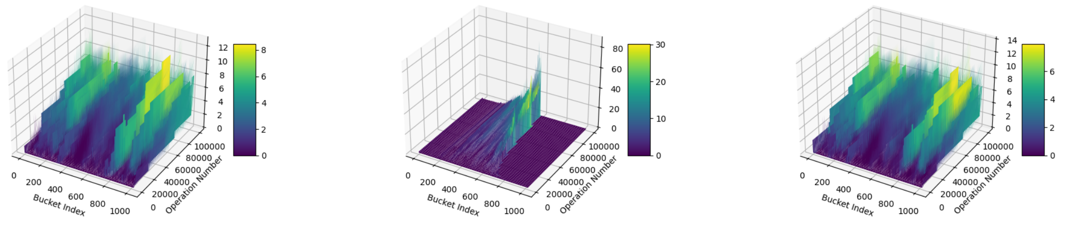

Figure 4: Worst-fit bucket size distributions for the three hash tables.

As observed above, the bucket sizes of `free_addresses_hash_table` and `allocated_addresses_hash_table` are uniformly distributed. However, that of `free_sizes_hash_table` seems divergent. Most elements are concentrated in the middle portion of the buckets. This is a compromise of the hash function selection: by selecting a hash function such that the ranges of keys amongst the buckets are continuous, and having an undeterminable distribution of free block sizes, we inevitably end up with a concentration of keys in some buckets.

However, this way of hashing allows us to implement specific functionalities more efficiently. This will be discussed in the "Using Binary Search Tree as a Second-Level Data Structure" section.

2.2 Hash Table Capacity Selection

The value of C is dynamically adjusted based on the size of the memory map. When the memory map is x bits in size, C is defined as:

$$C = 2^{\lfloor \frac{x}{3} \rfloor}$$

For example:

- If the memory map is 1024 in size, the capacity of the hash table becomes 8.
- If the memory map is 32 bits in size, the capacity of the hash table becomes 1024.

If the capacity is too large, it would waste space and leave many buckets unused. Conversely, if the capacity is too small, there would be a lot of entries in a single bucket. With the above definition of C, it is assumed that the expected number of entries n scales proportionally with the size of the memory map.

The **Load Factor** L is defined as:

$$L = \frac{n}{C}$$

Ideally, we want L to be around 0.75. Taking this assumption:

- n is expected to be 6 when the memory map is 1024 in size. This means we expect 6 allocated blocks and 6 free blocks to coexist at a given moment.
- Similarly, n is expected to be 768 when the memory map is 32 bits. This means we expect 768 allocated blocks and 768 free blocks to coexist at a given moment.

2.3 Using Binary Search Tree as a Second-Level Data Structure

The benefit of using a **Binary Search Tree (BST)** as a secondary structure, combined with the fact that the buckets in the hash table contain a continuous range of keys, is that we can efficiently implement the following operations, as mentioned in the "Approach Overview" section:

1. Finding the next larger key and the corresponding value given a key input.
2. Finding the next smaller key and the corresponding value given a key input.
3. Finding the maximum key and the corresponding value.

In the following, let n represent the number of nodes in the BST.

Operation (1): Finding the Next Larger Key
To implement this operation:

1. First, we identify the bucket that the key belongs to. This step takes $O(1)$ time.
2. Next, we conduct a **successor search** in the BST of that bucket. If the successor is not found, we move to the next non-empty bucket and find the minimum in the BST.

Since both the successor search and finding the minimum take $O(\log n)$ time, the whole operation takes $O(\log n)$ time.

Operation (2): Finding the Next Smaller Key
To implement this operation:

1. First, we identify the bucket that the key belongs to. This step takes $O(1)$ time.
2. Next, we conduct a **predecessor search** in the BST of that bucket. If the predecessor is not found, we move to the previous non-empty bucket and find the maximum in the BST.

Since both the predecessor search and finding the maximum take $O(\log n)$ time, the whole operation takes $O(\log n)$ time.

Operation (3): Finding the Maximum Key
To implement this operation:

1. First, we go to the non-empty bucket with the largest index. Using the bitmap, this step takes $O(1)$ time.
2. Next, we find the maximum in the BST of that bucket.

The whole operation takes $O(\log n)$ time.

Efficiency of Hash Table with Secondary BSTs
Compared to a single BST with no hashing, we are effectively dividing the keys into C BSTs and using a hash

function to determine which BST(s) to investigate. This approach makes a hash table with secondary BSTs more efficient for implementing the three operations described above.

Limitations of Using a Secondary Hash Table
Using a secondary hash table instead of a secondary BST within each bucket is not optimal because:

- Hash tables do not necessarily preserve the order of keys.
- Hash tables do not intrinsically support predecessor and successor searches.

However, the trade-off is that we increase query time from $O(1)$ to $O(\log n)$. Despite this, the secondary BST structure allows us to efficiently handle ordered operations like finding the next larger or smaller key.

3 Performance Summary

3.1 Data Structures

The hash table has a capacity of:

$$2^{\left\lfloor \frac{\log_2 M}{3} \right\rfloor},$$

as previously defined, where M is the total size of the memory map. The space complexity for the hash locations is:

$$O\left(M^{\frac{1}{3}}\right).$$

If there are N keys to be stored, then on average, each BST occupies a space of:

$$O\left(\frac{N}{M^{\frac{1}{3}}}\right).$$

Hence, the overall space complexity of the hash table is:

$$O\left(\frac{N}{M^{\frac{1}{3}}}\right) \times O\left(M^{\frac{1}{3}}\right) = O(N).$$

Query Cost
For each probe into the hash table:

- The query cost is $O(1)$ for finding the hash location.
- The query cost is $O(\log n)$ for finding the key within the hash location, where n is the number of keys within the hash location.

Assuming universality, we have:

$$n = O\left(\frac{N}{M^{\frac{1}{3}}}\right).$$

Therefore, the query cost is:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

If $M \gg N$ (there are many more possible memory addresses than the number of allocated blocks), as M increases, the query cost approaches $O(1)$. This means that as the memory map grows, the performance of the two-level hash table with a secondary BST approaches the performance of a two-level hash table with a secondary hash table, as long as $M \gg N$.

For example:

- When $M = 2^{32}$ and $N = 2^{10} = 1024$, it suffices to say $M \gg N$.

Insertion and Deletion Cost
For insertions and deletions:

- The cost is $O(1)$ for finding the hash location.
- The cost is $O(\log n)$ for inserting or deleting a key from the BST.

Using similar logic as above, the cost is:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

For $M \gg N$, the cost approaches $O(1)$.

Operations Within the BST
Within a BST:

- Successor search, predecessor search, minimum key search, and maximum key search all take $O(\log n)$.

This is common knowledge mentioned in lectures and will not be detailed here.

3.2 Request Operation

Validating the Operation

Checking whether the operation is valid involves:

- Calling `query` and `next_smaller_key` on `free_addresses_hash_table`, or
- Calling `max_key` on `free_sizes_hash_table`.

`query` has already been proven to take:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

`next_smaller_key` involves a failed predecessor search in one bucket and a maximum key search in another in the worst case. The total cost is:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right) \times 2 = O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Meanwhile, `max_key` involves a maximum key search in one bucket. Thus, the total cost is also:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Block Finding Strategies

If an address is not provided, the algorithm searches for a block using one of the following strategies:

First-Fit
First-fit involves searching for the first available free block. In the worst-case scenario, this could take $O(N)$, as we may need to go through every bucket and key only to find that the last bucket contains the desired location.

Best-Fit
Best-fit involves:

- A `query`, and
- A `next_larger_key` operation on `free_sizes_hash_table`.

`next_larger_key` involves a failed successor search in one bucket and a minimum key search in another in the worst case. The total cost is:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right) \times 2 = O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Thus, the entire best-fit strategy takes:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Worst-Fit
Worst-fit involves a `max_key` operation on `free_sizes_hash_table`. This takes:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Allocation Process

The allocation process consists of a constant number of insertion and deletion operations on the hash tables. The cost is:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Merging Allocated Blocks

The merging of allocated blocks also consists of a constant number of insertion and deletion operations on the hash tables. The cost for this operation is:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Total Costs

- When an address is provided, or when a best-fit or worst-fit strategy is used, the total cost is:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

- When a first-fit strategy is used, the algorithm could take up to:

$$O(N),$$

  in the worst case.

3.3 Release Operation

The deallocation process and the merger of free blocks both take:

$$O\left(\log\left(\frac{N}{M^{\frac{1}{3}}}\right)\right).$$

Both processes involve a constant number of insertion, deletion, and other function calls on the hash tables, as discussed above.

4 Potential Improvements

To enhance the current memory management system, implementing a dynamic hash table that adjusts its size and rehashes its contents based on the existing load could significantly improve space complexity and operational efficiency. This adaptability would ensure that the load factor remains within an optimal range, such as 0.75, as mentioned in this report, thus minimizing space waste and potentially improving cache utilization.

The rehashing process itself may be expensive. Therefore, it is important to define a suitable rule for a rehashing trigger such that the **amortized cost** is $O(1)$.

For example, once the hash table reaches a particular load factor, it doubles in size, and all existing elements must be rehashed and inserted into the new, larger table. This rehashing process has a time complexity of:

$$O(n),$$

where n is the number of elements in the hash table at the time of resizing. However, the **amortized cost** for each element is:

$$O(1).$$