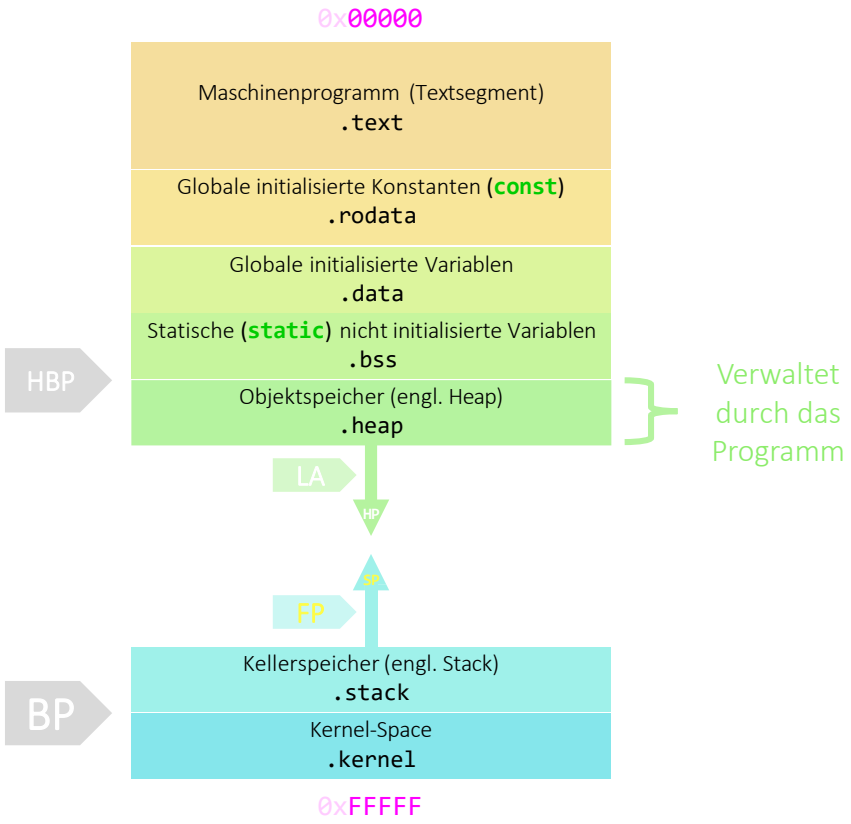
Implementierung dynamischer Speicherverwaltung

Speicherlayout eines Prozesses



HBP ≡ heap
LA ≡ last_alloc_chunk_idx
SETHP(...) : HP anpassen
HPCHK(...) : HP überprüfen
ADDRERR() im Fehlerfall

Dynamische Speicherverwaltung in C

- Funktion zur **Allokation** von Speicher: `malloc(size_t size)`
- Reserviert zur Laufzeit Speicher im Heap
- `size` legt die Größe des zu reservierenden Bereichs in Bytes fest
- Rückgabewerte:
  - `≠ NULL`: wenn Allokation erfolgreich
  - `= NULL`: wenn ein Fehler aufgetreten ist

Speicherplatzierungsstrategie 'Next Fit'

- Sucht nächste passende Lücke im Speicher
- Merkt sich Position der letzten Allokation, setzt Suche bei nächster Anfrage dort fort
- Schnelle Platzierungsstrategie die sich besonders für sehr kleine Blöcke gut eignet

Allokation

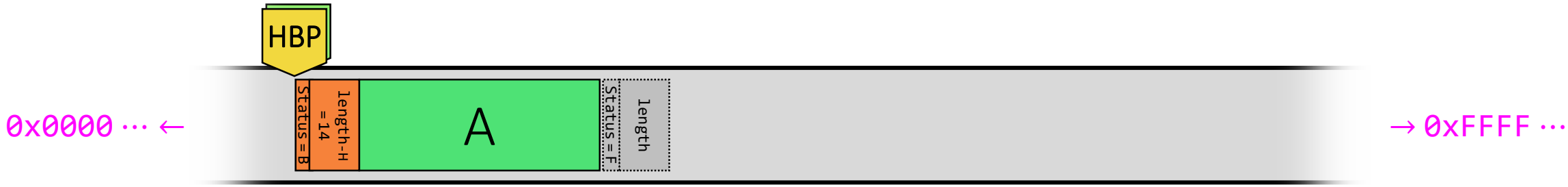
- Metadaten pro Block
  - Status des Blocks: **frei** oder **belegt**
  - Länge des Blocks: falls **0**, ist der Status irrelevant
- Anfänglich können Daten hintereinander in den Speicher gelegt werden

F ≡ CHUNK_FREE ≔ 0 : frei
B ≡ CHUNK_ALLOCATED ≔ 1 : belegt

Ausgangszustand

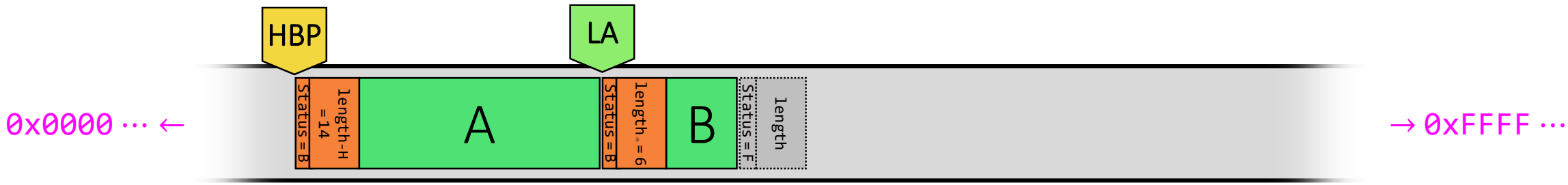
H entspricht hier der Länge der Metadaten.

Allokation eines Blockes A der Länge 14



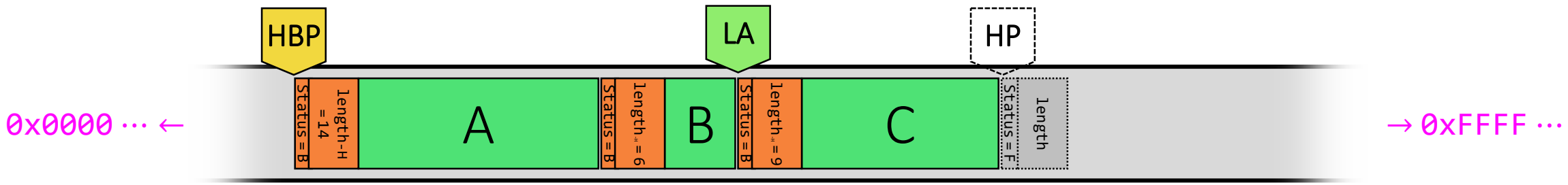
H entspricht hier der Länge der Metadaten.

Allokation eines Blockes B der Länge 6



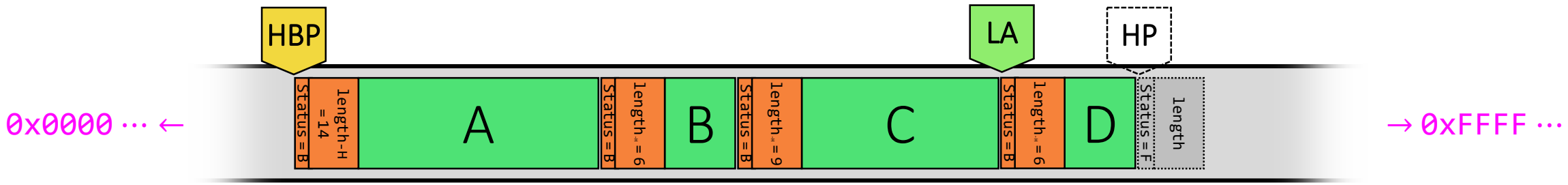
H entspricht hier der Länge der Metadaten.

Allokation eines Blockes C der Länge 9



H entspricht hier der Länge der Metadaten.

Allokation eines Blockes D der Länge 6

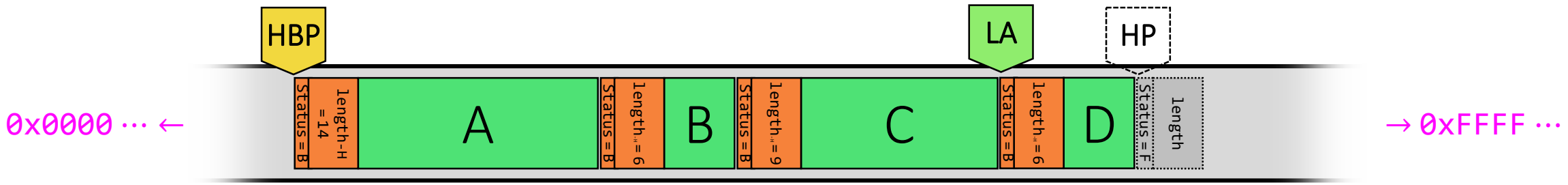


H entspricht hier der Länge der Metadaten.

In-Place Reallokation

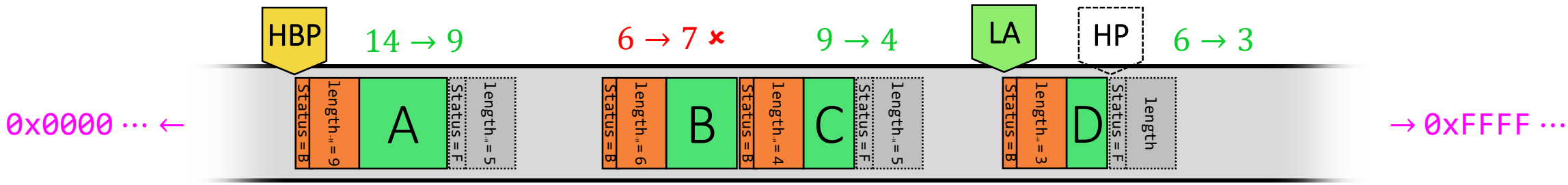
- Blöcke können **in-place realloziert** werden, indem der Status entsprechend angepasst wird
- Vergrößern ist so nur möglich, wenn hinter dem Block eine entsprechend große Lücke frei ist

Blöcke vor einer in-place Reallokation



H entspricht hier der Länge der Metadaten.

Blöcke nach der in-place Reallokation

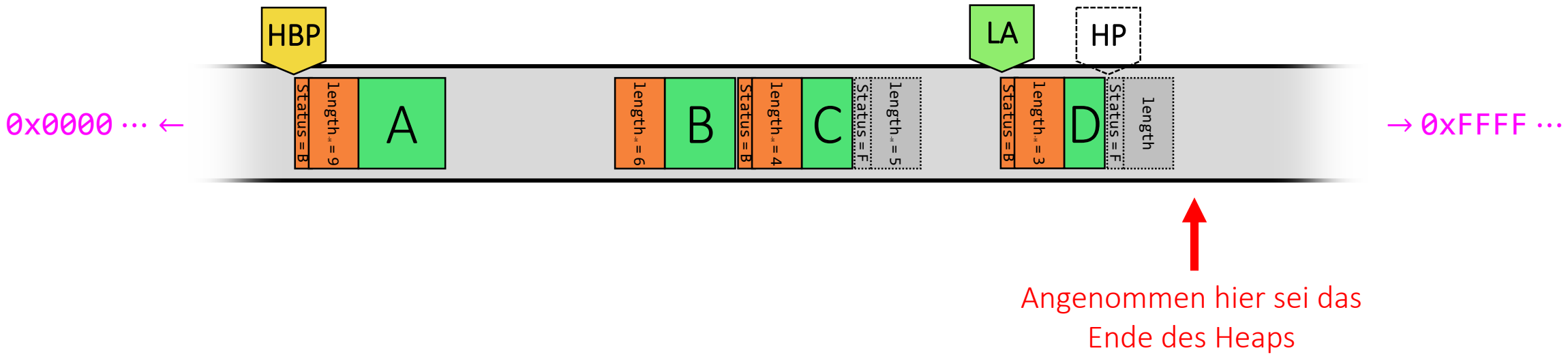


H entspricht hier der Länge der Metadaten.

Suche nach entsprechend großer Lücke

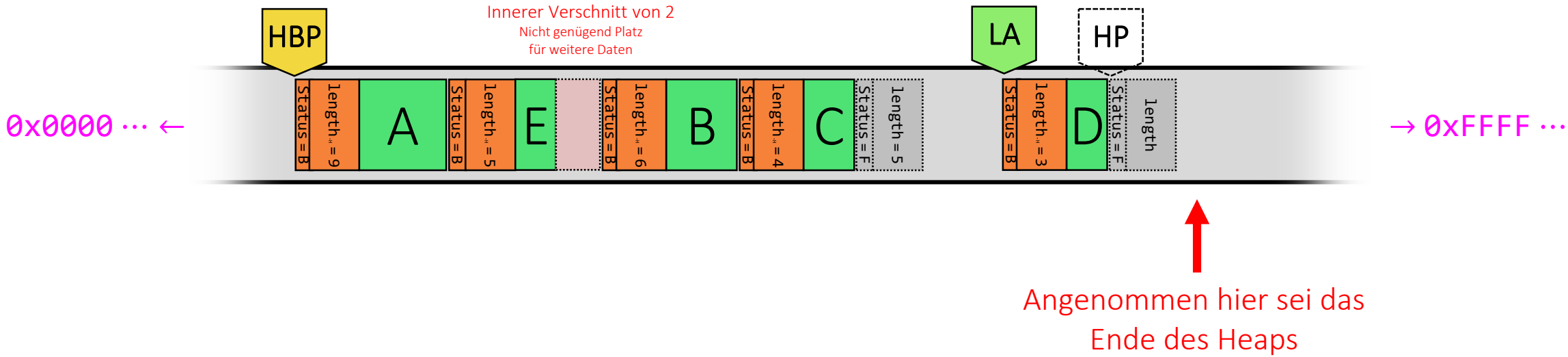
- Suche nach einem freien Bereich
  - Falls Bereich reserviert, um Anzahl Blöcke weiter springen
  - Wenn Ende des Speichers erreicht, dann Vorne beginnen

Allokation eines Blockes $\mathbf{E}$ der Länge $\mathbf{3}$



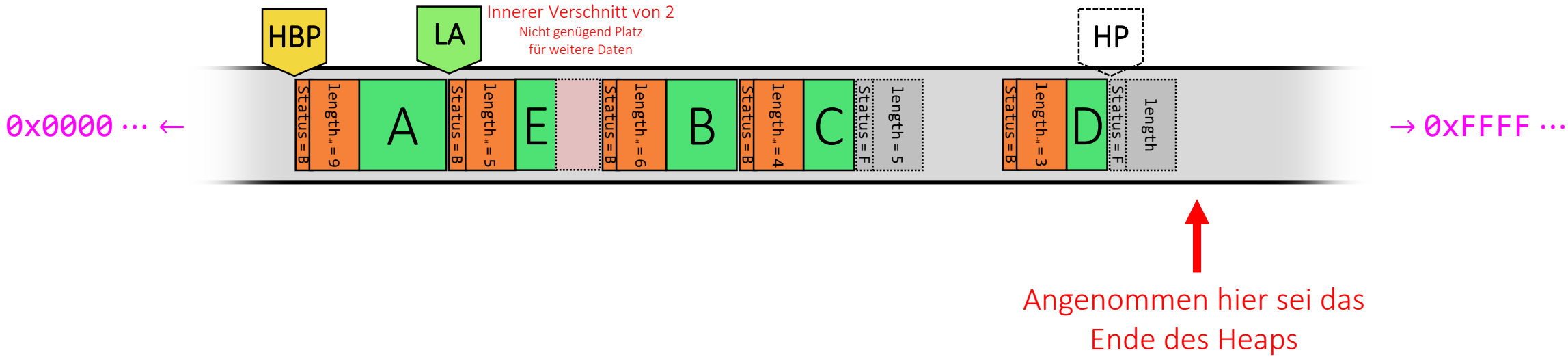
H entspricht hier der Länge der Metadaten.

Allokation eines Blockes E der Länge 3



H entspricht hier der Länge der Metadaten.

Allokation eines Blockes E der Länge 3

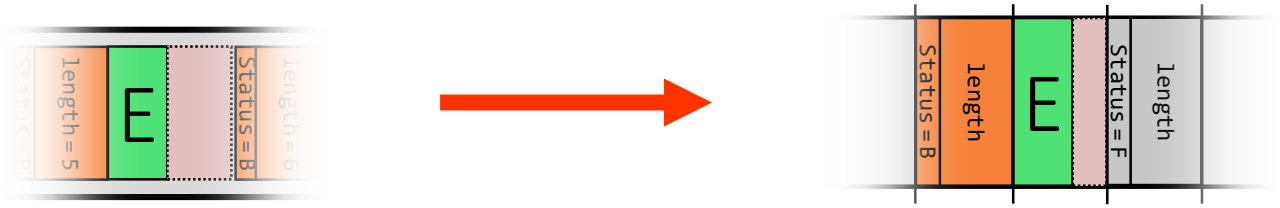


H entspricht hier der Länge der Metadaten.

Chunking

- Heap wird in sehr kleine, gleichgroße Blöcke (**Chunks**) geteilt
- Bei der Reservierung von Speicherbereichen wird die geforderte Speichergröße in Chunks umgerechnet und aufgerundet

  → Ggf. innerer Verschnitt
- Chunkgröße ≥ Metadaten ⇒ jede freie Lücke im Speicher kann auch als solche entsprechend markiert werden: In der Implementierung spart man sich so eine Fallunterscheidung!



Suche nach der nächsten passenden Lücke I

```
static size_t find_gap(size_t req_chunks_with_header) { size_t i;

    // Search from last_alloc_chunk_idx
    for (i = last_alloc_chunk_idx; i < NUM_CHUNKS; ) {
        mem_info* current_info = (mem_info*)(heap + i * CHUNK_SIZE);
        if (current_info->length == 0) { // Invalid or already coalesced block
            // This should not happen if freeing is done correctly with coalescing.
            return NUM_CHUNKS; // Error case
        }

        if (current_info->status == CHUNK_ALLOCATED) {
            i += current_info->length;
        } else if (current_info->length < req_chunks_with_header) {
            i += current_info->length;
        } else {
            return i; // Suitable free block found
        }
    }
    ⋮
}
```

Suche nach der nächsten passenden Lücke II

```
static size_t find_gap(size_t req_chunks_with_header) { size_t i;
    ⋮
    // Search from the beginning if no suitable block was found at the end of the heap
    for (i = 0; i < last_alloc_chunk_idx; ) {
        mem_info* current_info = (mem_info*)(heap + i * CHUNK_SIZE);
        if (current_info->length == 0) { // Invalid or already coalesced block
            // This should not happen if freeing is done correctly with coalescing.
            return NUM_CHUNKS; // Error case
        }

        if (current_info->status == CHUNK_ALLOCATED) {
            i += current_info->length;
        } else if (current_info->length < req_chunks_with_header) {
            i += current_info->length;
        } else {
            return i; // Suitable free block found
        }
    }
    return NUM_CHUNKS; // No suitable block found
}
```

Implementierung von 'Next-Fit' I

```
void* nf_alloc(size_t size) {
    if (size == 0) { return NULL; }

    // req_chunks_with_header is the requested size
    // in chunks PLUS space for the header
    size_t req_chunks_with_header = size_to_chunks(size);
    size_t chunk_index = find_gap(req_chunks_with_header);

    mem_info* allocated_block_info = (mem_info*)(heap
                                     + chunk_index * CHUNK_SIZE);
    size_t old_total_length = allocated_block_info->length;

    if (chunk_index >= NUM_CHUNKS) { return NULL; }

    ⋮
}
```

Implementierung von 'Next-Fit' II

```
void* nf_alloc(size_t size) {
    ⋮
    // Splitting logic
    if (old_total_length > req_chunks_with_header) {
        // Calculate the starting index of the remaining free block
        size_t remaining_chunk_idx = chunk_index + req_chunks_with_header;
        mem_info* remaining_block_info = (mem_info*)(heap
                                          + remaining_chunk_idx * CHUNK_SIZE);

        remaining_block_info->status = CHUNK_FREE;
        remaining_block_info->length = old_total_length - req_chunks_with_header;
    }

    ⋮
}
```

Implementierung von 'Next-Fit' III

```
void* nf_alloc(size_t size) {
    ⋮
    // Set status
    allocated_block_info->status = CHUNK_ALLOCATED;
    allocated_block_info->length = req_chunks_with_header;

    // Update last_alloc_chunk_idx for Next Fit
    last_alloc_chunk_idx = chunk_index;
    SETHP(chunk_index + req_chunks_with_header); HPCHK();

    // Return pointer to the user data area (after the header)
    return (void*)(heap + chunk_index * CHUNK_SIZE + sizeof(mem_info));
}
```

Speicherplatzierungsstrategie 'Best Fit'

- Sucht die am besten passende Lücke im Speicher
- Aufwendigere Platzierungsstrategie die aber für größere Blöcke sehr gute Ergebnisse liefert

Speicherplatzierungsstrategie 'Best Fit'

- $\mathrm{D}_n(x) \coloneqq \|x\| - n > 0 \Rightarrow$ Äußerer Verschnitt
- Suche $\arg\min_{\langle \Omega_n, \leq \rangle} \mathrm{D}_n$:

```
while (current_chunk_idx < NUM_CHUNKS) {
    mem_info* current_info = (mem_info*)(…);

    if ( Keine passende freie Lücke ) {
        Zum nächsten Block fortbewegen
    } else {   → Passende freie Lücke gefunden
        if ( Betrachtete Lücke ist besser als bisheriges Best-Fit ) {
            bestfit = current_chunk_idx;
        }
        Zum nächsten Block fortbewegen
    }
}
```

$n - \mathrm{H} \equiv$ `req_chunks_with_header`, Ω_n sei die Menge der Lücken, die eine Länge größer oder gleich n besitzen.

Suche nach der besten passenden Lücke

```
static size_t find_bestgap(size_t req_chunks_with_header) {
    size_t bestfit = NUM_CHUNKS;
    size_t current_chunk_idx = 0;

    while (current_chunk_idx < NUM_CHUNKS) {
        mem_info* current_info = (mem_info*)(heap + current_chunk_idx * CHUNK_SIZE);
        if (current_info->length == 0) { current_chunk_idx++; continue; }

        if (current_info->status == CHUNK_ALLOCATED) {
            current_chunk_idx += current_info->length;
        } else if (current_info->length < req_chunks_with_header) {
            current_chunk_idx += current_info->length;
        } else { // Valid free block found that is large enough

            if (bestfit == NUM_CHUNKS
                || current_info->length < ((mem_info*)(heap + bestfit * CHUNK_SIZE))->length) {
                bestfit = current_chunk_idx;            // If it's the first suitable block found,
            }                                           // or smaller than the current bestfit
            current_chunk_idx += current_info->length; // Move to the next block
        }
    }
    return bestfit; // Returns NUM_CHUNKS if no suitable block found
}
```

Implementierung von 'Best-Fit' I

```
void* bf_alloc(size_t size) {
    if (size == 0) { return NULL; }

    // req_chunks_with_header is the requested size
    // in chunks PLUS space for the header
    size_t req_chunks_with_header = size_to_chunks(size);
    size_t chunk_index = find_bestgap(req_chunks_with_header);

    mem_info* allocated_block_info = (mem_info*)(heap
                                     + chunk_index * CHUNK_SIZE);
    size_t old_total_length = allocated_block_info->length;

    if (chunk_index >= NUM_CHUNKS) { return NULL; }

    ⋮
}
```

Implementierung von 'Best-Fit' II

```
void* bf_alloc(size_t size) {
    ⋮
    // Splitting logic
    if (old_total_length > req_chunks_with_header) {
        // Calculate the starting index of the remaining free block
        size_t remaining_chunk_idx = chunk_index + req_chunks_with_header;
        mem_info* remaining_block_info = (mem_info*)(heap
                                          + remaining_chunk_idx * CHUNK_SIZE);

        remaining_block_info->status = CHUNK_FREE;
        remaining_block_info->length = old_total_length - req_chunks_with_header;
    }

    ⋮
}
```

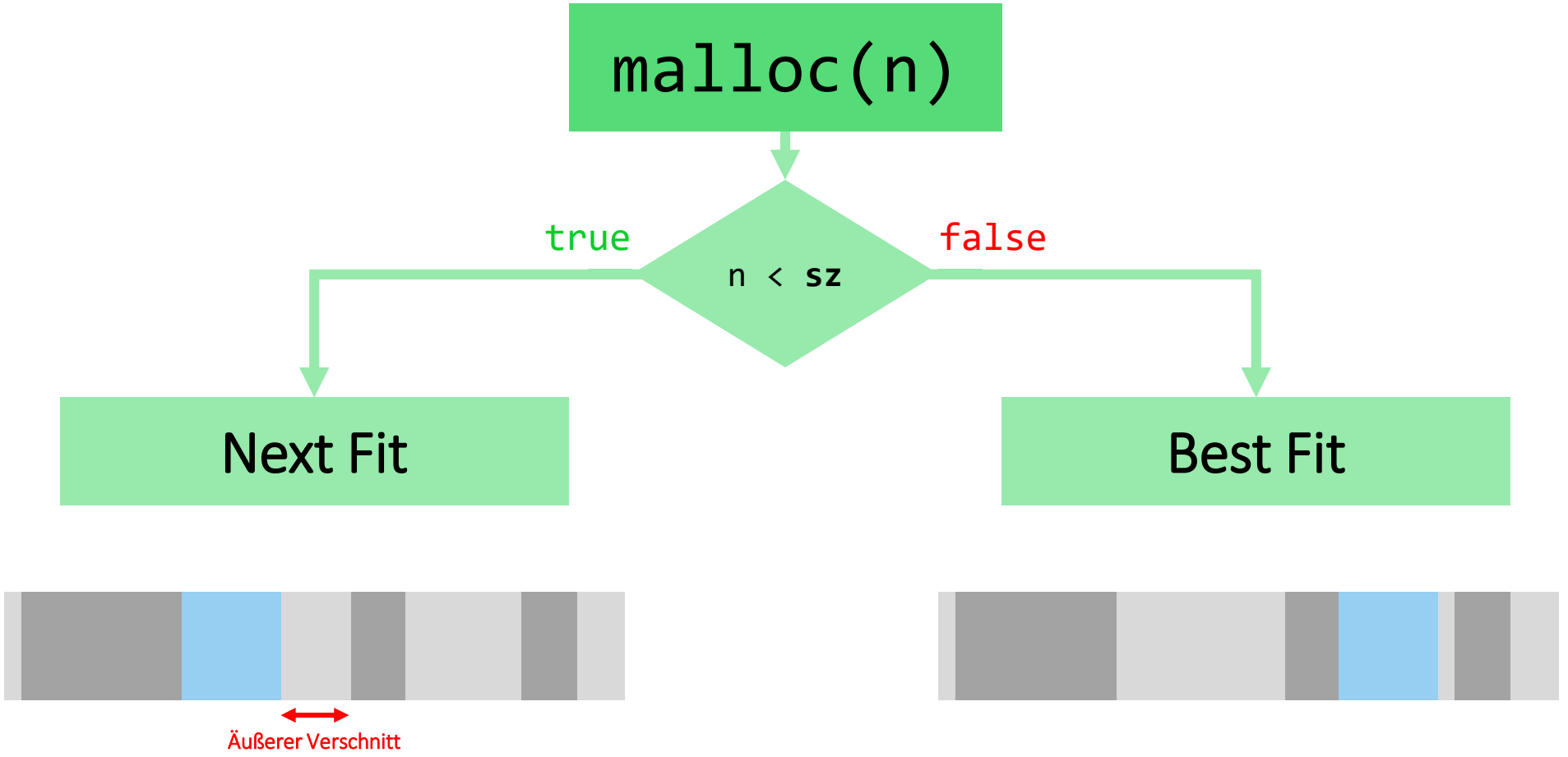
Implementierung von 'Best-Fit' III

```
void* bf_alloc(size_t size) {
    ⋮
    // Set status
    allocated_block_info->status = CHUNK_ALLOCATED;
    allocated_block_info->length = req_chunks_with_header;

    // Update last_alloc_chunk_idx for Next Fit
    last_alloc_chunk_idx = chunk_index;
    SETHP(chunk_index + req_chunks_with_header); HPCHK();

    // Return pointer to the user data area (after the header)
    return (void*)(heap + chunk_index * CHUNK_SIZE + sizeof(mem_info));
}
```

Wahl der geigneten Platzierungsstrategie



SZ ≡ MALLOC_SZ

Implementierung von `malloc`

```
void* mymalloc(size_t size) {
    if (size == 0) return NULL;

    if (size >= MALLOC_SZ) {
        return bf_alloc(size);
    } else {
        return nf_alloc(size);
    }
}
```

Gießen (Casting) von Zeigern

- Wie schreibt man den Integer-Wert `0x12345678` in einen Bereich der Größe 4 mit `char* x = malloc(4)`?
- Der Zeiger `x` wird **gegossen** (**gecastet**):
  - `x` ist ein Byte-Zeiger auf ein `char`
  - `(int*)(x)` castet `x` zu einen Zeiger auf ein `int`
  - Wie gewohnt dereferenzieren:
- `*((int*)(x)) = 0x12345678`

Array im Heap

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int n = 200;

int main(void) {
    int* ptr;
    ptr = malloc(n * sizeof(*ptr));
    if (ptr == NULL) { perror("malloc"); exit(125); }
    for (int i = 0; i < n; ++i) { ptr[i] = i * i; }
    printf("10*10 = %d\n", ptr[10]);
    return 0;
}
```

Initialisierung mit Null

- `void* calloc(size_t` n`, size_t` m`)`
  - Reserviert für n Objekte der Größe m im Heap $n \cdot m$-Bytes und initialisiert diesen mit 0
  - Bei `malloc` ist dies allgemein nicht der Fall:
    Reservierter Speicher kann möglicherweise noch Altlasten vom Stack oder vom Heap beinhalten!

Implementierung von `calloc`

```
void* mycalloc(size_t nmemb, size_t size) {
    size_t total_size = nmemb * size;
    void* ptr = mymalloc(total_size);
    if (ptr) { memset(ptr, 0, total_size); }
    return ptr;
}
```

Array im Heap mit `calloc`

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int n = 200;

int main(void) {
    int* ptr = (int*)(calloc(n, sizeof(*int)));
    if (ptr == NULL) { perror("calloc"); exit(125); }
    for (int i = 0; i < n; ++i) { ptr[i] = i * i; }
    printf("10*10 = %d\n", ptr[10]);
    return 0;
}
```

Speicherplatz freigeben

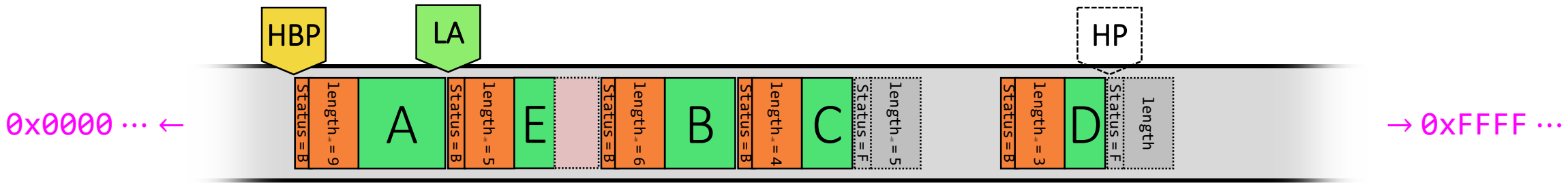
- Gegenstück zu `malloc` (bzw. `calloc`) ist `free(void* ptr)`
- Gibt Speicher an Adresse `ptr` wieder frei
- Speicher darf nur einmal freigegeben werden
- `free` bedarf keine Fehlerbehandlung

Freigabe

Vorgehen bei der Freigabe:

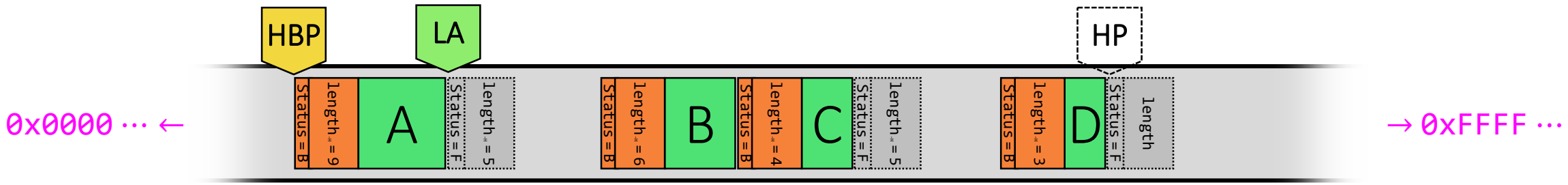
- Markiere Block als frei
- Angrenzende Bereich überprüfen
  - Bereich davor frei? → Verschmelzen
  - Bereich dahinter frei? → Verschmelzen

Vor der Freigabe von E und C



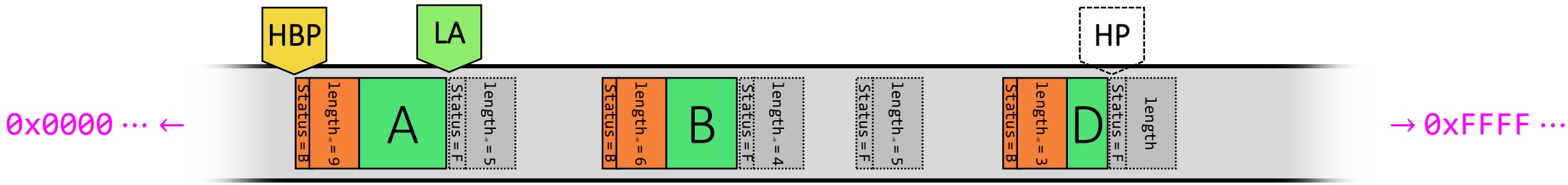
H entspricht hier der Länge der Metadaten.

Nach der Freigabe von E



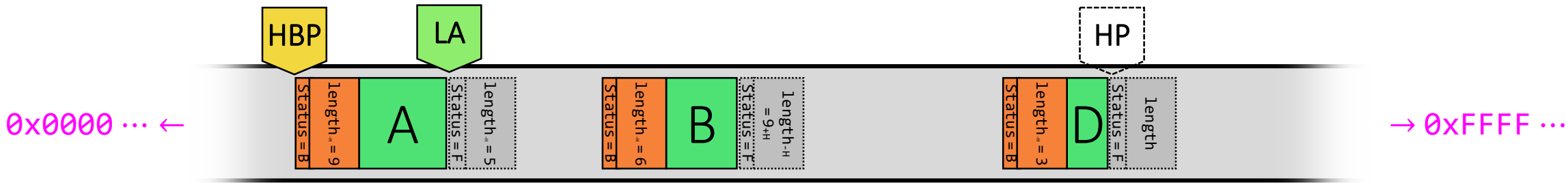
H entspricht hier der Länge der Metadaten.

Nach der Freigabe von C



H entspricht hier der Länge der Metadaten.

Verschmelzen



H entspricht hier der Länge der Metadaten.

Implementierung von `free`

```
void* myfree(void* addr) {
    if (addr == NULL) { return; }

    size_t chunk_index = ((char*)addr - heap - sizeof(mem_info)) / CHUNK_SIZE;

    if (chunk_index >= NUM_CHUNKS
    || (char*)addr  < heap + sizeof(mem_info)
    || (char*)addr  >= heap + CHUNK_SIZE * NUM_CHUNKS) { ADDRERR(); }

    mem_info* current_info = (mem_info*)(heap + chunk_index * CHUNK_SIZE);
    mem_info* next_info = NULL;
    mem_info* prev_info = NULL;

    if (current_info->status == CHUNK_FREE || current_info->length == 0) { ADDRERR(); }

    size_t old_length = current_info->length;
    current_info->status = CHUNK_FREE;

    ⋮
}
```

Implementierung von **free** (Coalescing ←)

```
void* myfree(void* addr) {
    ⋮
    if ((chunk_index + old_length < NUM_CHUNKS)) {
        next_info = (mem_info*)(heap + (chunk_index + old_length) * CHUNK_SIZE);
        // Check if the next block is valid and free
        if (next_info->length > 0 && next_info->status == CHUNK_FREE) {
            current_info->length += next_info->length;
            next_info->length = 0; // Mark the merged block as absorbed
            next_info->status = 0; // Reset status as well

            if (last_alloc_chunk_idx == (chunk_index + old_length)) {
                last_alloc_chunk_idx = chunk_index;
            }
        }
    }
    ⋮
}
```

Implementierung von **free** (Coalescing →)

```
void* myfree(void* addr) {
    ⋮
    if (chunk_index > 0) {
        size_t prev_chunk_idx = 0; prev_info = (mem_info*)heap;

        while (prev_chunk_idx + prev_info->length < chunk_index) {
            prev_chunk_idx += prev_info->length;
            prev_info = (mem_info*)(heap + prev_chunk_idx * CHUNK_SIZE);
            if (prev_info->length == 0) { CONSISTERR(); }
        }

        SETHP(prev_chunk_idx);

        if (prev_info->status == CHUNK_FREE) { // Preceding block is also free -> coalesce
            prev_info->length += current_info->length;
            current_info->length = 0; // Mark the current block as merged
            current_info->status = 0; // Reset status as well

            if (last_alloc_chunk_idx == chunk_index) { last_alloc_chunk_idx = prev_chunk_idx; }
        }
    }
}
```

Verwendung von `free`

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int n = 200;

int main(void) {
    int* ptr;
    ptr = malloc(n * sizeof(*ptr));
    if (ptr == NULL) { perror("malloc"); exit(125); }
    for (int i = 0; i < n; ++i) { ptr[i] = i * i; }
    printf("10*10 = %d\n", ptr[10]);
    free(ptr);
      ⋮
    return 0;
}
```

Beliebige Reallokation

- Blöcke können **in-place realloziert** werden, indem der Status entsprechend angepasst wird
- Vergrößern ist so nur möglich, wenn hinter dem Block eine entsprechend große Lücke frei ist
- Ansonsten müsste ein weiterer Block mit der geforderten Größe reserviert werden, in den der Inhalt dann hineinkopiert wird
  → Den Originalblock würde man danach freigeben

Reallokation

Aktualisieren des Zeigers

- `ptr = realloc(ptr, n)` ändert die Größe des Blocks `*ptr`
- Gelingt `realloc`, besitzt der Block `*ptr` die Größe n
- Schlägt `realloc` fehl, wird `NULL` zurückgeliefert

- `realloc(ptr, 0)` entspricht einem `free(ptr)`
- `realloc(NULL, n)` entspricht einem `malloc(n)`

Implementierung von `realloc` I

```
void* myrealloc(void* ptr, size_t size) {
    if (ptr == NULL) { return mymalloc(size); }
    if (size == 0) { myfree(ptr); return NULL; }

    mem_info* current_info = (mem_info*)((char*)ptr - sizeof(mem_info));
    mem_info* next_info = NULL;
    mem_info* new_next = NULL;
    size_t current_chunks = current_info->length;
    size_t required_chunks = size_to_chunks(size);

    if (required_chunks <= current_chunks) { return ptr; }

    size_t chunk_index = ((char*)ptr - heap - sizeof(mem_info)) / CHUNK_SIZE;
    size_t next_chunk_index = chunk_index + current_chunks;

    ⋮
}
```

Implementierung von `realloc` II

```
void* myrealloc(void* ptr, size_t size) {
    ⋮
    if (next_chunk_index < NUM_CHUNKS) {
        next_info = (mem_info*)(heap + next_chunk_index * CHUNK_SIZE);
        if (next_info->status == CHUNK_FREE && next_info->length > 0
         && current_chunks + next_info->length >= required_chunks) { // Merge with next block
            size_t extra = required_chunks - current_chunks;
            if (next_info->length > extra) { // Take over part of it, the rest remains free
                new_next = (mem_info*)(heap + (next_chunk_index + extra) * CHUNK_SIZE);
                new_next->status = CHUNK_FREE;
                new_next->length = next_info->length - extra;
            }
            current_info->length = required_chunks;
            current_info->status = CHUNK_ALLOCATED;
            return ptr;
        }
    }
    ⋮
}
```

Implementierung von `realloc` III

```
void* myrealloc(void* ptr, size_t size) {
    ⋮
    // New, bigger block required
    void* newptr = malloc(size);
    if (!newptr) return NULL;

    // Cut and paste old data
    size_t copy_size = (current_chunks * CHUNK_SIZE) - sizeof(mem_info);
    if (copy_size > size) copy_size = size;
    memcpy(newptr, ptr, copy_size);

    myfree(ptr);
    return newptr;
}
```