An explanation of Doug Lea's memory manager

John Wickerson

Abstract

We describe the basic functionality of Doug Lea's memory manager.

1 The routines

There are four main public routines:

- `malloc(`n`)` returns a pointer to a newly-allocated chunk with payload at least n bytes, or `null` if no space is available. It uses the internal `chunk_alloc` routine.
- `free(`p`)` releases the chunk identified by p, and has no effect if p is `null`. It uses the internal `chunk_free` routine.
- `realloc(`p`,` n`)` returns a pointer to a chunk with payload at least n bytes, that contains the first n bytes of data in p (or all of it if it has fewer than n bytes of payload), or `null` if no space is available. The pointer returned may or may not be the same as p. We do not plan to verify this routine.
- `calloc` is the same as `malloc`, except the payload of the returned chunk is fully zeroed. We do not plan to verify this routine.

2 The arena and boundary tags

The arena is shown in Fig. 1. The "blocks" of the Unix V7 memory manager are now called "chunks". The layout of these chunks evolves over time as they are allocated, freed, split and coalesced. The procedures ensure that no free chunk ever borders another free chunk – these would always be coalesced. Boundary tags (as first described by Knuth in 1973) are used to simply this coalescing: each chunk stores its own size and the size of its predecessor (in the `head` and `prev_foot` fields respectively), which effectively makes the arena into a doubly-linked list. Put concretely: given a pointer `p` to a chunk, the addresses of the previous and next chunks can be computed as follows:

```
prev_chunk(p) = (mchunk *)(((char *)p) - (p -> prev_foot))
next_chunk(p) = (mchunk *)(((char *)p) + ((p -> head) & ~111_b))
```

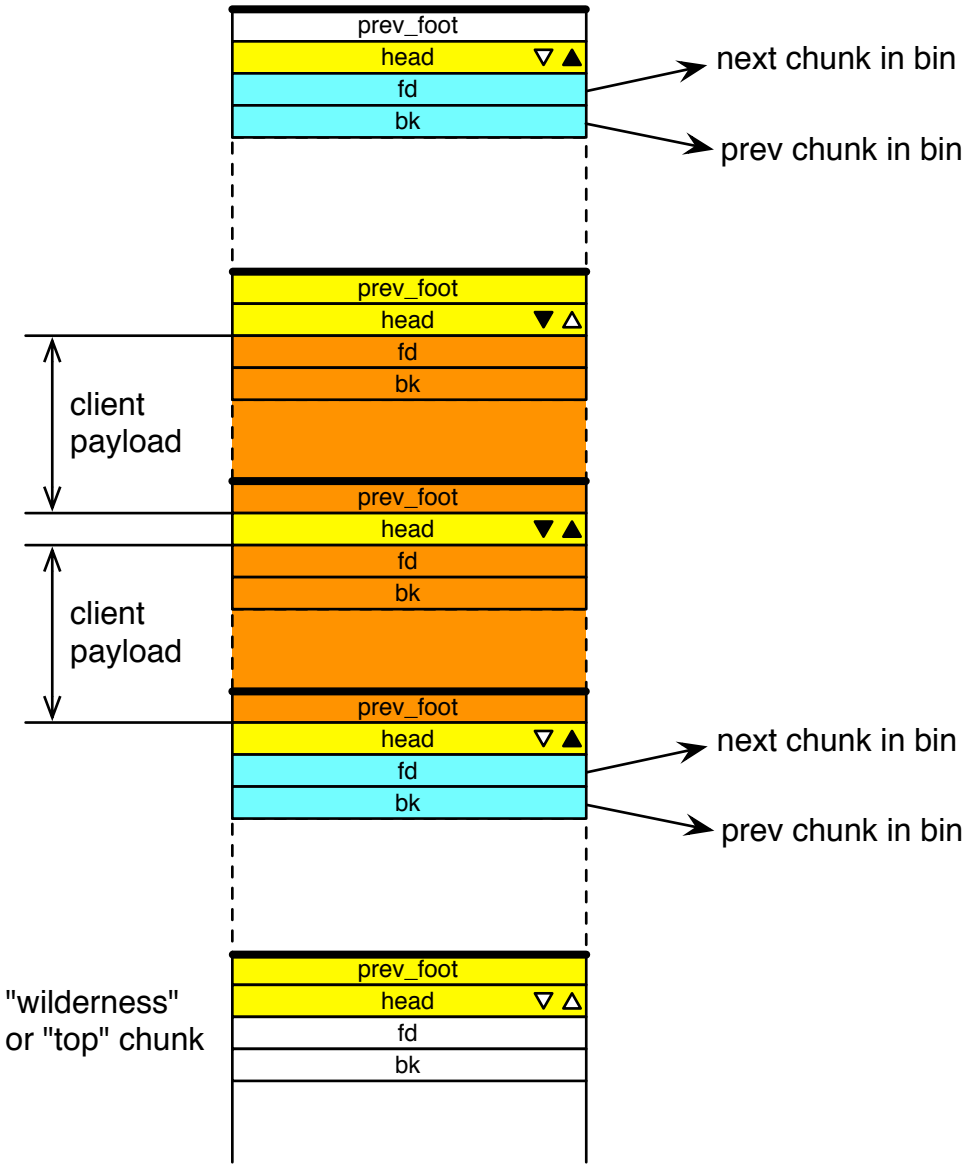

Figure 1: The arena

CINUSE	PINUSE	interpretation
0	1	previous chunk is in use, current block is free
1	0	previous chunk is free, current block is in use
1	1	both previous and current blocks are in use
0	0	current block is memory mapped

Table 1: The flags in each chunk's `head` field

The divisions between chunks are highlighted with thick black lines. See how the payload of an in-use chunk actually overlaps the `prev_foot` field of the next chunk. This is fine: the next chunk doesn't need to able to reach the in-use chunk until the in-use chunk becomes free. Upon becoming free, the `prev_foot` field can be recalculated.

2.1 Flags

Chunks are 8-byte aligned, so the three lowest order bits of the `head` field are redundant. The two lowest are employed as flags; the third-lowest is "not used by this malloc, but might be useful in extensions". It is impossible for both the previous and the current blocks to be free (they would have already been coalesced), so the fourth configuration in Table 1 is used to signify that the block is memory mapped (see §3.3). In Fig. 1, the downwards-pointing triangle refers to the current chunk, while the upwards-pointing triangle refers to the previous chunk. A filled triangle denotes a set bit.

Remark. Interestingly, this interpretation was changed in the 2009 release. Previously, the lowest bit stored whether or not the previous chunk was in use, and the second-lowest stored whether or not the current chunk was memory mapped. This meant that to determine whether the current chunk was in use, one had to look at the `prev_foot` field of the *next* chunk!

Note that `PINUSE` is permanently set for the first chunk in the arena, so as to avoid attempts to coalesce with non-existant memory.

2.2 The top chunk

The last chunk in the arena is called the top chunk, or the wilderness chunk. It is always free, and its size is defined to be bigger than any other chunk, since it can almost always be made so via a call to `sbrk`. This way, it is only used when no other chunk is big enough, and we thus avoid unnecessary `sbrk` calls. Page 39 of the source code comments that "the actual size of [the] topmost space is `topsize`+`TOP_FOOT_SIZE`, which includes space reserved for adding fenceposts and segment records if necessary when getting more space from the system."

2.3 Bin pointers

The `fd` and `bk` fields locate the chunk within a second datastructure called a bin, which is essentially a circular linked-list comprising chunks of the same or a similar size. The fields are overwritten when the chunk is allocated, and hence are only used by free chunks. Because each chunk has these four 4-byte fields, the minimum chunk size is 16 bytes – that is, `malloc(0)` allocates a chunk of 16 bytes.

3 Binning

Free chunks are maintained in bins, which are circular doubly-linked lists of chunks of the same or similar size. Free chunks whose size is a multiple of 8 bytes up to 248 bytes are stored in one of 32 "small" bins that link identically-sized chunks. Chunks of 256 bytes or more are stored in one of 32 "tree" bins.

Initially, all of the bins are empty, because at the start the whole heap is composed of one single chunk, the wilderness chunk, which is never included in any bin.

3.1 Small bins

A chunk of size n bytes is stored in the bin at array index $\lfloor n/8 \rfloor$. Since the smallest chunk size is 16 bytes, the first two bins seem to be redundant[check this]. There is something a little funny here: the variable `NSMALLBINS` is set to 32, but the array `smallbins` has $(\texttt{NSMALLBINS} + 1) \times 2 = 66$ elements.

3.2 Tree bins

There are 32 more bins, which have ranges that increase approximately logarithmically. The maximum chunk size that can be accommodated is about 2 GB. Chunks are sorted by size within each bin. All free chunks are binned, except the remainder of the most recently split chunk (called the designated victim) and the top chunk.

Suppose a chunk has size n bytes. The `compute_tree_index` macro is responsible for calculating the appropriate index I into the array of tree bins. If n is less than 256 bytes, then I is 0 (although in such a case, the `compute_tree_index` macro wouldn't be used: the chunk would be in a small bin). If n is greater than FFFFFF_h then I is the last index in the array: `NTREEBINS` $- 1$[1]. In general, I is defined as follows:

$$I = \begin{cases} 0 & \text{if } n < 2^8 \\ \texttt{NTREEBINS} - 1 & \text{if } n \geq 2^{24} \\ 2(\lfloor \log_2 n \rfloor - 8) & \text{if } \{\!\{n\}\!\} < \lfloor\!\lfloor n \rfloor\!\rfloor / 2 \\ 2(\lfloor \log_2 n \rfloor - 8) + 1 & \text{if } \{\!\{n\}\!\} \geq \lfloor\!\lfloor n \rfloor\!\rfloor / 2 \end{cases}$$

[1] Not sure why the upper limit is FFFFFF.

where the $\lfloor\!\lfloor - \rfloor\!\rfloor$ operator rounds its input down to the nearest power of two, that is: $\lfloor\!\lfloor n \rfloor\!\rfloor = 2^{\lfloor \log_2 n \rfloor}$. We also define $\{\!\{n\}\!\} = n - \lfloor\!\lfloor n \rfloor\!\rfloor$, notationally recalling the $\{-\}$ operator from arithmetic, which returns the fractional part of a real number. Note that for all $n > 0$, $\{\!\{n\}\!\}$ must always lie between 0 (inclusive) and $\lfloor\!\lfloor n \rfloor\!\rfloor$ (exclusive). The last two clauses in the definition above pivot upon the median of this range, thus yielding two equally-spaced bins for each power of two. The offset 8 is required because the tree bins only begin at chunk sizes of 2^8 bytes. To clarify, the ranges of each tree bin are as follows. Note that all numbers are written in hexadecimal.

Size n in hex (bytes)	Bin index
$100 \le n < 180$	0
$180 \le n < 200$	1
$200 \le n < 300$	2
$300 \le n < 400$	3
$400 \le n < 600$	4
$600 \le n < 800$	5
$\vdots$	$\vdots$

3.3 Memory mapping

Page 34 of the source code explains that the `prev_foot` field of a memory-mapped chunk is used to hold the chunk's offset within its memory mapped region. This is needed to preserve alignment. Each memory-mapped chunk is trailed by the first two fields of a fake next chunk, for the sake of usage checks.

4 Outline of `dlmalloc`

The algorithm runs roughly as follows:

```
void* dlmalloc(size_t bytes) {
  if bytes ≤ MAX_SMALL_REQUEST
    if a smallbin of the right size is non-empty
      return a chunk from that smallbin
    else if designated victim is not big enough
      if there exists a non-empty sufficiently-large smallbin
        return (first part of) chunk from smallest such smallbin
      else if there exists a non-empty treebin
        if allocation in a treebin succeeds, return the resulting chunk
  else if bytes ≥ MAX_REQUEST, return fail
  else // MAX_SMALL_REQUEST < bytes < MAX_REQUEST
    if allocation in a treebin succeeds, return the resulting chunk
  // either the designated victim is big enough,
  // or all bins are empty, or treebin allocation failed
  if designated victim is big enough
```

```
            return (first part of) designated victim
        else if top chunk is big enough
18          return (first part of) top chunk
        try to obtain the memory from the system
    }
```

The above presentation is, however, quite hard to follow because of the combination of nested conditionals with multiple return points. A clearer presentation is given by the flowchart in Fig. 2.

5 Description of the state

First define:

$$\begin{array}{rcll} p.\blacktriangledown & \Leftrightarrow & p.\texttt{head} \gg 1 \,\&\, 1 & \\ p.\triangledown & \Leftrightarrow & \neg(p.\blacktriangledown) & \\ p.\blacktriangle & \Leftrightarrow & p.\texttt{head} \,\&\, 1 & \\ p.\triangle & = & \neg(p.\blacktriangle) & \\ p.CP & = & p.C \wedge p.P & \forall C \in \{\blacktriangledown, \triangledown\}, P \in \{\blacktriangle, \triangle\} \\ p.size & = & p.\texttt{head} \,\&\, \tilde{}3 & \\ p \to q & = & p + p.size = q & \\ p \leftarrow q & = & b.\triangledown \Rightarrow (q - q.\texttt{prev\_foot} = p) & \\ \leftrightarrow & = & \to \cap \leftarrow & \\ \leftrightarrow^*, \leftarrow^*, \to^* & = & \text{reflexive transitive closures of } \leftrightarrow, \leftarrow, \to & \end{array}$$

The internal state of the memory manager comprises the arena plus several state state variables.

$$\text{Overall state} = \boxed{mstate * arena}$$

Where *mstate* comprises:

- `start`: an auxiliary variable added in for the purposes of the proof. Identifies the first chunk in the arena.
- `dv`, `dvsize`: `dv` identifies an arena chunk, and `dvsize` its size. Define
$$dvOK \Leftrightarrow (\texttt{dv} = 0 \wedge \texttt{dvsize} = 0) \vee (\texttt{start} \to^* \texttt{dv} \wedge \texttt{dv}.size = \texttt{dvsize}).$$
- `top`, `topsize`: `top` identifies the topmost arena chunk, and `topsize` its size. See the remark in §2.2 about `topsize`. Define
$$topOK \Leftrightarrow \texttt{start} \to^* \texttt{top} \wedge \texttt{top}.size = \texttt{topsize}.$$
- `smallbins`: an array of smallbins. Note that the array has size 66, yet it holds only 32 smallbins. Define
$$smallbin(i) = \texttt{smallbin}[2i]$$
$$smallbinsOK \Leftrightarrow \forall i \in [0, 32).\, smallbinOK(smallbin(i))$$
$$smallbinOK(b) \Leftrightarrow \text{“check that all chunks are free, in the arena, and the right size”}.$$

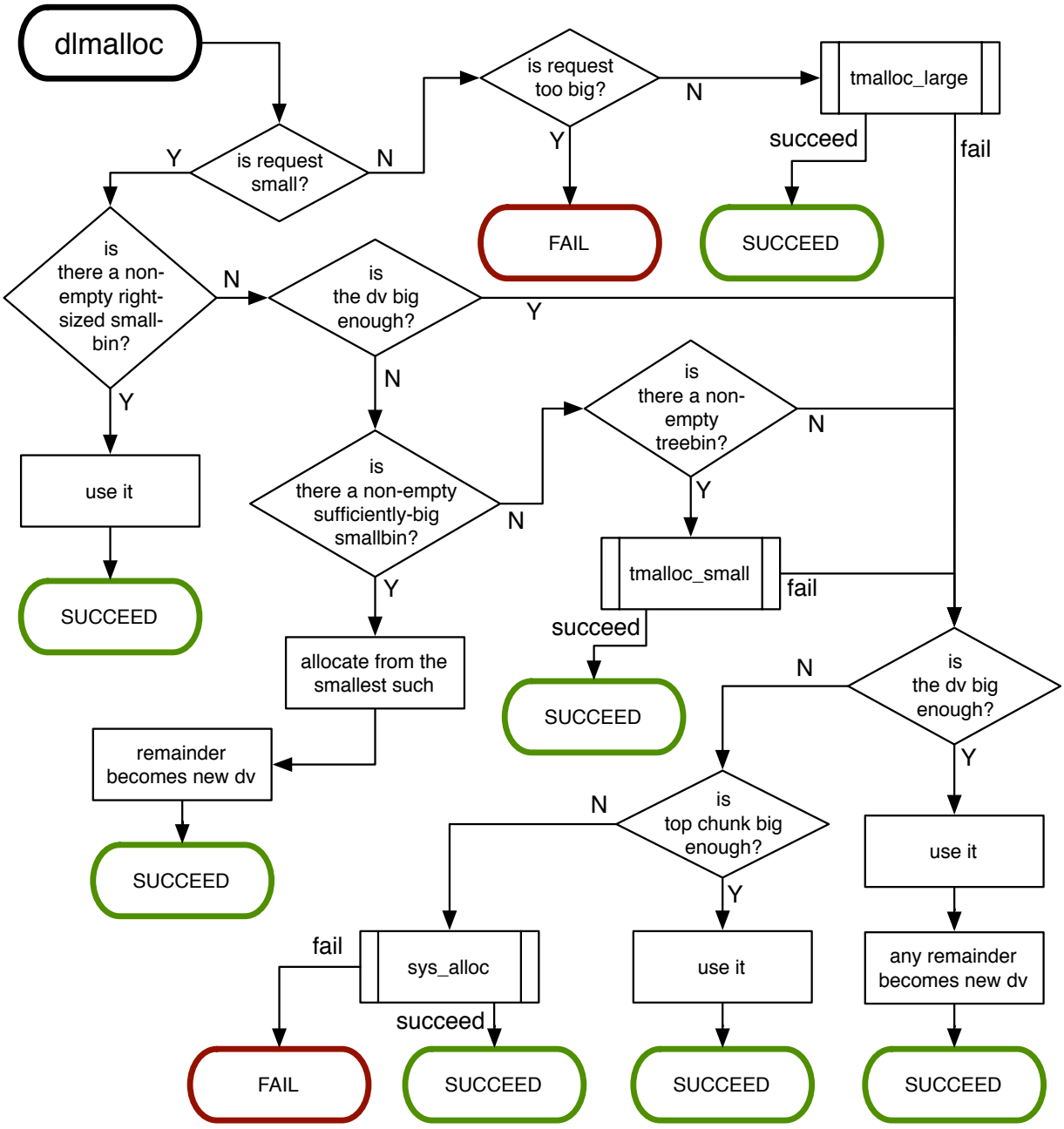

Figure 2: Flowchart for `dlmalloc`

- `treebins`: an array of 32 pointers to treebins. Define

$$treebinsOK \Leftrightarrow \forall i \in [0, 32).\, treebinOK(\texttt{treebins}[i])$$
$$treebinOK(b) \Leftrightarrow \text{“check that all chunks are free, in the arena, and the right size”.}$$

- `smallmap`, `treemap`: bit vectors that store whether each smallbin and treebin is empty or not. Define

$$smallmapOK \Leftrightarrow \forall i \in [0, 32).\, \texttt{smallmap}[i] = 0 \Leftrightarrow smallbin(i) = \varnothing$$
$$treemapOK \Leftrightarrow \forall i \in [0, 32).\, \texttt{treemap}[i] = 0 \Leftrightarrow treebin(i) = \varnothing$$

- `least_addr`: the least address ever obtained from `sbrk` or `mmap`. In contrast, `start` is the least address ever obtained from `sbrk` (I think).

So we can define *mstate* like so:

$$\begin{aligned} mstate = \;& dvOK * topOK * smallbinsOK * treebinsOK \\ & * smallmapOK * treemapOK * \texttt{least\_addr} \mapsto 5 \end{aligned}$$

The *arena* predicate can be defined like so:

$$\texttt{start} \leftrightarrow^* \texttt{top}$$

This states that the arena is valid. The validity of the superimposed bin structure comes from the *smallbinsOK* and *treebinsOK* predicates. As for how to link a particular chunk to both its tree node and its arena node: this remains unclear for now.

6 Outline of `dlfree`

For a rough overview, see the flowchart in Fig. 3.

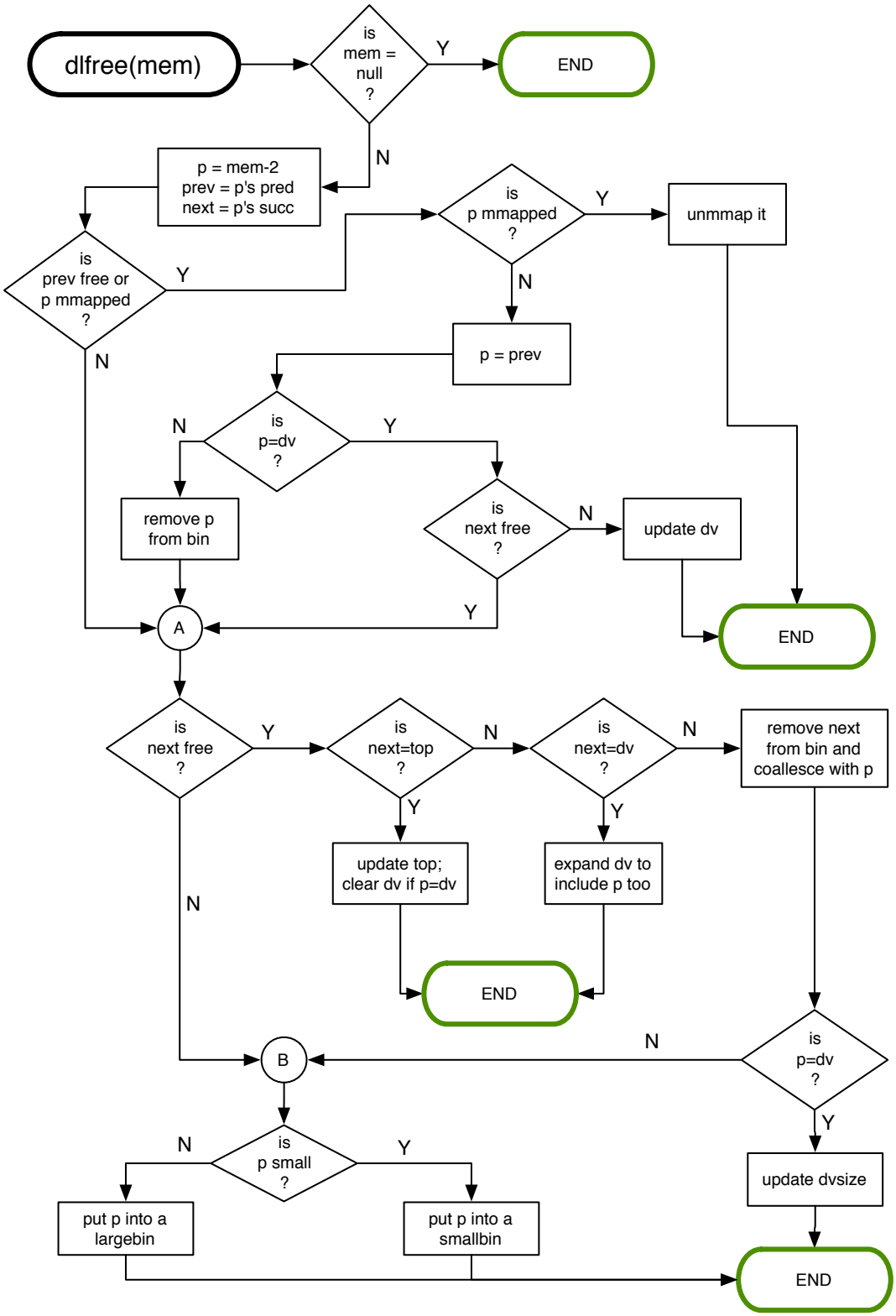

Figure 3: Flowchart for `dlfree`