Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 23

MEMORY ALLOCATION TECHNIQUES

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

This lecture covers a range of memory allocation techniques in some detail. In particular, the techniques of pointer increment and sequential fits using a free list are discussed

After introducing segregated free lists and double-linked free lists, the lecture explores coalescing and boundary tags in depth

The lecture ends with a memory management example of how to administer a heap with a sequence of calls to *malloc* and *free*

CONTENTS

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

We start by assuming from now on that the program will request and use **variable-size** heap blocks. Dividing the heap up into variable-sized blocks supports much more efficient use of memory with far fewer pointers linking between blocks. However, the management of the heap becomes much more difficult because we need to track where blocks start and end

VARIABLE-SIZE BLOCKS

- From now on assume variable-size heap blocks
- Much more efficient use of memory
- Far fewer pointers between blocks
- But more difficult to manage – must track block sizes (where they start and end)

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

POINTER INCREMENT

- The simplest and fastest possible allocation mechanism
- A pointer TOP indicates the lowest free memory location in the heap
    - All memory locations above TOP are free.
- A pointer END indicates the first memory location after the end of the heap
- Assume every **byte** in the heap has a separate memory address:
- Possible strategies to follow:
    1. fast allocation, no memory re-use
    2. slower allocation, with compaction

Pointer increment is the simplest and fastest possible allocation mechanism, but as we shall see it is not ideal as an overall dynamic memory management strategy

A "pointer-increment" allocator may for example use:

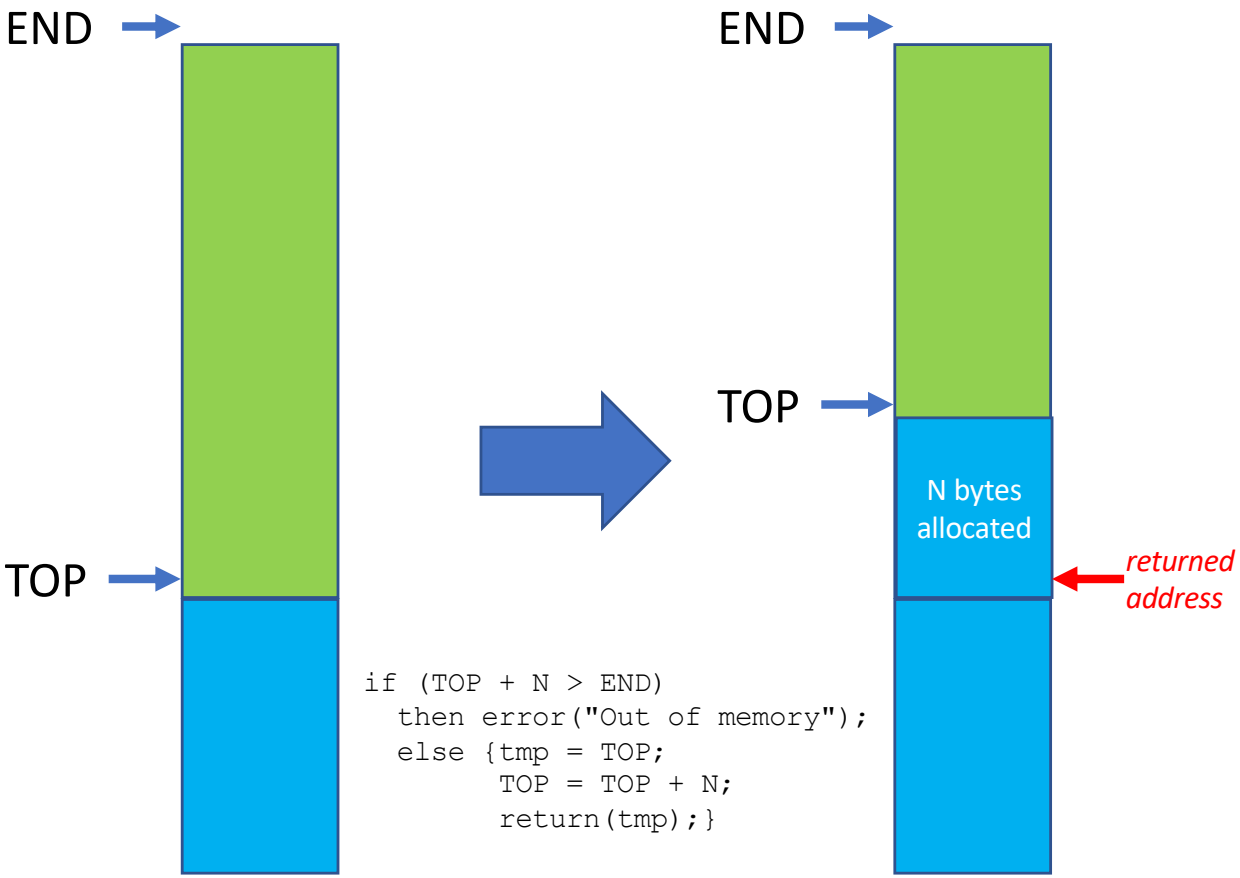
- A pointer (called "TOP") to the lowest free memory location in the heap. An invariant of pointer increment allocation is that all memory locations above TOP are free.
- A pointer (let's call it "END") to the first memory location after the end of the heap.

We will present two example strategies for pointer increment allocation. In both strategies we assume that every **byte** in the heap has a separate memory address:

- Strategy 1 will provide fast allocation but with no re-use of memory
- Strategy 2 combines pointer-increment allocation with a compactor – it provides slower allocation, but with re-use of memory

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

POINTER INCREMENT – Strategy 1



For Strategy 1 (fast allocation, no memory re-use) we consider the function *malloc* that takes an argument N giving the number of bytes to be allocated. There is no *free* function

Assume there is a heap and two administrative pointers TOP and END (that might be held in registers)

Imperative pseudo-code for malloc is very simple, with only three words of administrative overhead (TOP, END and tmp):

```
if (TOP + N > END)
then error("Out of memory");
else {tmp = TOP;
      TOP = TOP + N;
      return(tmp);}
```

Notice that this code can be modelled in Miranda if we view memory as being a list of char and memory addresses are indexes into that list. The heap could then be a three-tuple comprising the list and the two values for TOP and END:

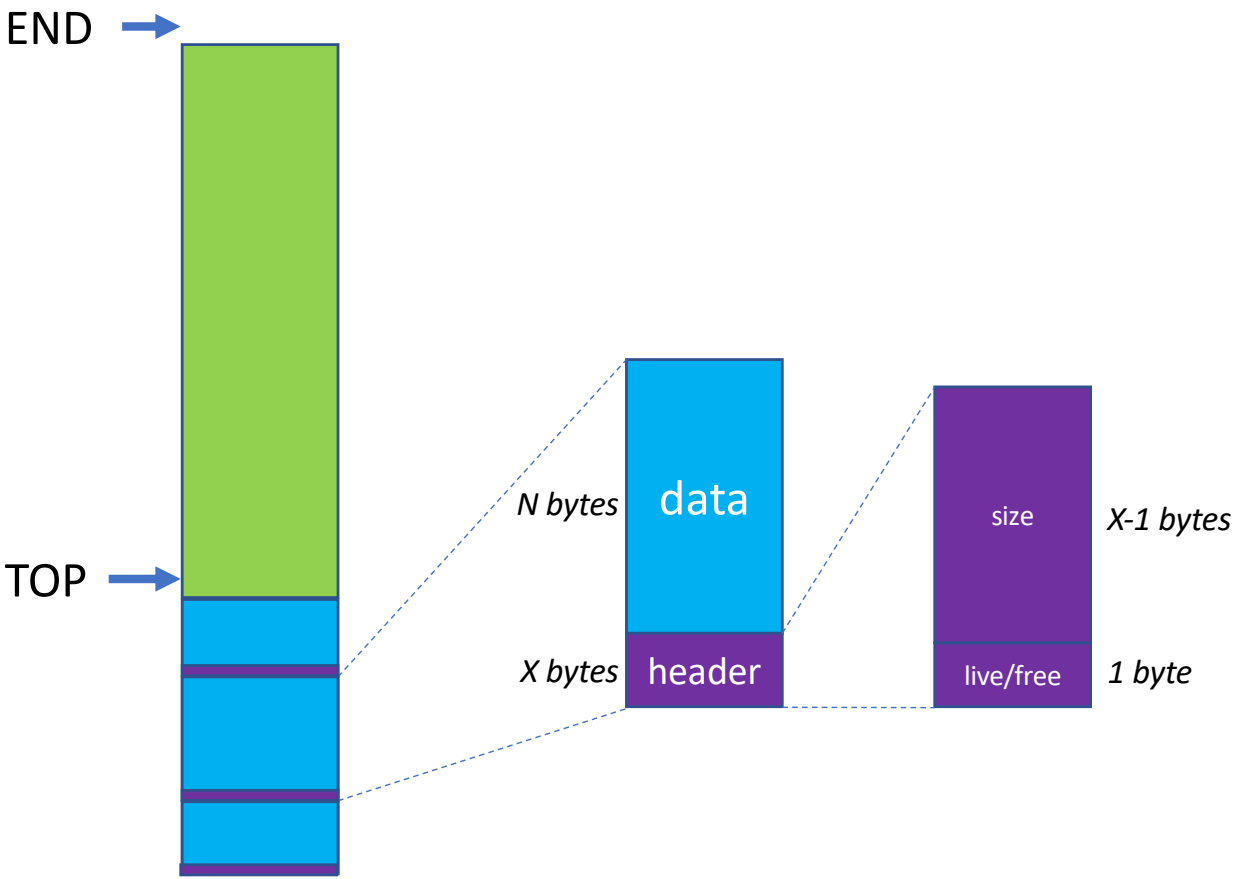
heap == ([char],num,num)

The Miranda version of pointer-increment *malloc* can be a function that takes a num and a heap and returns a two-tuple of the new heap and the returned index into the new heap:

```
malloc :: num -> heap -> (heap, num)
malloc n (h,t,e)
      = error "out of memory", if (t+n) > e
      = ((h,(t+n),e), t), otherwise
```

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

POINTER INCREMENT – Strategy 2



In order to support re-use of memory locations that are no longer used by the program, Strategy 2 adds a "header" to each data block (live or free) to hold:

- information about whether the block is live or free (this only needs one bit, but we may need to use a whole byte), and
- information about the size of the block

We will assume that there is a maximum limit to the block size that can be requested, so we know how many bits are needed in the block header to represent that size (e.g. 8 bits can represent 256 different sizes). Note that sizes could be counted in numbers of bytes or numbers of words (e.g. where 4 bytes = 1 word) – it is most convenient if the thing being counted is the same as what is meant by the argument N to *malloc* – assume we are counting bytes

Assume a block header is X bytes:

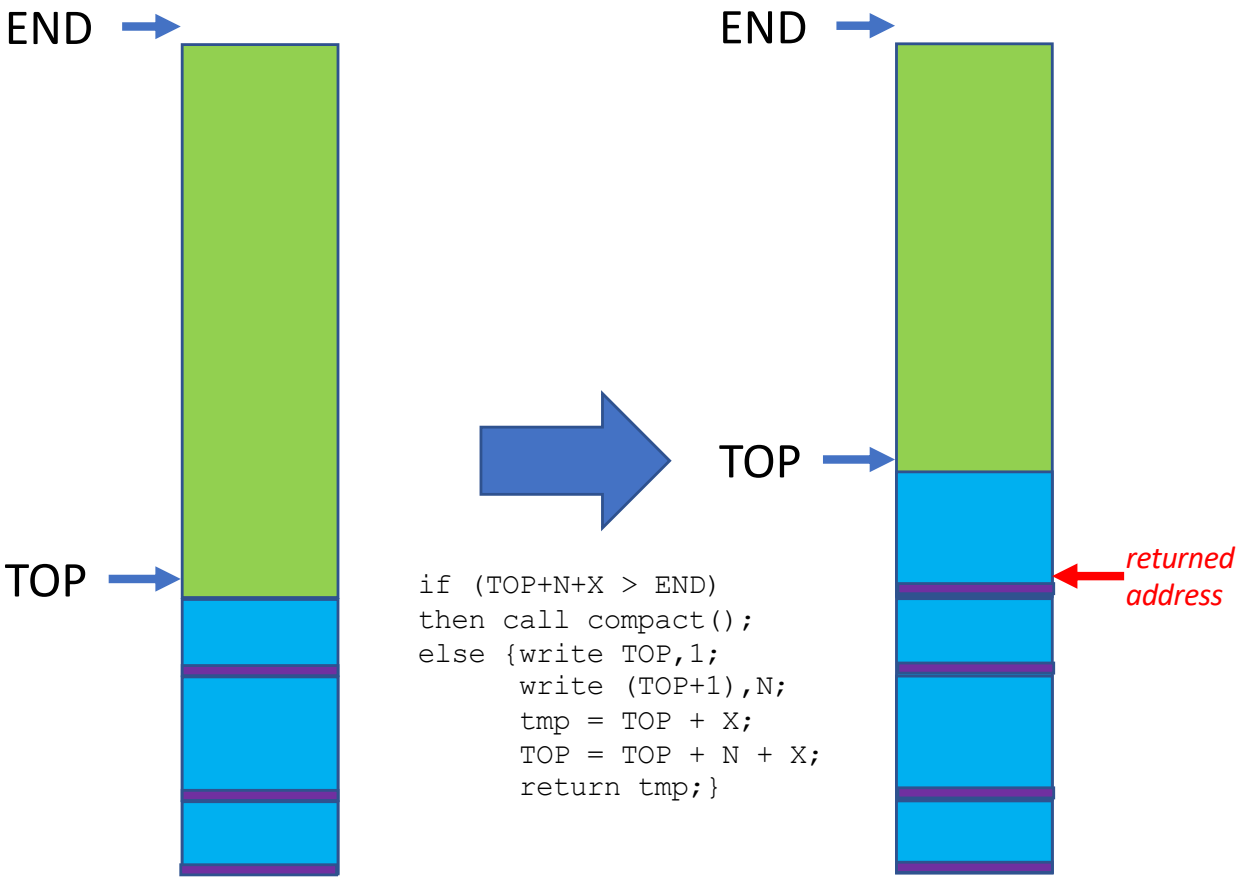
- 1 byte (one whole byte!) for live/free
- X-1 bytes (=(X-1)*8 bits) to represent block sizes from 0 to ($2^{8X-8} - 1$)

Because it is a pointer-increment strategy, it still uses administrative information TOP, END and the variable tmp

The code is shown on the next slide

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

POINTER INCREMENT – Strategy 2



Here is the imperative pseudo-code for *malloc* for Strategy 2 (assuming that 1 indicates "live" and 0 indicates "free"):

```
if (TOP+N+X > END)
then call compact();
else {write TOP,1;
      write (TOP+1),N;
      tmp = TOP + X;
      TOP = TOP + N + X;
      return tmp;}
```

In the above code "write TOP, 1" writes the value 1 (taking 1 byte) at address TOP and "write (TOP+1),N" writes the value N (taking X-1 bytes) at address TOP+1. Notice that:

- the address returned is the address of the start of the data area (not the address of the header) since the code making the call to *malloc* doesn't need to know about the header
- instead of returning an "out of memory" error, it calls *compact* (code not shown) which either resets TOP after compaction or returns an error if TOP has not changed

Here is the imperative psuedo-code for *free* for Strategy 2 (it is passed a memory address M as an argument):

```
write M-X,0;
return;
```

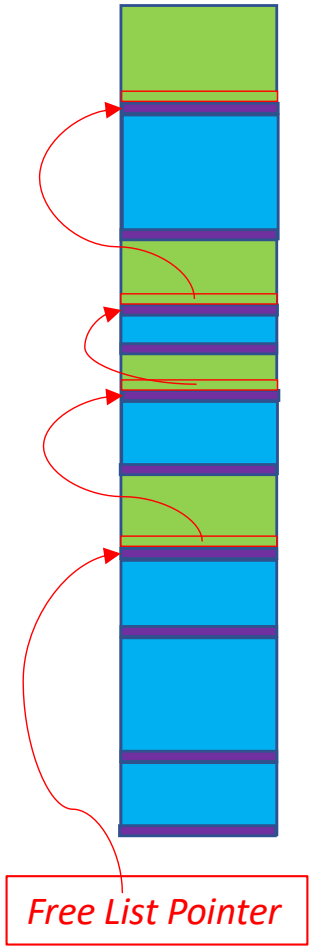
Notice how the *free* code subtracts X from M so that it can write 0 into the first byte of the header

The downside of compaction is the time it takes (an "embarrassing pause")

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – SEQUENTIAL FITS

- An in-place storage management technique (live blocks don't move)
- Free List: a linked list of all free blocks
- Free List Pointer: points to start of Free List
- *malloc* searches free list for suitable block
- *free* places freed block on free list
- Blocks have headers with liveness and size
- Each free block holds in its data area a pointer to the next block in the free list
- **minimum block data size** = size of pointer
- Typically *malloc* returns a pointer to the start of the data area



The free list mechanism is an in-place storage management technique. The **Free List** is a linked list of all the free blocks in the heap (the address of the first block on this linked list, if it exists, is held in a register and is typically called the **Free List Pointer**)

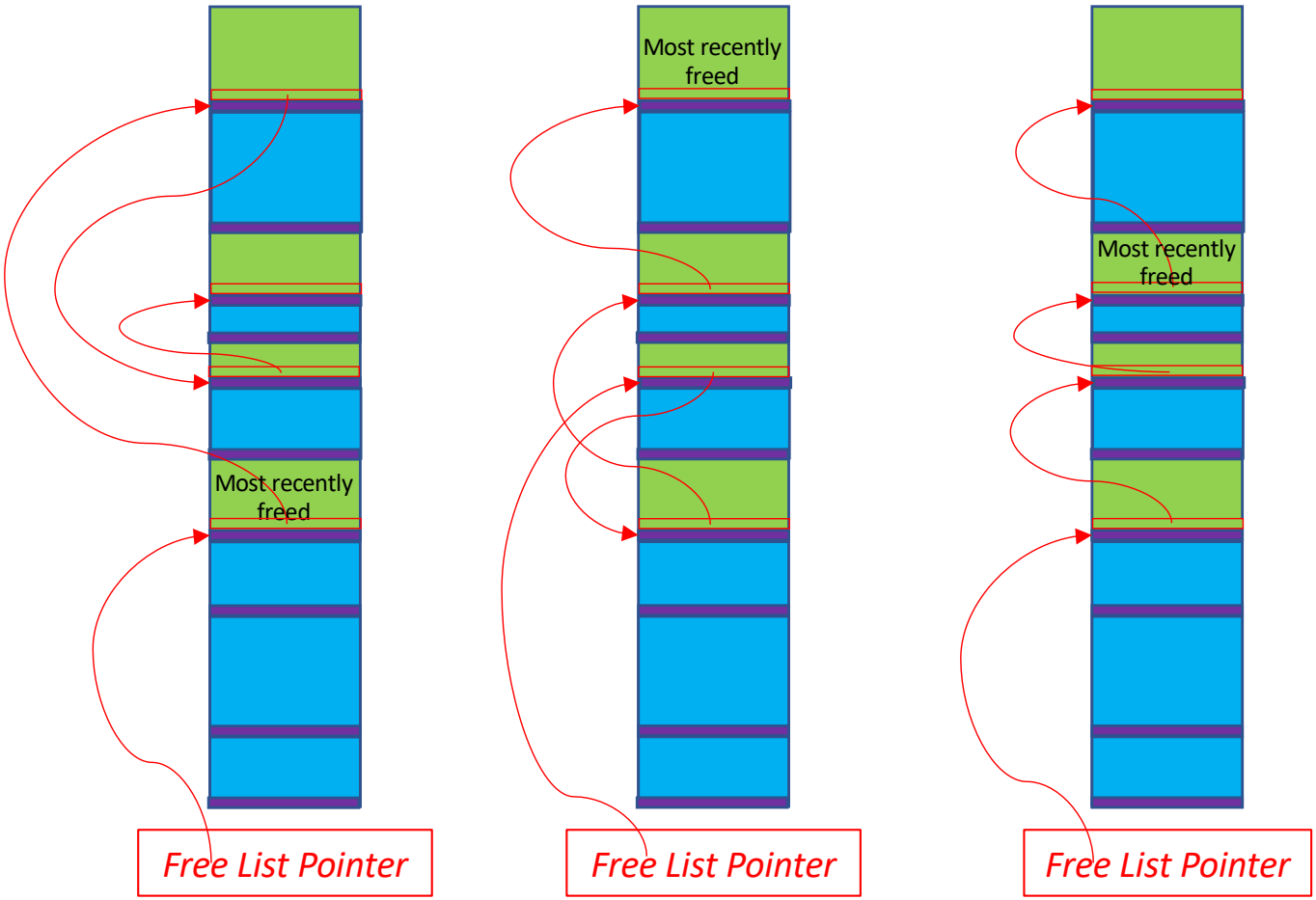
When the main program calls the *malloc* function, a suitably sized block is found on (or created from a larger block found on) the free list. When the main program calls the *free* function (passing it a pointer that was previously allocated), the block is attached to the free list so that it is available for re-use

Each block has (i) a header containing its liveness and its size, and (ii) a data area. A liveness bit is not strictly necessary – any block's liveness could be checked by traversing the whole free list, but that would be very slow. Each free block holds in its data area the address of (a pointer to) the header of the next free block in the free list (unless it is the last free block on the free list) – this means that there is a **minimum size** for the data area that is just big enough to hold a pointer (typically 4 bytes for a 32-bit pointer)

Typically the *malloc* function returns a pointer to the start of the data area (though in some older texts you may find algorithms described that return a pointer to the header, even though the program that calls *malloc* should never need to read from or write to the header)

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – SEQUENTIAL FITS



LIFO ordering FIFO ordering AO ordering

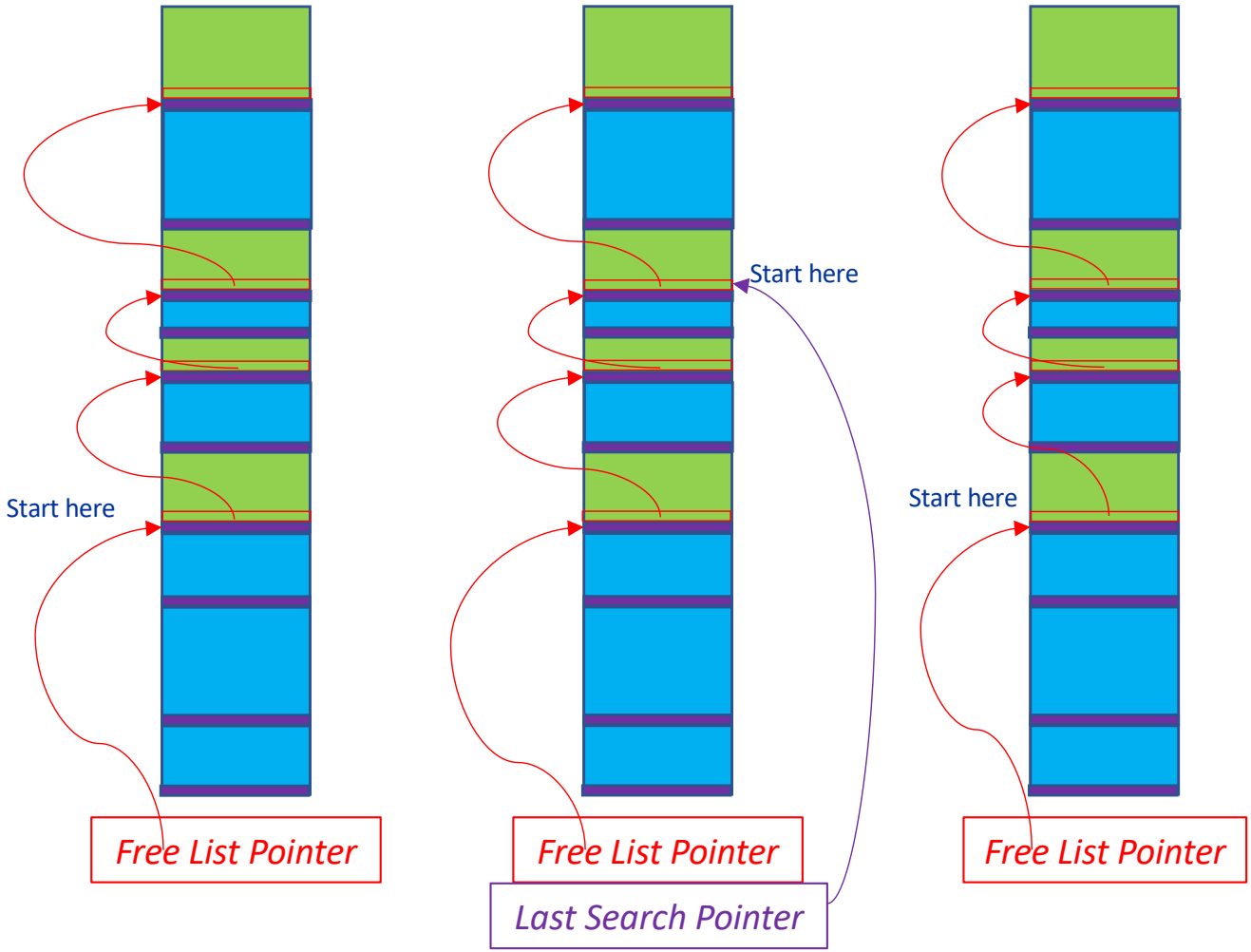
Ordering policies

Free list ordering policies (assuming that allocations are made by searching from the start of the free list):

- a Last-In-First-Out (LIFO) ordering policy will place a newly freed block at the start of the free list (thus, the mostly recently freed block is considered first for allocation)
- a First-In-First-Out (FIFO) ordering policy will place a newly freed block at the end of the free list (thus the oldest block – the first one placed on the free list – is considered first for allocation)
- an Address-Ordered (AO) ordering policy will place a newly freed block into the appropriate position on the free list so that the free block with the lowest address is at the start of the free list and all blocks on the free list increase in address until the free block with the highest address is at the end of the free list (thus the free block with the lowest address is considered first for allocation, which tends to cluster live memory blocks at the lower end of memory which of course is better for virtual memory performance)

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – SEQUENTIAL FITS



AO First-fit AO Next-fit AO Best-fit

Allocation policies

There are several popular allocation policies (these are often called "sequential fits" policies). For requested size N, *malloc* looks for a free block with size Y such that Y >= X + (max (N,4)). Recall the minimum data size of 4 bytes for a pointer

The next slide will discuss how the block can be split if Y >> X + (max(N,4)). On this slide we assume an exact fit and the entire block is used to fulfil the request

There are three popular allocation policies:

1. First-fit - start at the first block on the free list and either use that first block if it is big enough, or follow the pointers in the linked list until a sufficiently large block is found (see left figure)

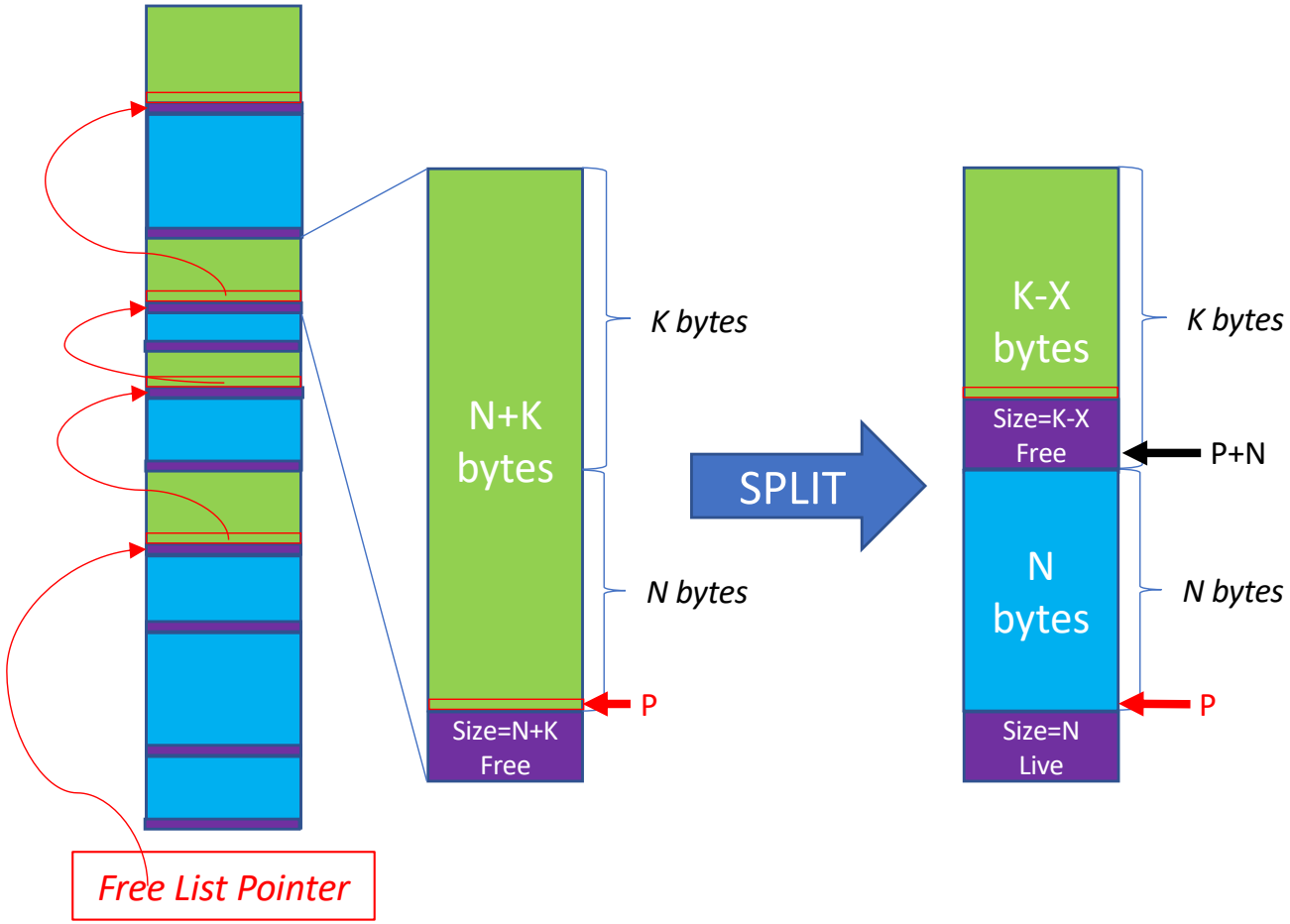
2. Next fit - keep a second pointer in a register to remember the point in the free list where the previous allocation search was stopped; then start the search from that point, wrapping around to the start of the free list if necessary (see centre figure)

3. Best fit - scan the entire free list and find the block with big enough Y that has the smallest Y – (X + (max(N,4))) (see right figure)

4. Worst fit - scan the entire free list and find the block with the largest data size (not shown)

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – SEQUENTIAL FITS



Splitting blocks

A free list allocator searches the free list to find a free block whose data area is big enough to accommodate the memory requested in the call to *malloc*

If the requested size is N (for simplicity assume N>4), then either:

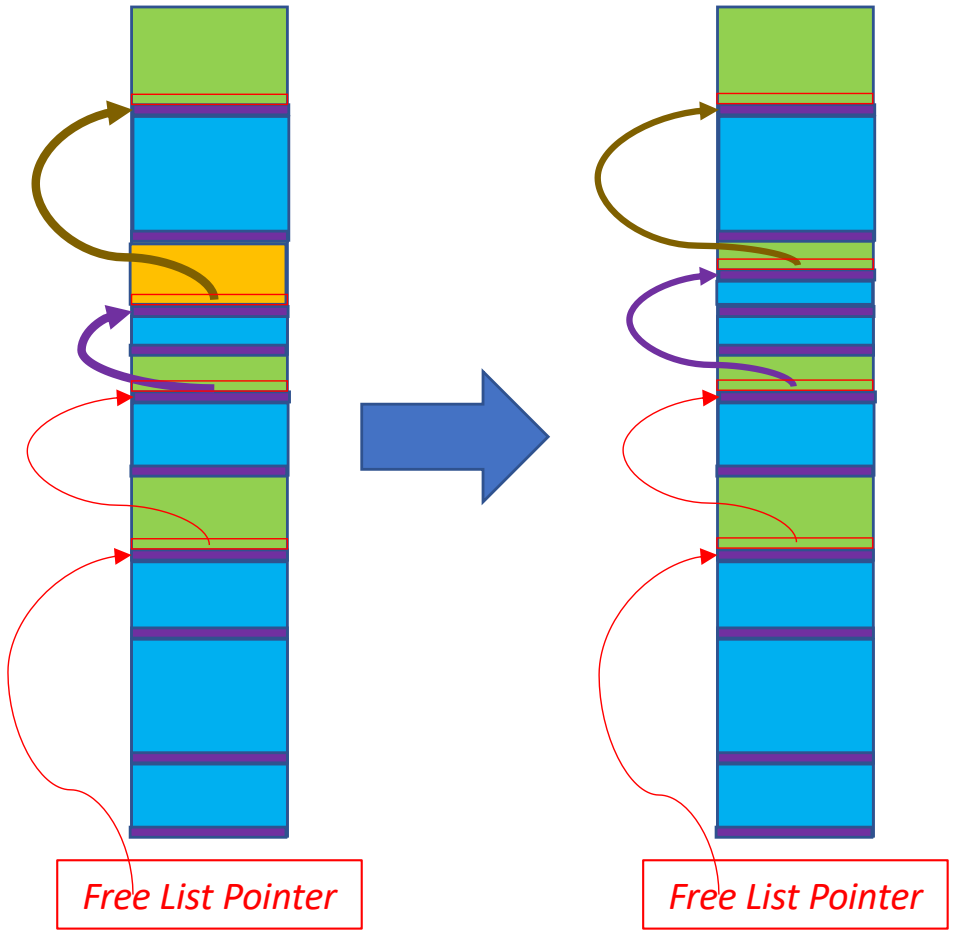
- malloc cannot find a free block whose data size is >=N, and an error results
- or malloc finds a free block at address **P** whose data size is N+K where K>=0

Now consider the size of K:

- If K is large then *malloc* splits the data area to create two blocks – one live and one free
  - *malloc* updates the existing header to indicate Live and data size N
  - At address **P**+N a block header is created to indicate Free and data size K-X
  - *malloc* restructures the free list (see next slide) and adds the newly constructed free block to that free list
  - *malloc* returns the address **P**
- If K=0 (an exact match) or if K is too small for *malloc* to create a new free block with a header and data area large enough to accommodate a single pointer, then *malloc* restructures the free list, updates the header of the found block to indicate is is now Live, and then returns the address **P** (if K is too small to split, or N<4, we'll get internal fragmentation)

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – SEQUENTIAL FITS



Restructuring the free list

When a free block is found that is large enough to satisfy an allocation request, *malloc* must (as mentioned on the previous slide) restructure the free list

In the figure on the left, assume that the orange block (currently free) has been identified as the block to be used for the allocation request. Also assume that this orange block is sufficiently large to be split (as explained on the previous slide). Finally, assume that the free list is operating an AO ordering policy

In this case *malloc* must:

- copy the brown pointer (found at the start of the data area of the orange block) to the start of the data area of the new (split) free block
- change the purple pointer (found at the start of the data area of the free block that points to the orange block) so that it now points to the header of the new free block

If the orange block were not big enough to split, *malloc* would simply overwrite the purple pointer with a copy of the brown pointer

There are two ways for *malloc* to find the purple pointer:

- traverse the free list again, from the start, until the purple pointer is found
- when traversing the free list the first time, use two pointers where the second lags one block behind the first

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – SEQUENTIAL FITS

Comparison of allocation policies:

- First-fit: many small blocks at the start of the free list
  - degrades performance
- Next-fit: improves the first-fit problem of small blocks at the start of the free list, but in practice tends to increase fragmentation of the heap
- Best-fit: should lead to very many small free blocks
  - in practice this does not appear to happen
  - normally slow, but some algorithms improve speed

Comparison of Allocation policies

First-fit: larger blocks near the start of the free list tend to be split first, resulting in many small blocks at the start of the free list – over time, this degrades performance since they are unlikely to be large enough for future allocation requests, but must always be traversed and checked during search

Next-fit: improves the problem of small blocks at the start of the free list, but in practice tends to increase fragmentation of the heap

Best-fit: theoretically, we might expect it to lead to very many small free blocks, but in practice this does not appear to happen. It is normally slow, but some algorithms improve best-fit speed

Other policies have also been suggested, such as "worst-fit" and "good-fit" (like best fit, but stop when K is small enough), but most practical implementations tend to use one of the main three policies described above

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

SEGREGATED FREE LISTS

- A set of free lists
  - each manages blocks of a chosen size (any block is good to satisfy a request for that size)
  - or each manages blocks in a small range of sizes (use first-fit, next-fit or best-fit within each free list)
- Use an array of pointers, each one being the Free List Pointer for one of the free lists
- Or use a tree of Free List Pointers

Segregated free lists

Searching for a free block of the right size is slow – each block's header must be read and tested against the required size, then the pointer to the next block must be read and used to read the next block on the free list, and so on

An alternative is to manage a set of free lists. Each such free list could either be:

- a linked list of free blocks all of the same size (different lists would be for different sizes) – therefore *any* block from a chosen free list would be the required size, or
- a linked list of free blocks all with sizes within a small range (different lists would be for different ranges) – any sequential fits mechanism could be used, with the likelihood of a match being higher by pre-choosing which free list to search

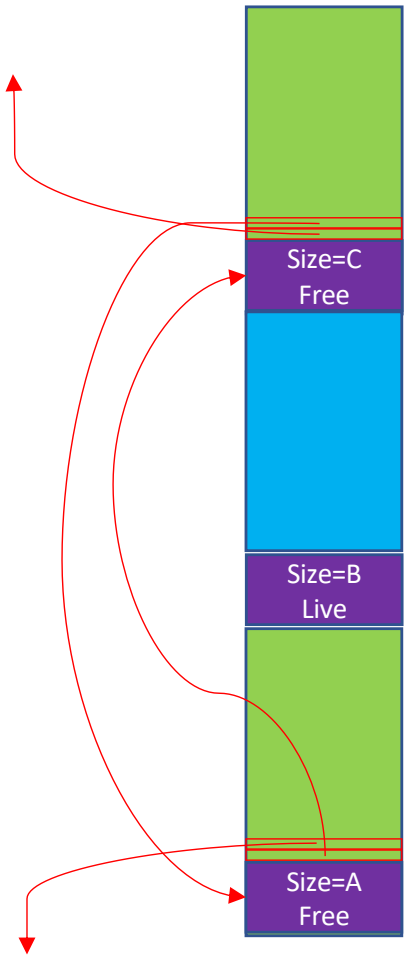
The set of free lists requires a set of free list pointers. These could be managed for example either:

- as a static array of free list pointers (each pointing to the start of a different free list), or
- as a dynamic data structure such as a tree of free list pointers (again, each pointing to the start of a different free list)

The dynamic data structure would be suitable for a large number of free lists, where that number changes at runtime

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

DOUBLE-LINKED FREE LISTS



Double-linked free lists

Free lists are typically single-linked, but some implementations are double-linked lists (so that each free block holds a pointer to the next block in the free list and to the preceding block in the free list)

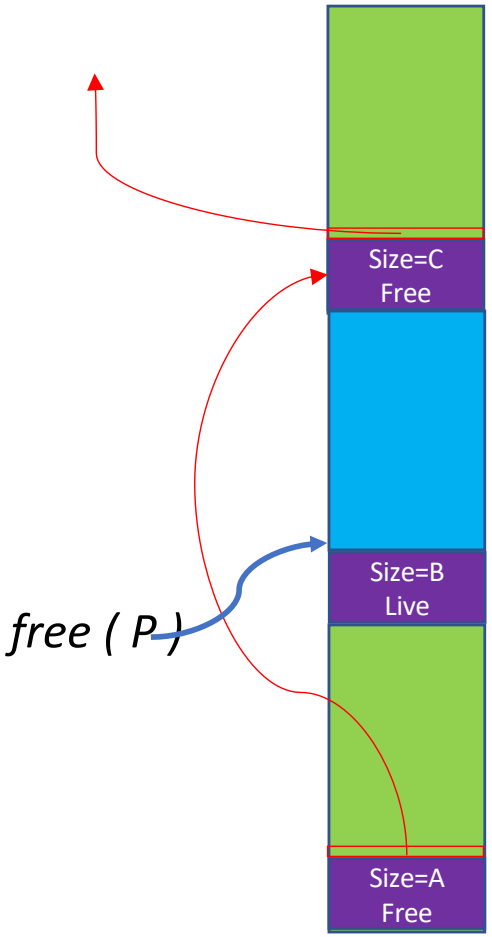
This improves the speed of splitting blocks and adding a free block into the free list, especially where full coalescing is supported (see later)

However, now there must be a larger **minimum block size** for its data area, so that it is big enough for two pointers

The figure on this slide shows the memory layout for a double-linked free list

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – COALESCING AND BOUNDARY TAGS



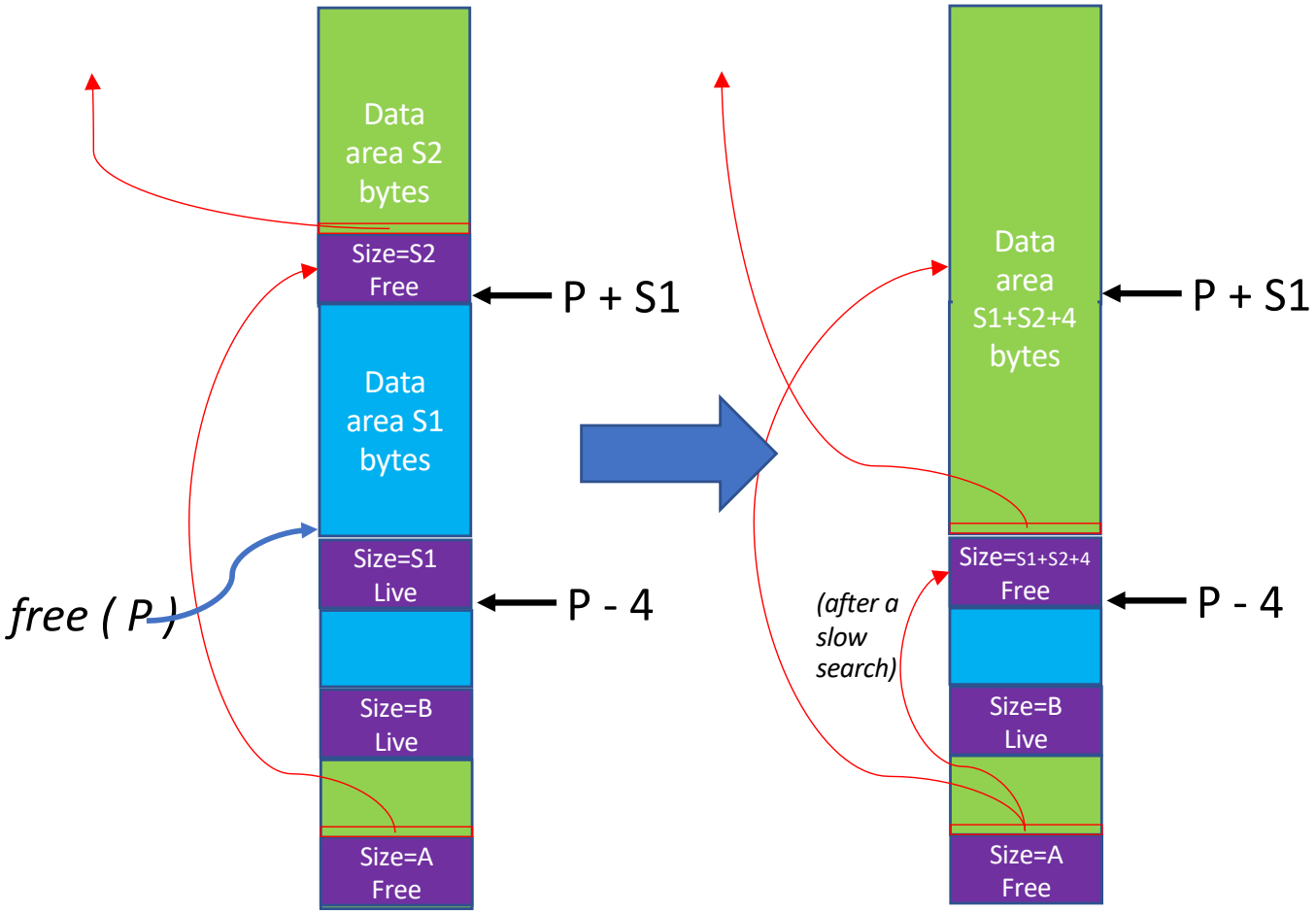
Some implementations support coalescing of adjacent free blocks to reduce that type of fragmentation. Specifically, coalescing is part of the *free* function where a freed block's neighbours in memory are checked to see whether **one** or **both** are free (and if so, to merge with them to create a single larger free block)

Note that blocks that are neighbours in physical memory will not necessarily be neighbours on the free list. The exception is where an AO ordering policy is used, because neighbouring free blocks in physical memory will then always also be neighbouring blocks on the free list

We can implement coalescing in a practical way if we are prepared to keep additional information in each block header

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – COALESCING AND BOUNDARY TAGS



Coalescing with the next block only

We assume the header includes a single bit (0 or 1) to indicate whether this block is live or free (called the "free bit" or the "liveness bit")

The *free* function can attempt to coalesce with the **next** block in memory if it happens to be free

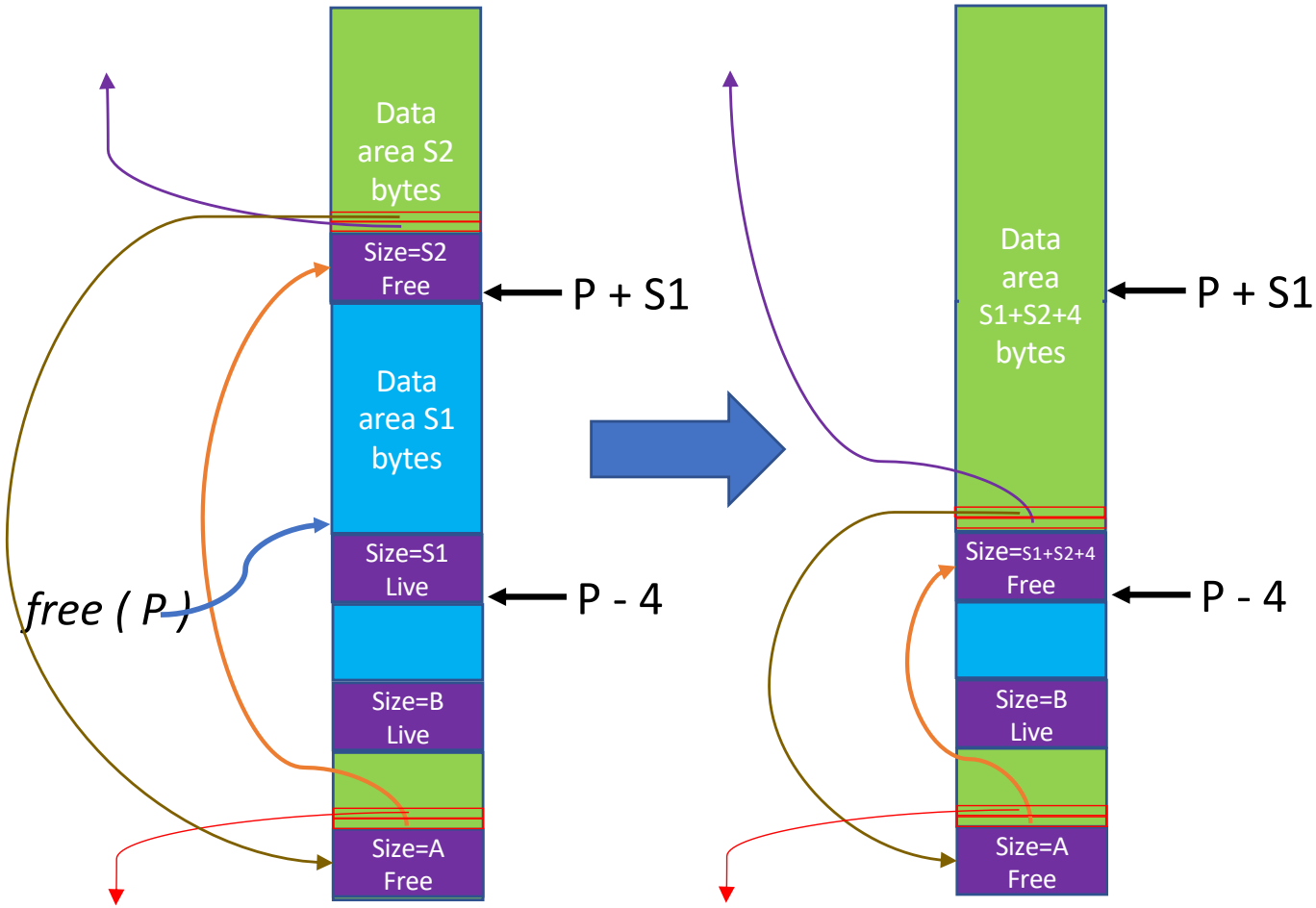
- *free* receives as an argument a pointer to the start of this block's data area (P)
- *free* can read the header of this block at address P-4, and get this block's size (S1)
- *free* can read the header of the next block in physical memory by reading address P+S1 – it can then read the next block's liveness and size S2
- if the next block is free, then the two blocks can be coalesced by manipulating pointers (see the figure)

However, the previous block on the free list needs to have its pointer updated to be "P-4". This can only be done by traversing the free list, and comparing the "next" pointer against the range of addresses for the new large free block data area (this is slow)

A faster method is shown on the next slide

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – COALESCING AND BOUNDARY TAGS



Coalescing with the next block only

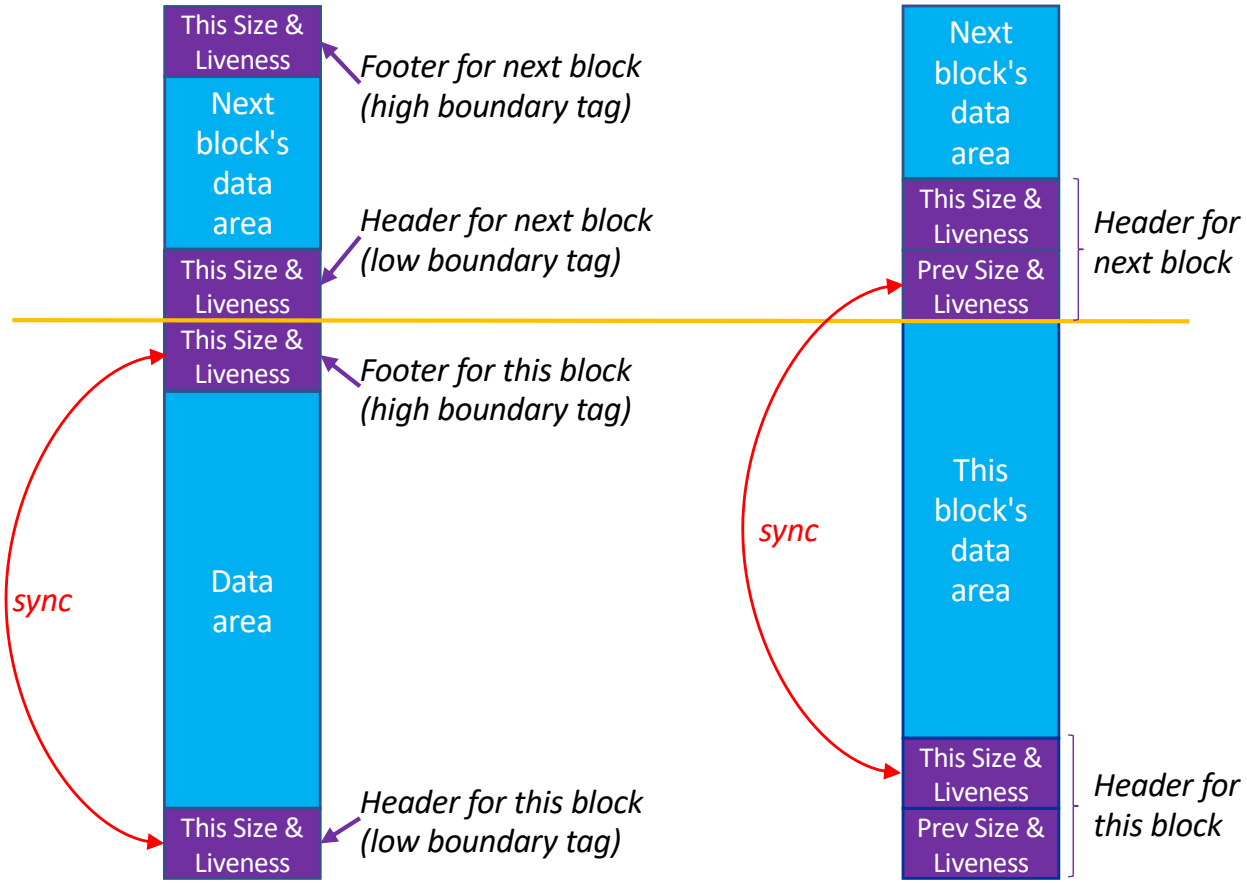
A faster way to coalesce with the next block (only) is to use a double-linked list, as illustrated in this figure

- *free* can read the header of the freed block at address P-4, and get this block's size (S1)
- *free* can read the header of the next block in physical memory by reading address P+S1 – it can then read the next block's liveness and size S2
- if the next block is free, then the two blocks can be coalesced (see the figure)
- Both the address of the previous free block (brown arrow) and the address of the next free block (purple arrow) can be copied from the data area of the next block into the data area of the newly created free block
- The brown pointer can be followed to find the previous block in the free list, and its "next" pointer (orange) can be updated to be P-4

There is still no efficient way to know whether the previous block in physical memory is free (because the size of the previous block's data area is not known, and we therefore cannot find the previous block's header) although an inefficient mechanism could scan the entire heap to find the previous block in physical memory

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – COALESCING AND BOUNDARY TAGS



Knuth's boundary tags

Clack's boundary tags

To provide efficient support for full coalescing we also need to store the size and "liveness" of the previous block in a place that can be accessed by the current block. There are two ways of describing how to do this (which very nearly amount to the same thing): Knuth's boundary tags and what I will call Clack's boundary tags

Knuth's boundary tags mechanism requires that every block has both a header (a "low boundary tag") and a footer (a "high boundary tag") - the header occurs (as usual) immediately before the data area and stores the "liveness" of this block and the size of its data area – the footer occurs immediately after the data area and stores identical information. The header and footer for a block must always be fully synchronised, and have identical contents

Clack's boundary tags mechanism requires that every block has a double-sized header that includes (i) the liveness and size of this block and (ii) the liveness and size of the previous block. The part of the header giving information for this block must always be fully synchronised with the part of the next block's header giving information for its previous block, and have identical contents

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – COALESCING AND BOUNDARY TAGS

- Knuth: last block in the heap has an unused high boundary tag
- Clack: first block in the heap has unused information about "liveness and size of the previous block"
- Both can be used for full coalescing
- Implementation note:
  - A 16-bit signed integer can hold:
    - "liveness" in the sign
    - "size" in the value
  - Two 16-bit signed integers fit in a 4-byte header
  - Good for block data sizes <= 32Kbytes - 1

With Knuth's boundary tags mechanism the very last block in memory has a high boundary tag with information about "liveness and size of the current block" that is never used. By contrast, with Clack's boundary tags mechanism the very first block in memory has header information about "liveness and size of the previous block" that is permanently set to be "Live"/"0" and is never used. Apart from this difference, they are essentially the same mechanism (though the latter may be easier to code)

In either case it is possible to use the pointer passed to the *free* function (P) to find the liveness and size of the previous block in physical memory, and then it is possible to coalesce with that previous block (see next slide)

Implementation note

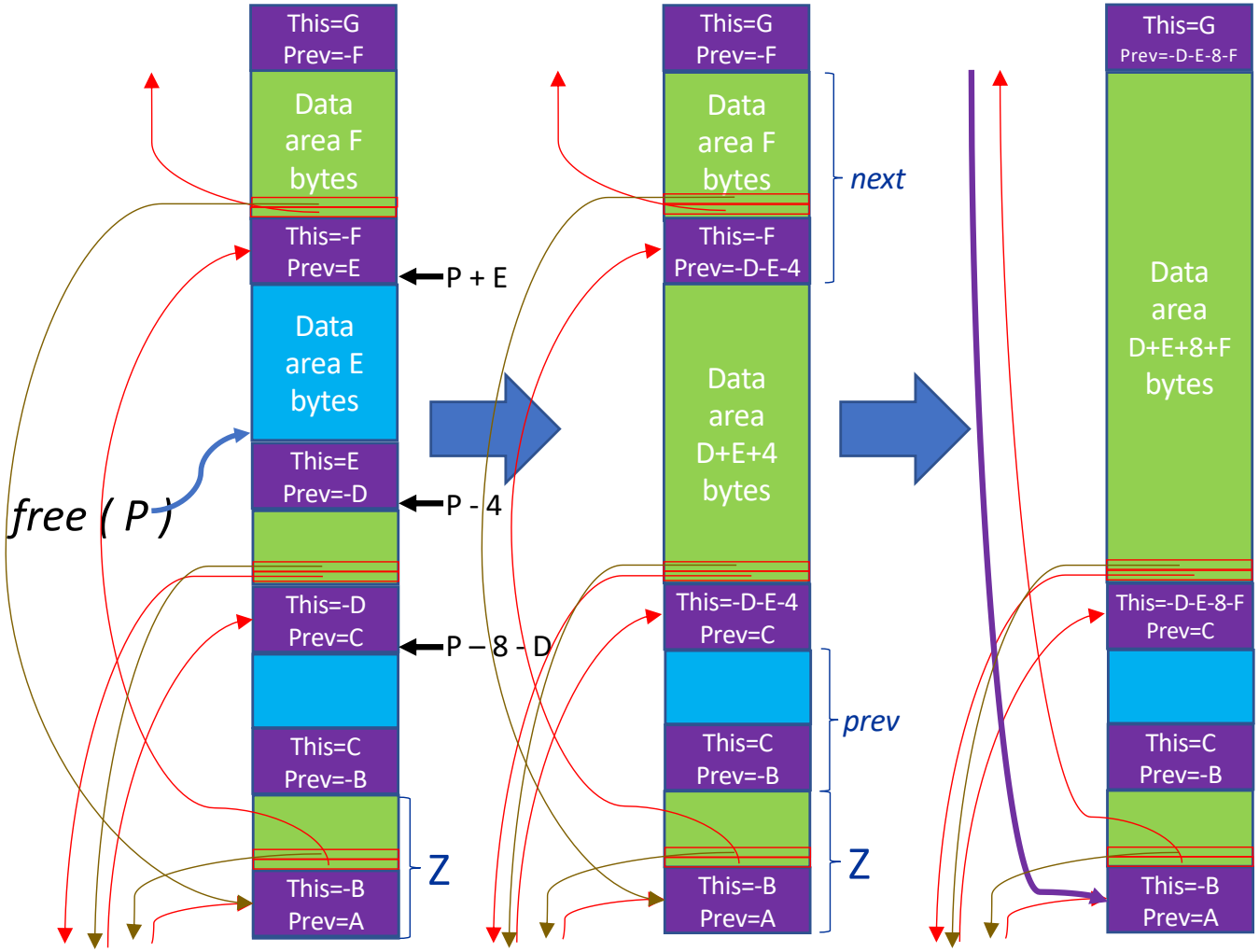
A 16-bit signed integer can provide both availability using the sign bit (e.g. positive numbers are live, negative numbers are free) and size (the absolute value of the number), and is appropriate for live block data sizes <= 2^{15}-1 "fields"

Two 16-bit signed integers can be held in a 4-byte header

A "field" is generally a separately-addressable unit of memory. We assume for these lectures that each counted "field" is 1 byte, which means a maximum data size of 32,767 bytes (i.e. 32Kbytes - 1)

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

FREE LIST – COALESCING AND BOUNDARY TAGS



Using Clack's boundary tags and a LIFO double-linked free list: when a block is freed, the liveness of the previous block is read from the current header

Merge with previous block: If the previous block is free, its size is used to jump to its header and modify it to indicate that its size is now increased by the size of the freed block and its header. That size is then used to jump ahead to change the next header "previous liveness and size" to be Free and the newly merged size

Merge with next block: the next header also gives the liveness and size of the next block: if free, then the header of the new freed block is increased by the size of the next block and its header. The "previous" information held in the block AFTER the "next" block must also be modified as shown in the figure

Coalescing with the next block requires changes to the free list. If there were no merger with the previous block (not shown), make the free block that used to point to the next block (call it "Z") now point to the new free block and copy the next block's 2 pointers into the new free block's data area. If also merging with the previous block, copy the next block's "next" pointer to Z's data area. Double-links make this faster. In both cases don't forget the purple backwards pointer (see figure)

Performance: O(free list length) for a single-linked list and O(1) for a double-linked list

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

The lecturer ends with a VERY SIMPLE example of administering heap memory, including an initial layout of the heap and then the memory layout after successive calls to *malloc* and *free*

For convenience, memory is now shown horizontally, with low heap memory on the left and high heap memory on the right. For simplicity we assume a heap size of H bytes (each byte has a separate address), the heap starts at address 0, and the last byte of the heap is at address H-1

VERY SIMPLE MEMORY MANAGEMENT EXAMPLE

Address = 0

Address = H - 1

Free List Pointer :

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

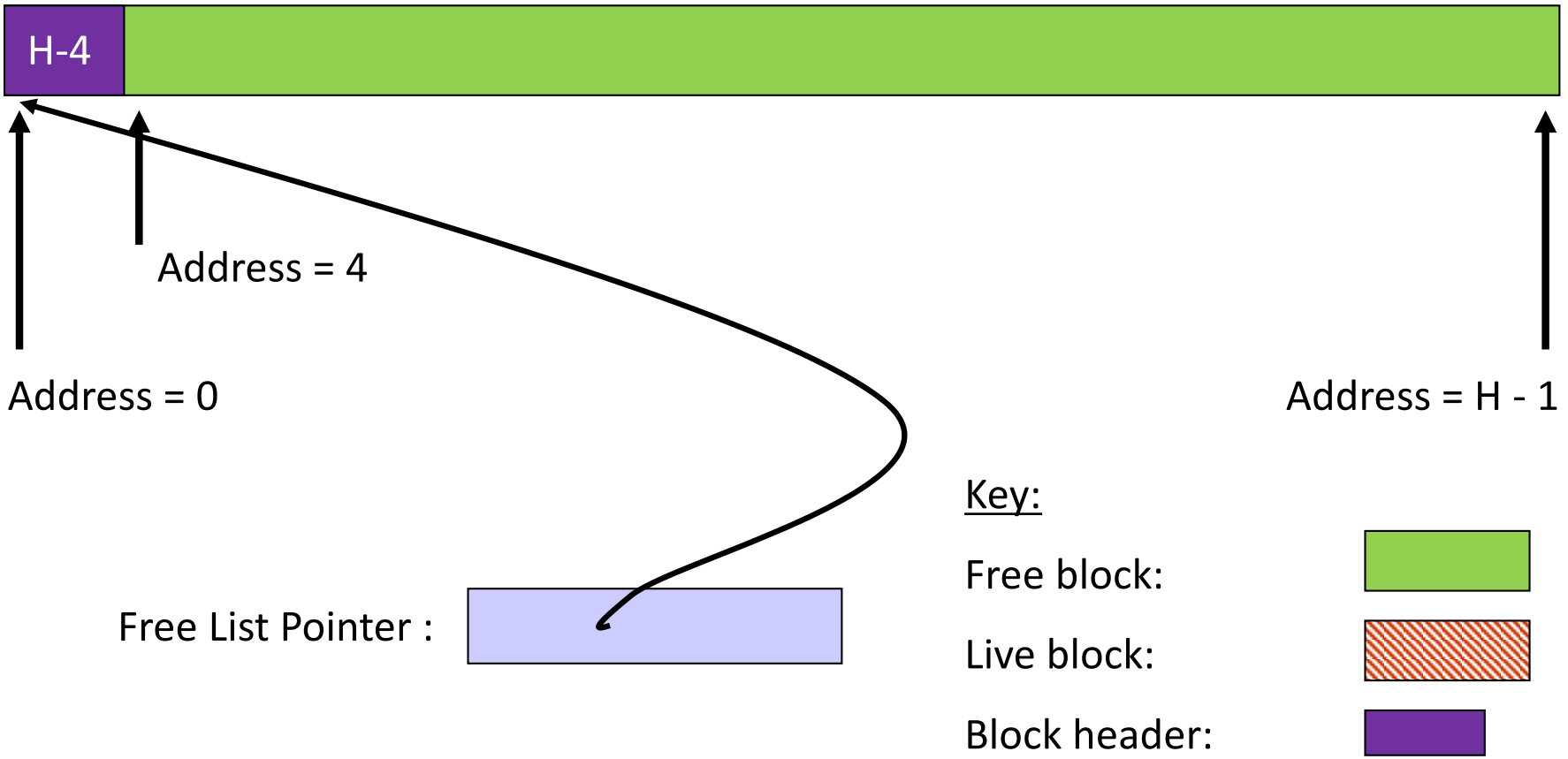
MEMORY MANAGEMENT EXAMPLE

Initial layout

Initially the heap is structured as a single free block whose data area has size H-4

This SIMPLE scheme has no footers: each header is 4 bytes and contains liveness and the size of the data area. Note the implied limit on H: at most (with a single liveness bit) there are 31 bits to represent 2^31 different sizes

The Free List Pointer holds the address of the header block of the first (and only) block on the free list



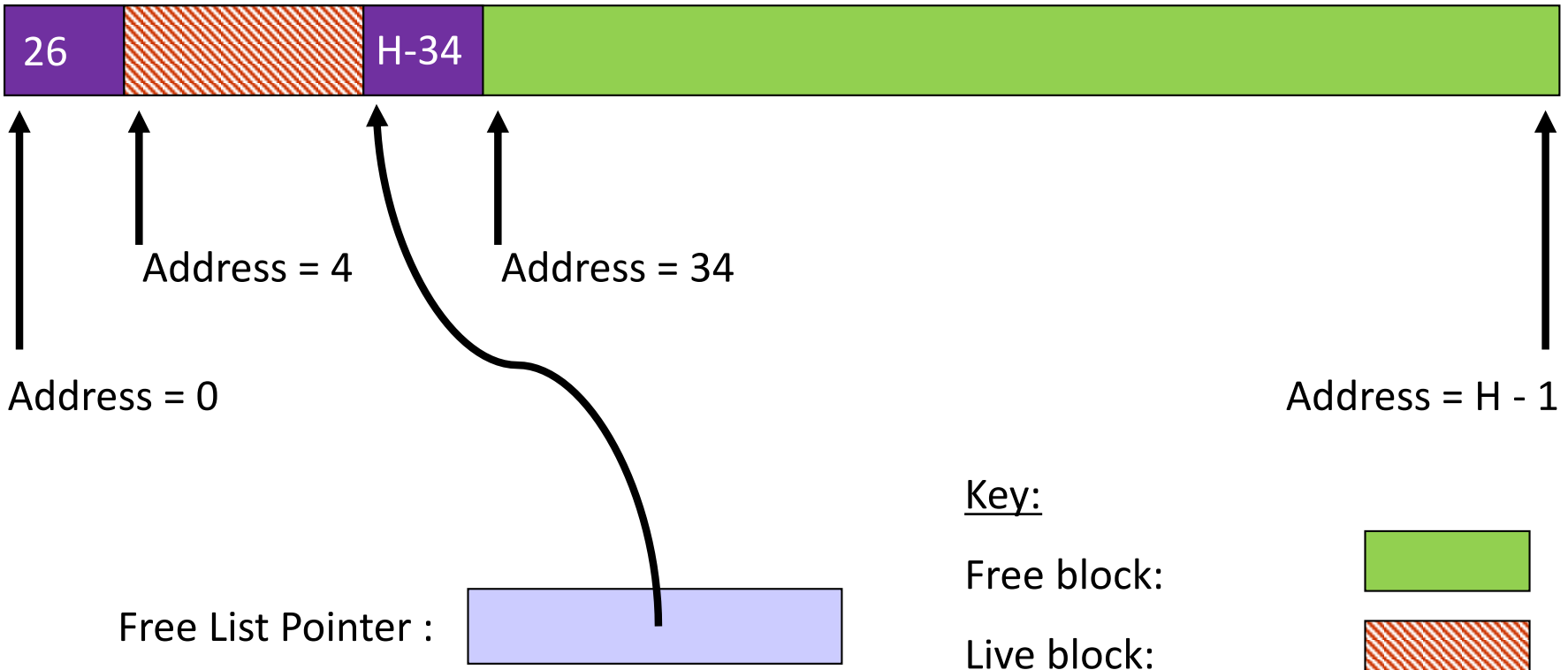
FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

This figure shows the memory layout of the heap after the program requests a block of 26 bytes

The previous free block has been split in two: we will choose that the lower part is now a live block with header and data area of size 26, and the upper part is a free block with a header and data area of size H-34. Alternatively we could choose to make the lower part free and the upper part live. *malloc* returns 4

The Free List Pointer is updated to point to the header of the free block

MEMORY MANAGEMENT EXAMPLE



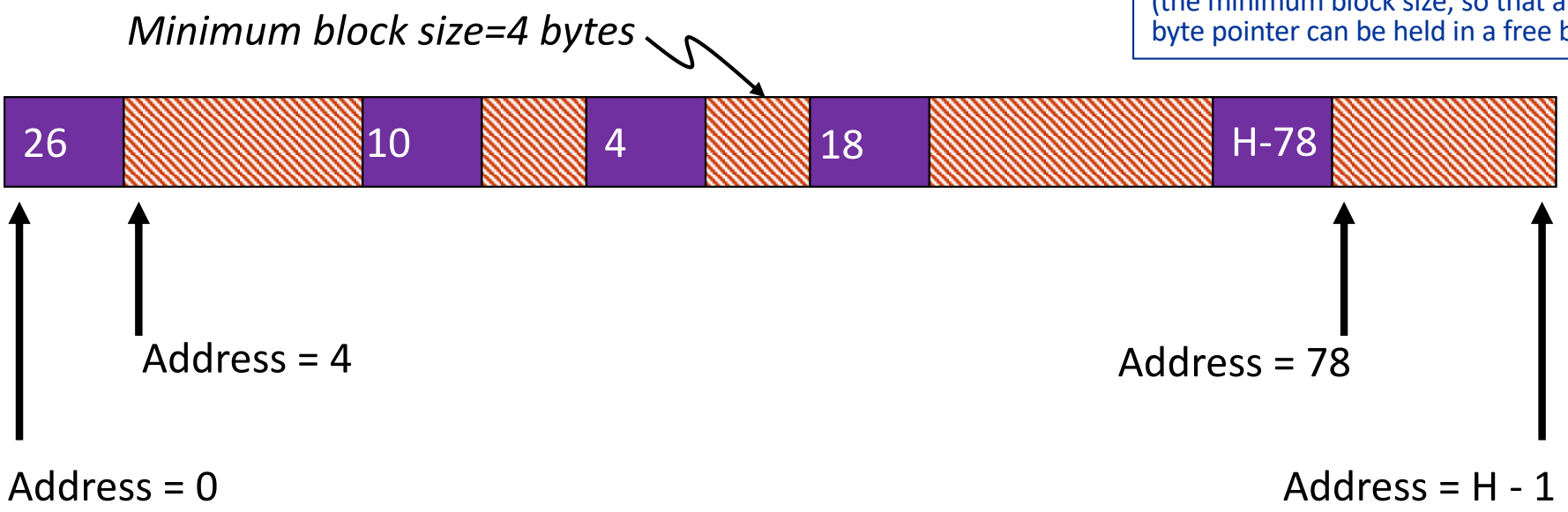
FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

This figure shows the memory layout of the heap after the program has requested further blocks of sizes 10, 2, 18 and H-78 (not shown to scale)

There are now no free blocks and the Free List Pointer is "null" (in a real situation the heap would not start at address 0, so 0 could be used for "null")

Notice that the request for 2 bytes has caused a block of 4 bytes to be allocated (the minimum block size, so that a 4-byte pointer can be held in a free block)

MEMORY MANAGEMENT EXAMPLE



Free List Pointer : *(null)*

Key:

Free block:

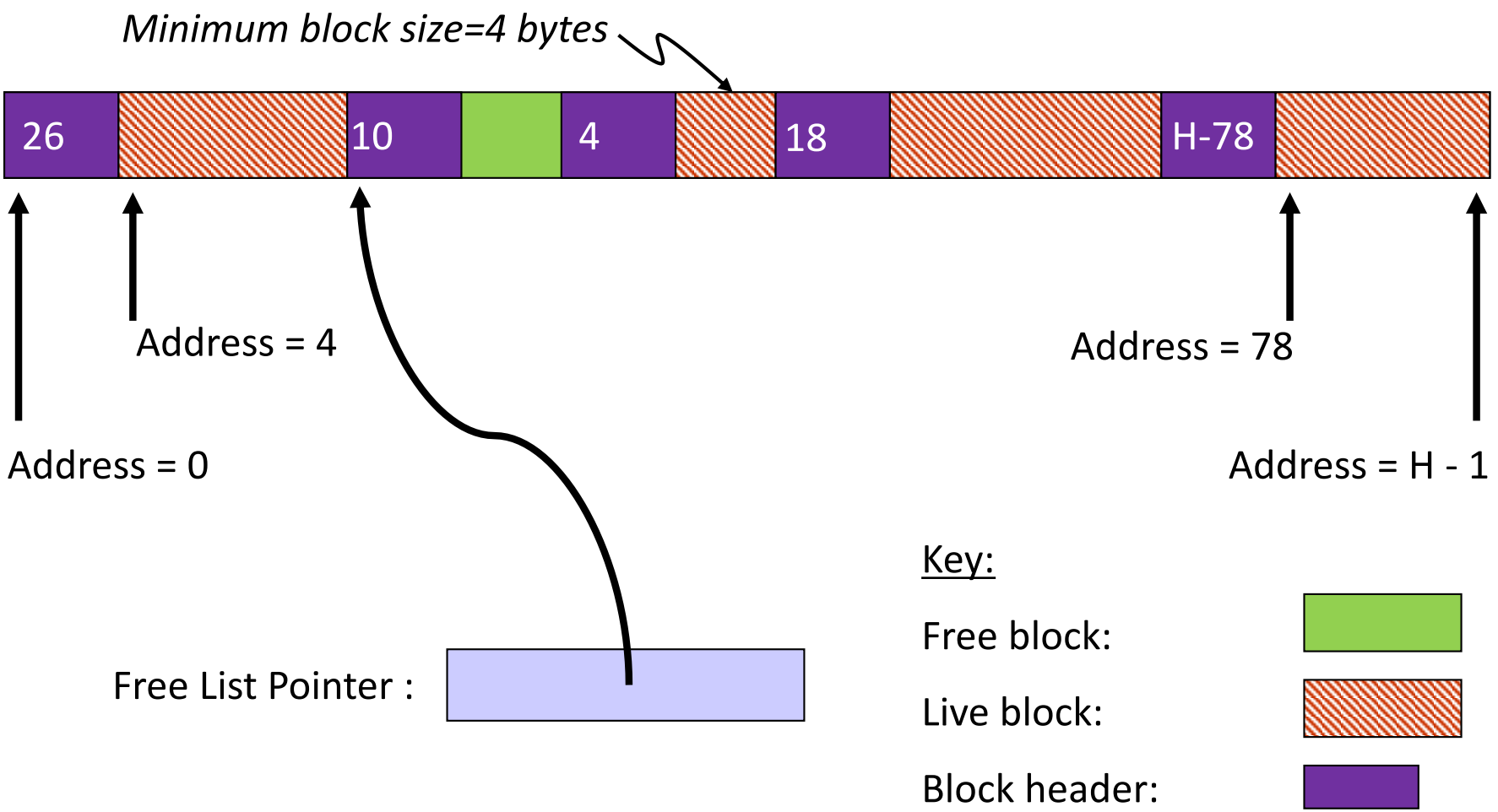
Live block:

Block header:

FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

This slide illustrates the memory layout after the program frees one block

MEMORY MANAGEMENT EXAMPLE

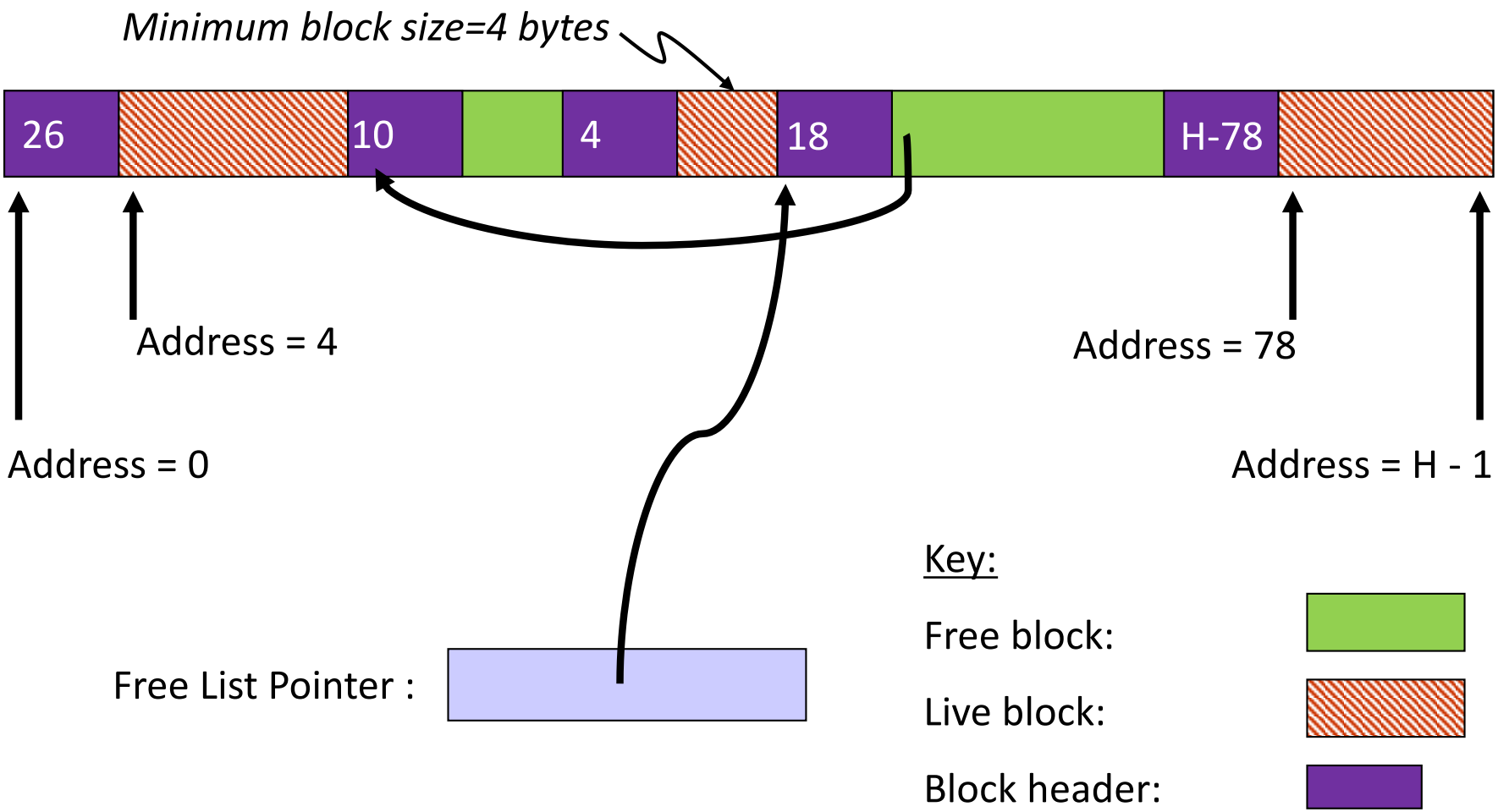


FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

This is the memory layout after the program has freed a second block (we assume that *free* uses LIFO ordering policy).

Also note that the free list is SINGLE linked (and with no boundary tags) – it can support partial coalescing but it requires a slow search to do this.

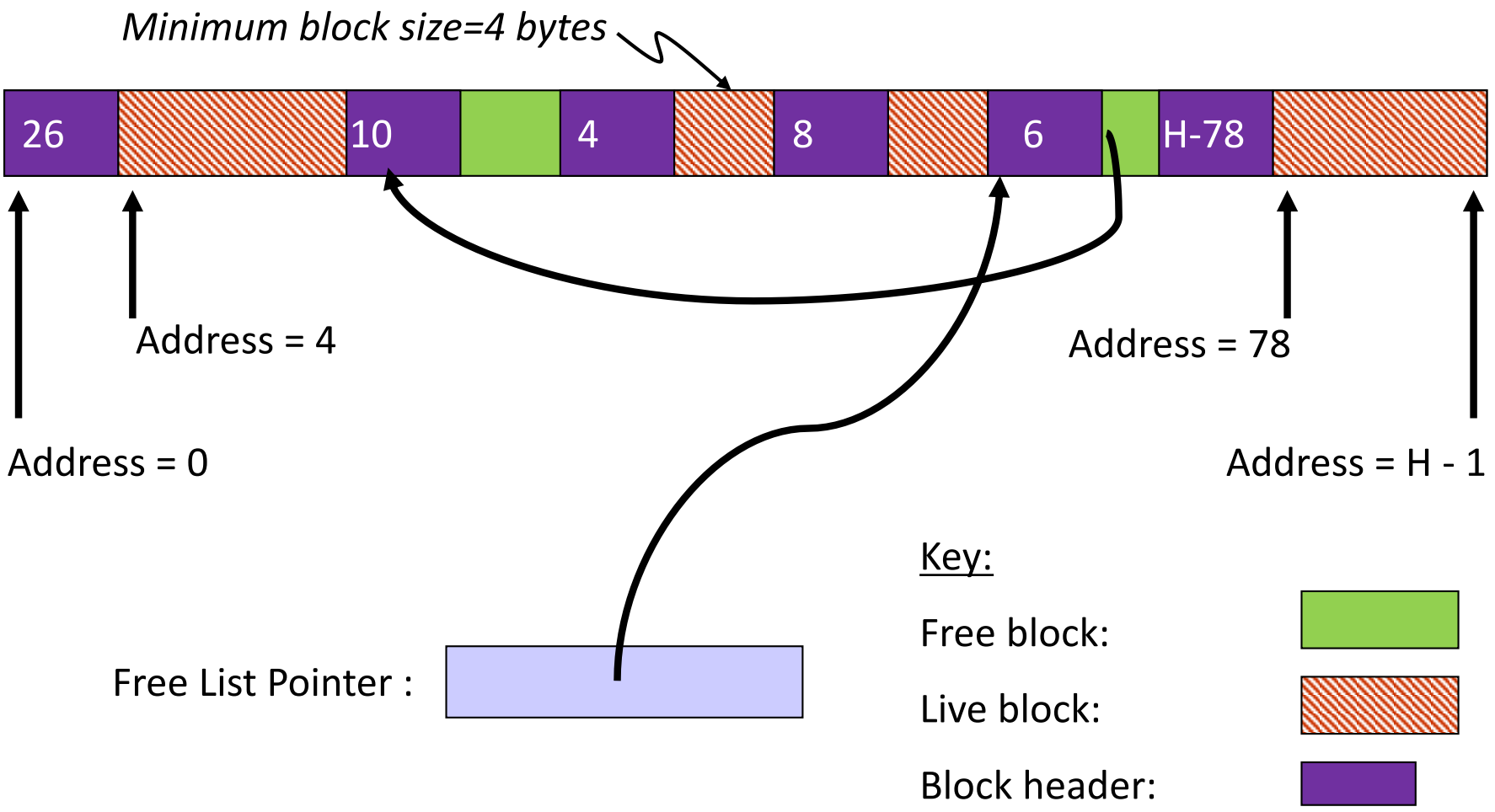
MEMORY MANAGEMENT EXAMPLE



FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

Following a request from the program for a block of size 8 bytes, the first free block on the free list (first-fit allocation policy) is large enough and is split into a live block (4 bytes header + 8 bytes data area) and a new free block (4 bytes header + 6 bytes data area), which is added to the front of the free list (LIFO ordering policy)

MEMORY MANAGEMENT EXAMPLE



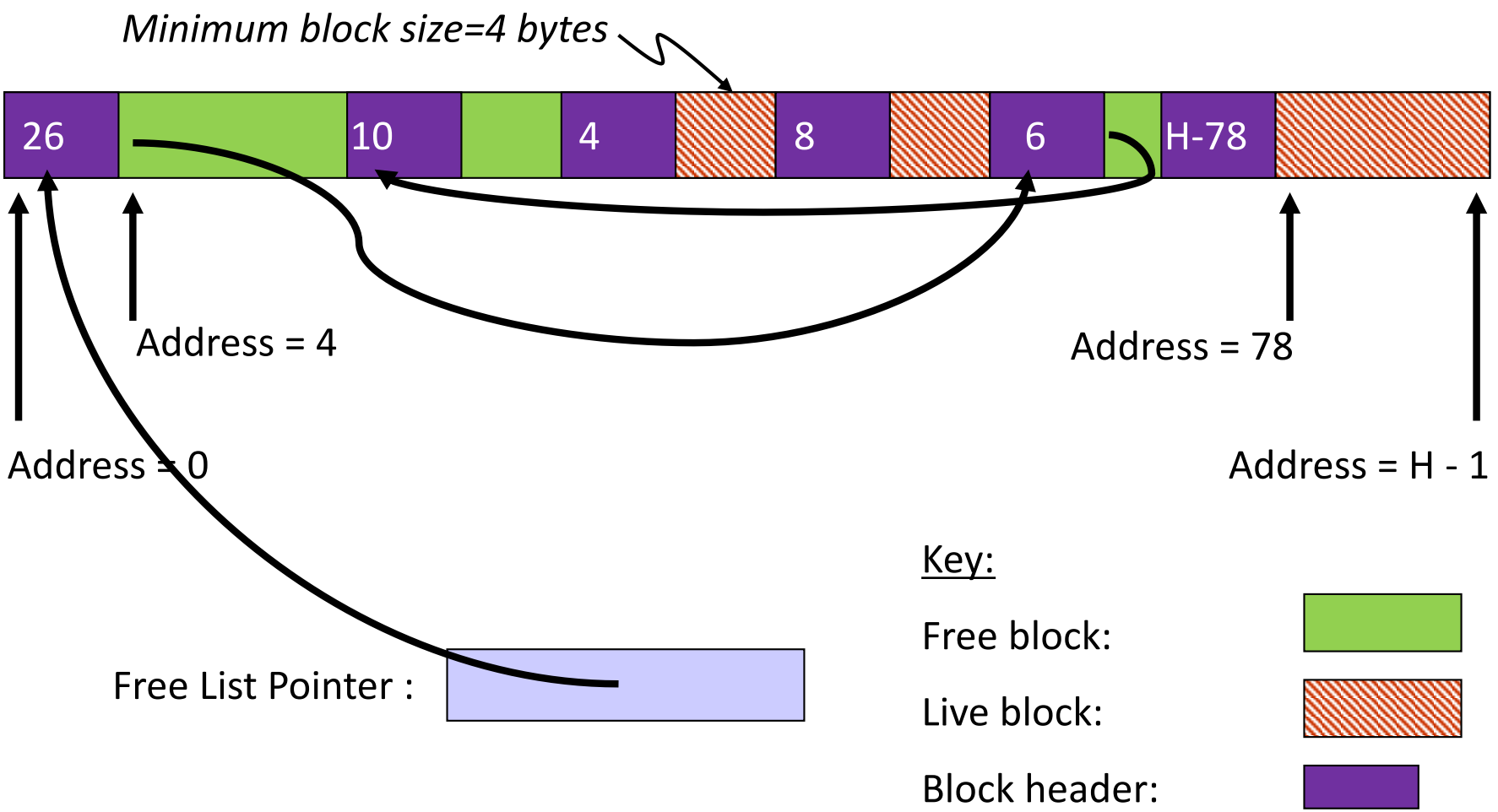
FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

This figure illustrates the **interim** layout of memory:

- after the program has freed the block with header at address=0
- before coalescing has been performed

Although there is a total of 42 bytes free space, the maximum allocatable is 26 bytes

MEMORY MANAGEMENT EXAMPLE

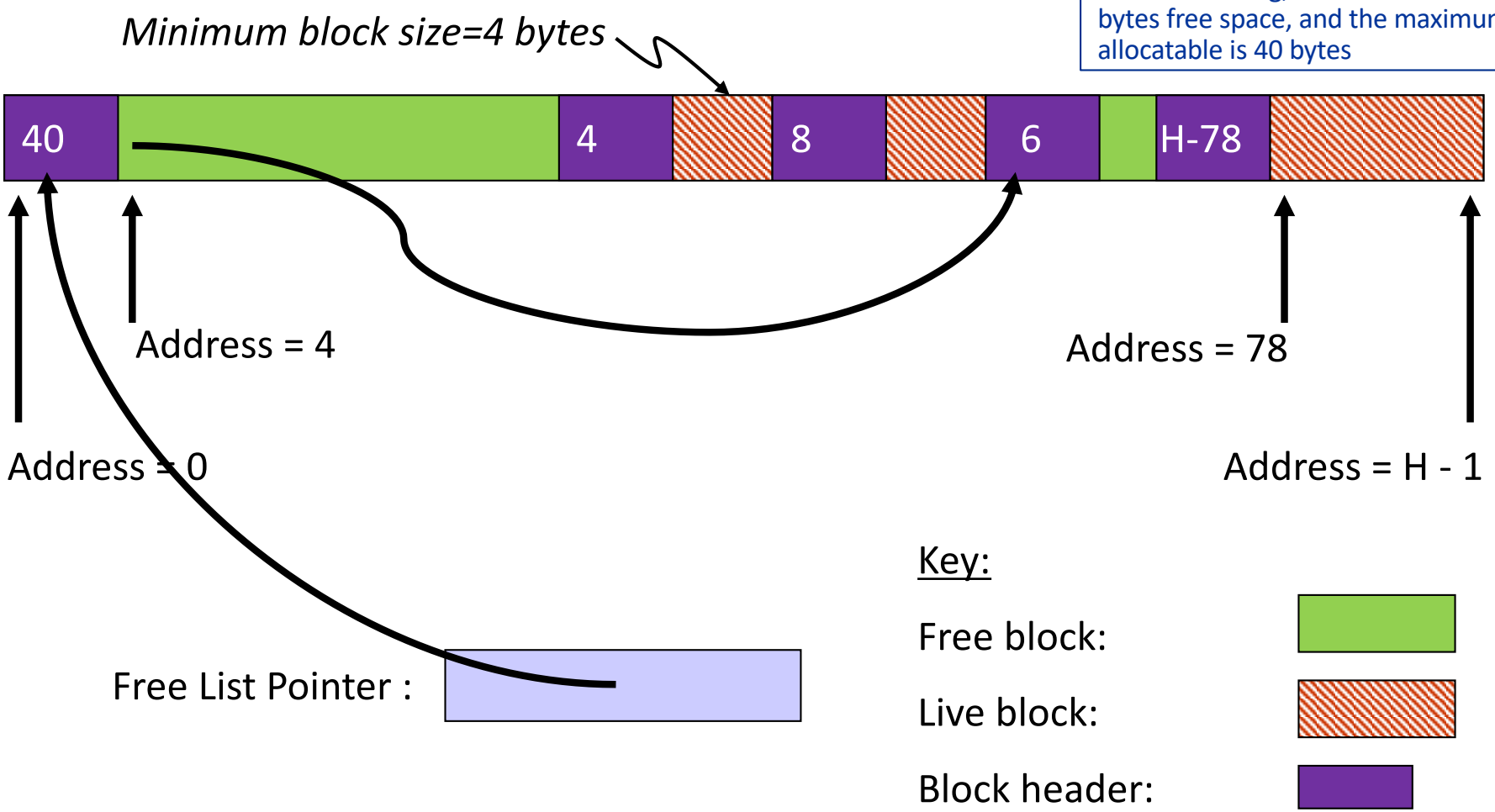


FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

This figure illustrates the **final** layout of memory after coalescing has been performed (following the previous call to *free*). Even without footers, it is possible to coalesce with the next highest block in memory – though resetting the free list pointers will be slow. Also note this is a LIFO example that differs from the AO example on slide 18.

After coalescing, there is a total of 46 bytes free space, and the maximum allocatable is 40 bytes

MEMORY MANAGEMENT EXAMPLE



FUNCTIONAL PROGRAMMING MEMORY ALLOCATION TECHNIQUES

SUMMARY

- Variable-size blocks
- Pointer increment
- Free list – sequential fits
- Segregated free lists
- Double-linked free lists
- Free lists – coalescing and boundary tags
- Memory management example

In summary, this lecture has covered a range of memory allocation techniques in some detail. In particular, the techniques of pointer increment and sequential fits using a free list have been discussed

After introducing segregated free lists and double-linked free lists, the lecture explored coalescing and boundary tags in depth

The lecture then ended with a VERY SIMPLE memory management example of how to administer a heap with a sequence of calls to *malloc* and *free*