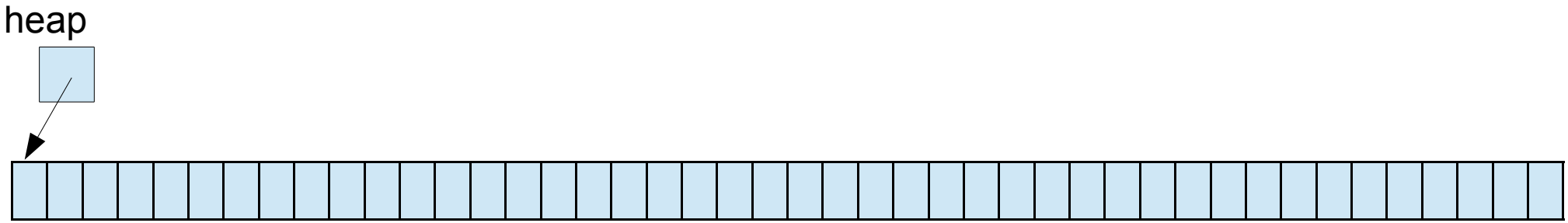
Assignment 4

- In this assignment you will be simulating a heap and dynamic memory allocation.
- Each allocated block of memory will contain
  - A head memory cell (which gives the size of the allocation with respect to how many memory cells we need for data/chars)
  - An address memory cell which says where the memory block above it on the heap is located (0 is none above it)
  - The data memory cells (the number in the head memory cell says how many of these there are)
  - Another memory cell which says where the next memory block below it on the heap is located (use the address just beyond the limit of the heap if there is no memory block below this one)
- Each allocation uses at least 4 memory cells in the heap (head, previous block address, at least 1 data, next block address)

The heap itself will simply be an array of Memory.



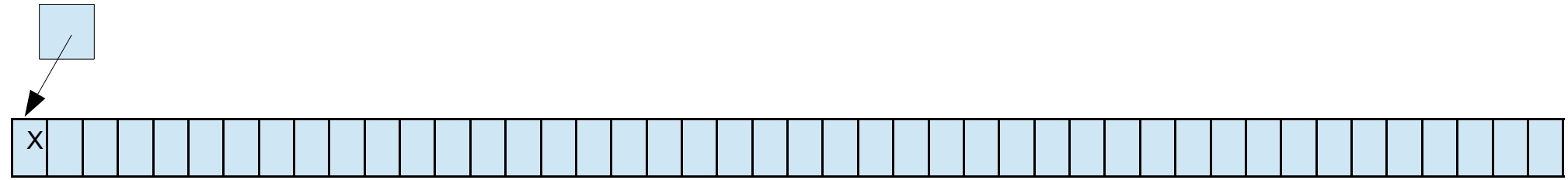
Blanks in the array will denote “free” memory. These are memory cells that Have not been allocated and used for the memory blocks.

The first memory cell is heap[0] and this is memory location zero (0) in the heap. It is off limits to write to. Visually, we'll put an X there (and you should print an X in the dumpHeap() function for it).

When we allocate a block of memory, we'll visually illustrate this by using H for the head node, A for the first address, then one or more Cs to denote the data and finally another A to denote the second address.

The heap itself will simply be an array of Memory.

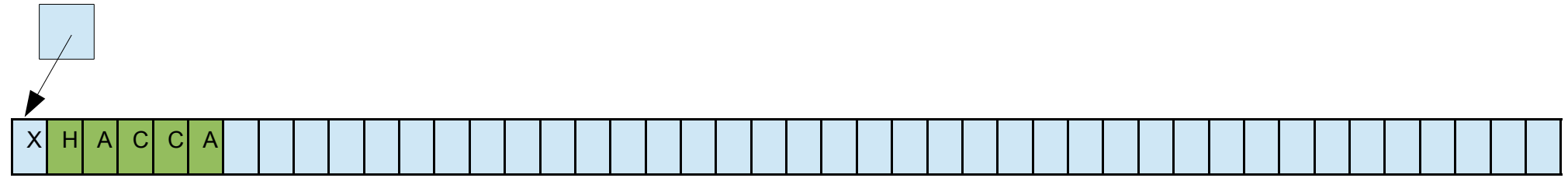
heap




```
Memory * heap = initializeHeap(44); // gives the heap above
```

The heap itself will simply be an array of Memory.

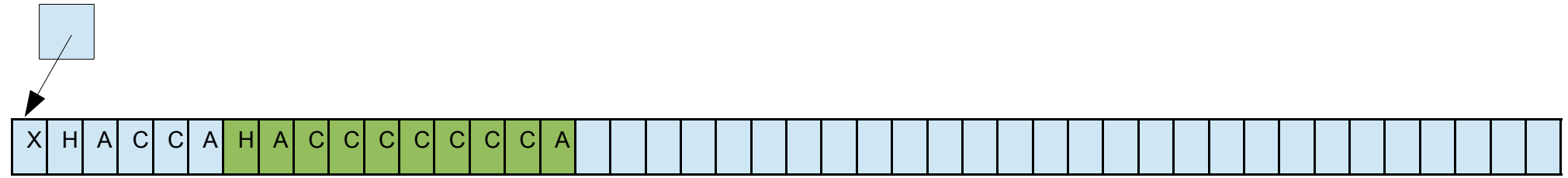
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
```

The heap itself will simply be an array of Memory.

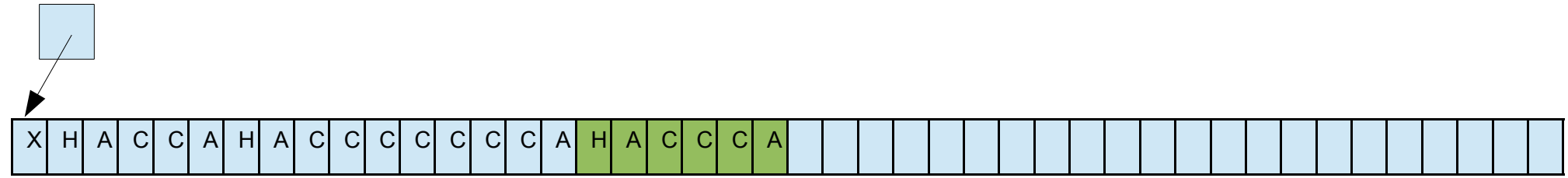
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
```

The heap itself will simply be an array of Memory.

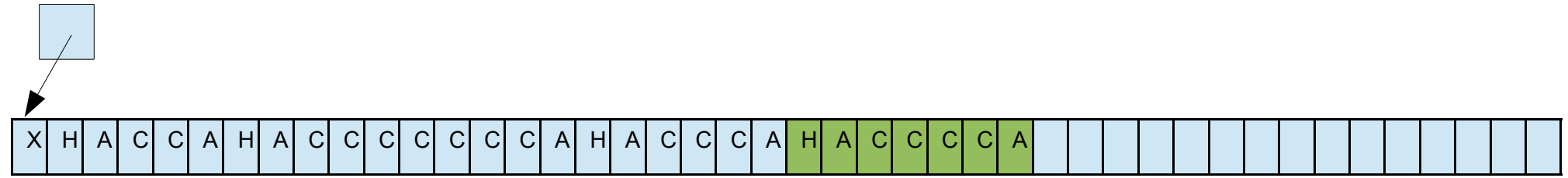
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
```

The heap itself will simply be an array of Memory.

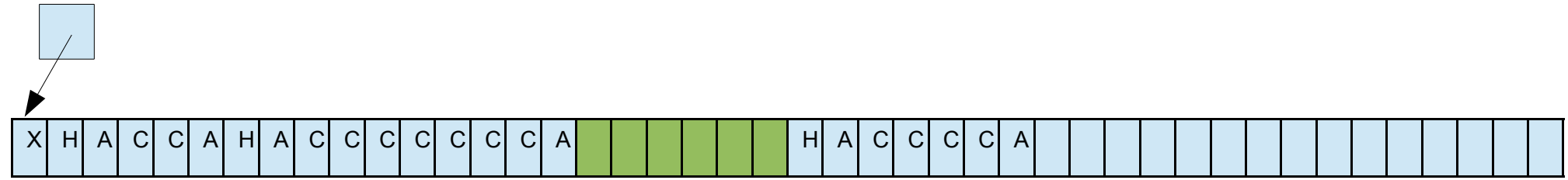
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
```

The heap itself will simply be an array of Memory.

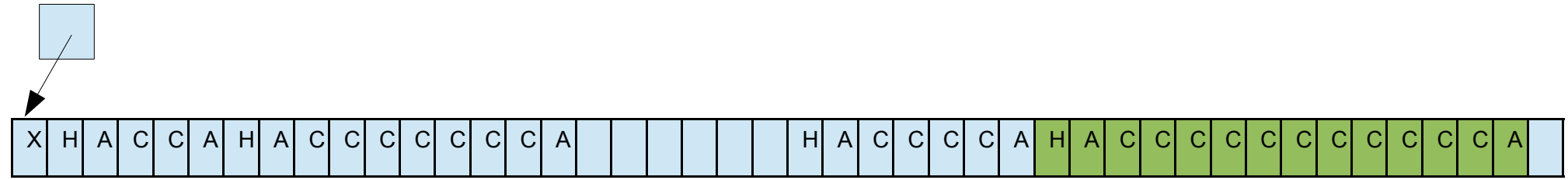
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
```

The heap itself will simply be an array of Memory.

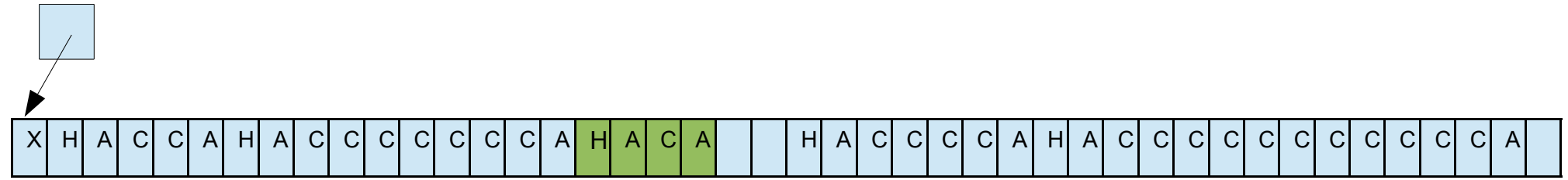
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
c = charMalloc(heap, 11);
```

The heap itself will simply be an array of Memory.

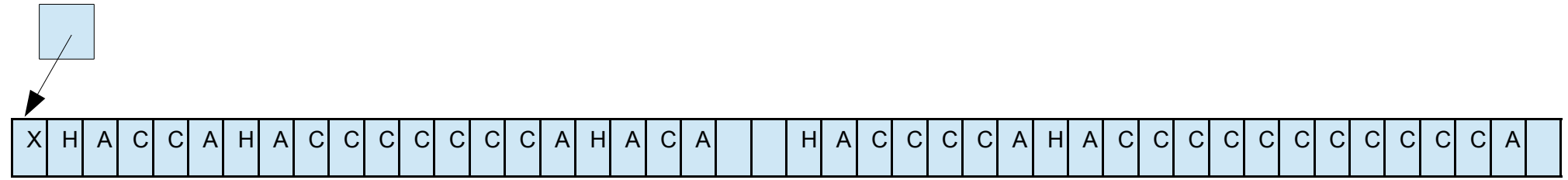
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
c = charMalloc(heap, 11);
charPointer e = charMalloc(heap,1);
```

The heap itself will simply be an array of Memory.

heap

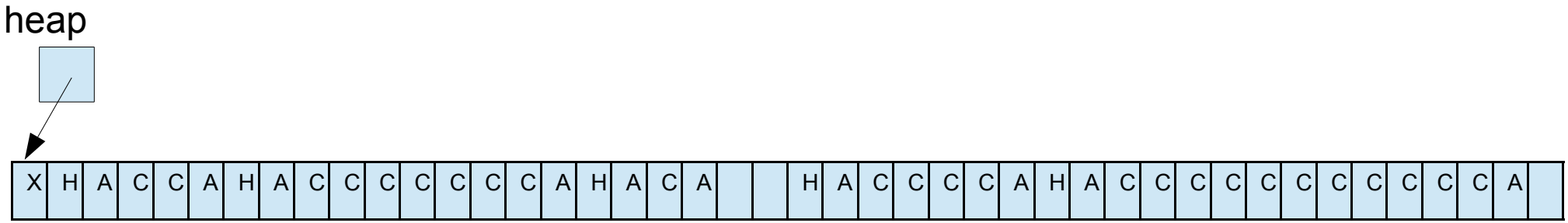


So how do we insert and remove memory block in the heap?

We will always add a new block in the first available space (that can fit the request) after the last allocated block. If the last block allocated has since been freed, then we add after the next most recently allocated block.

You will need to keep track of all the allocations up to this point so you can know where to add the next block. (I.e., think stack; when the top of the stack is an allocation that since been freed, you can remove it from the stack; you do not need to always have every allocation recorded in the stack, remove entries when easy to do so.)

The heap itself will simply be an array of Memory.



There are different ways that you can go about implementing your heap usage (but your dumpHeap() function should display the same output regardless of how you choose to go about it). I'll describe one method here.

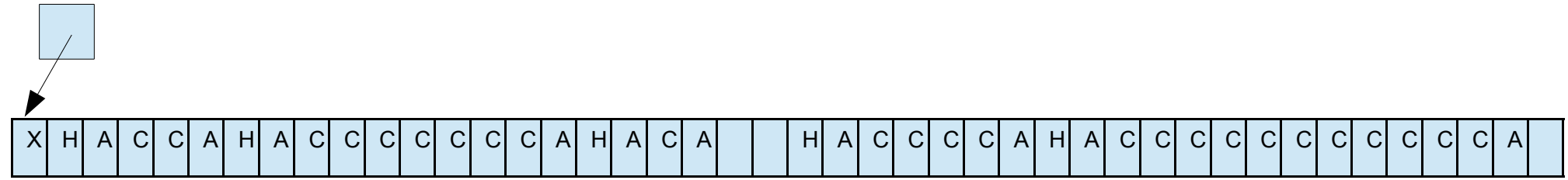
Essentially, you can view your allocated blocks as a doubly linked list in the heap, except that they have fixed positions in memory (in the heap). We'll let the first address memory cell give the location (in the heap array) where the memory block immediately before it lies in the heap and the second address memory cell give the location where the memory block immediately after it lies in the heap.

When there are no memory blocks allocated before a given block, its first address memory cell will point to memory location (heap array index) zero.

When there are no memory blocks allocated after a given block, its second address memory cell will point to just beyond the last valid memory location (heap array index just outside the array)

The heap itself will simply be an array of Memory.

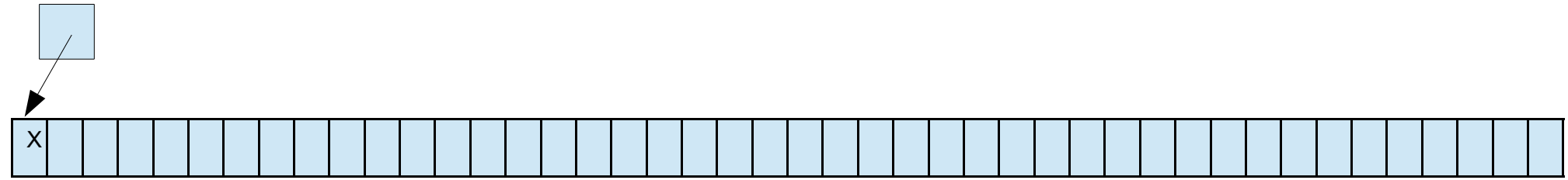
heap



When we are trying to add a new block of allocated memory to the heap, if we get to the end of the heap and cannot find a place to it, we start looking from the start of the heap. We have gone through the entire heap and cannot find a place to allocate the requested block, then we return 0 (the invalid location) to indicate that the charMalloc() call failed.

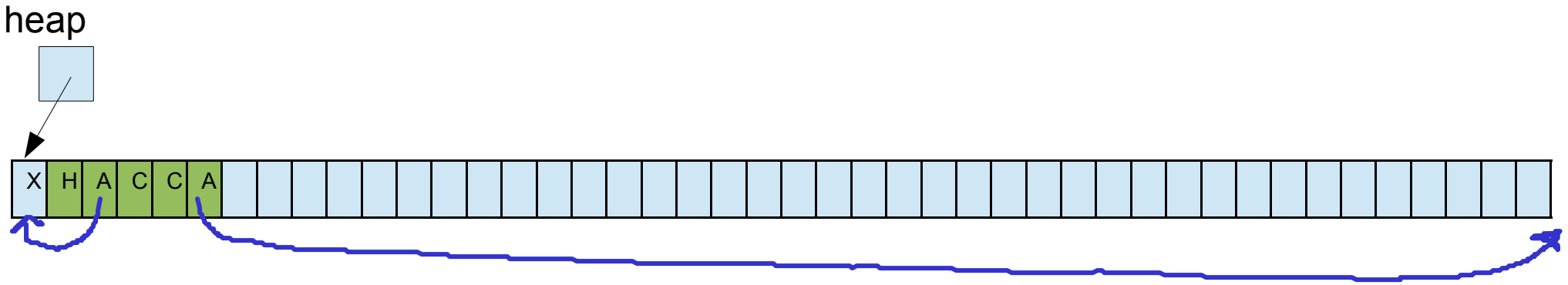
The heap itself will simply be an array of Memory.

heap



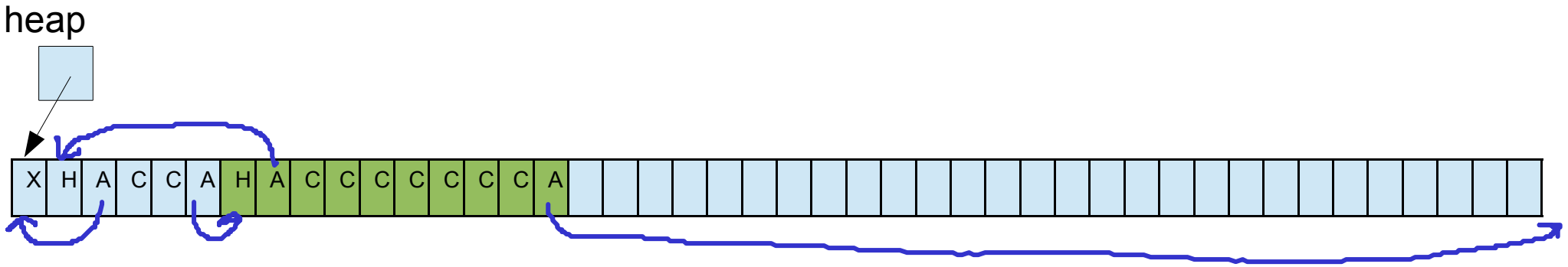

```
Memory * heap = initializeHeap(44); // gives the heap above
```

The heap itself will simply be an array of Memory.




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
```

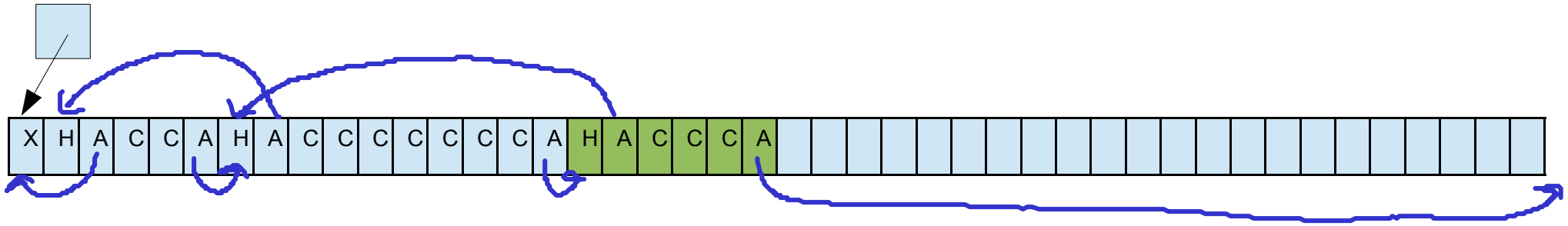
The heap itself will simply be an array of Memory.




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
```

The heap itself will simply be an array of Memory.

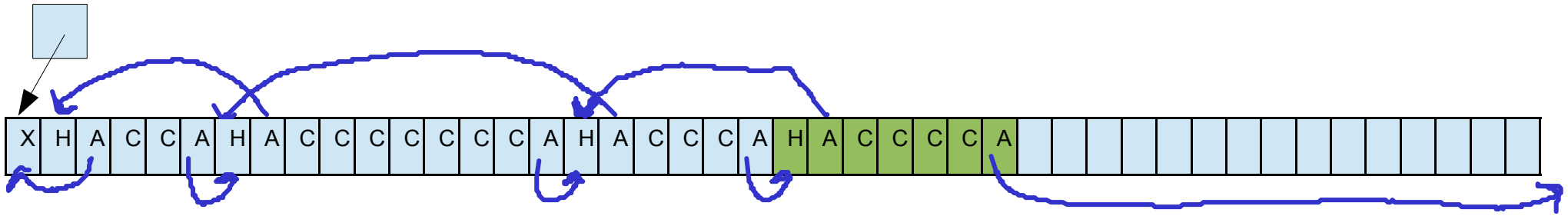
heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
```

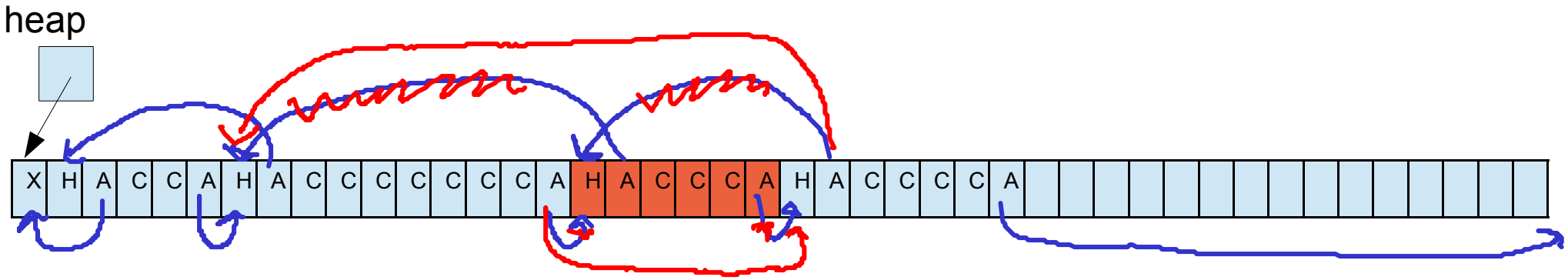
The heap itself will simply be an array of Memory.

heap



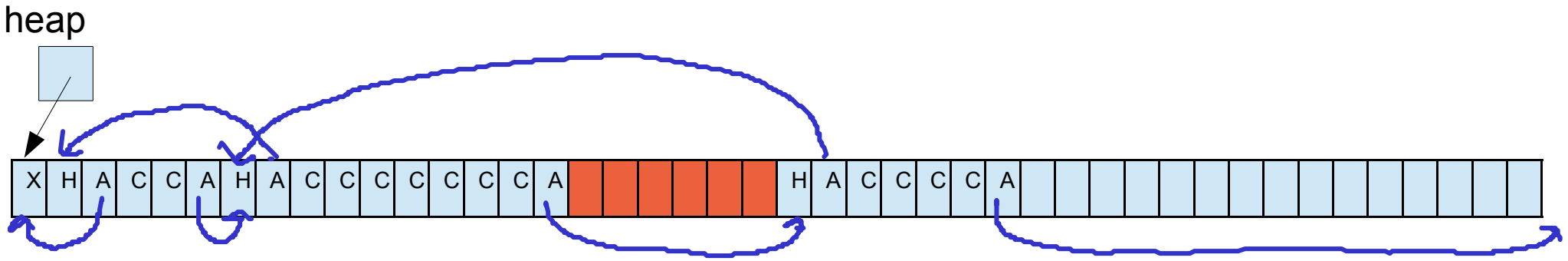

```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
```

The heap itself will simply be an array of Memory.




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
```

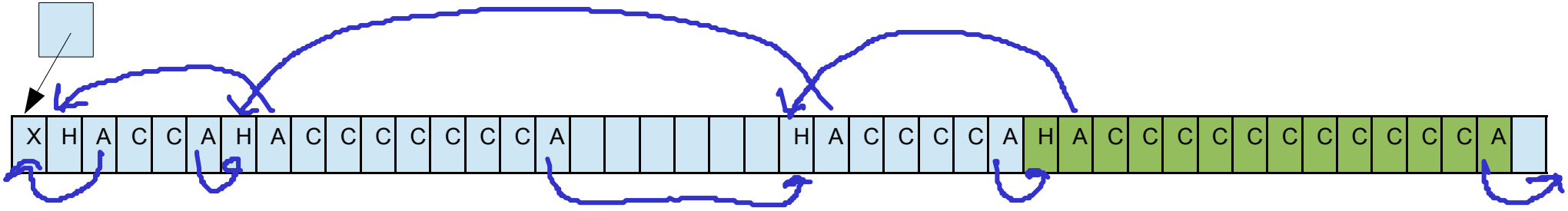
The heap itself will simply be an array of Memory.




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
```

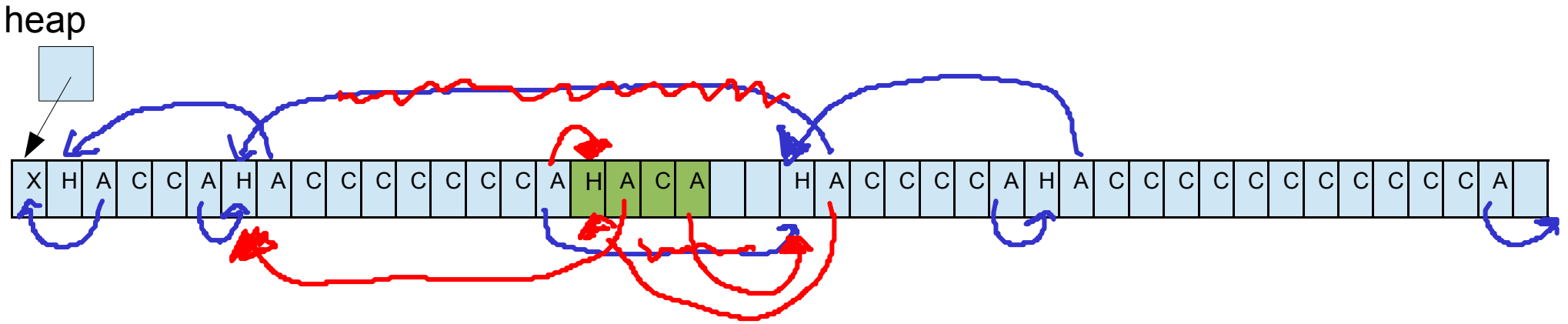
The heap itself will simply be an array of Memory.

heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
c = charMalloc(heap, 11);
```

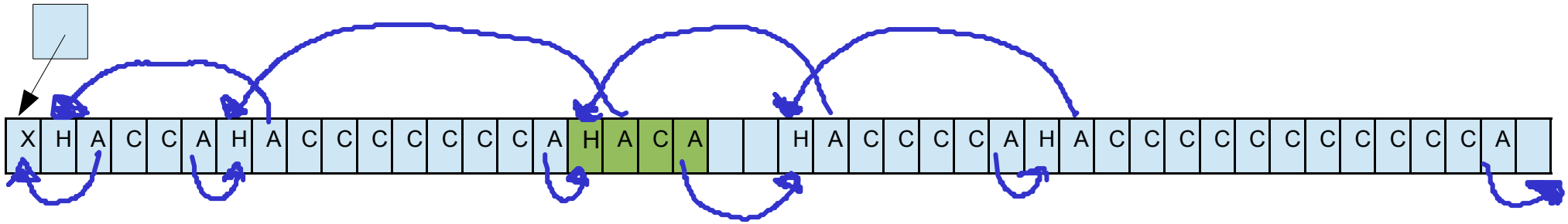
The heap itself will simply be an array of Memory.




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
c = charMalloc(heap, 11);
charPointer e = charMalloc(heap,1);
```

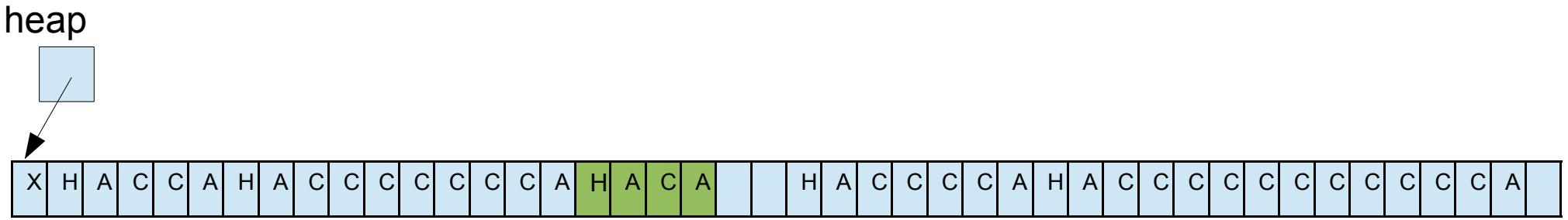
The heap itself will simply be an array of Memory.

heap




```
Memory * heap = initializeHeap(44);   // gives the heap above
charPointer a = charMalloc(heap, 2);  // allocate block of memory
charPointer b = charMalloc(heap, 7);
charPointer c = charMalloc(heap, 3);
charPointer d = charMolloc(heap, 4);
charFree(heap, c);
c = charMalloc(heap, 11);
charPointer e = charMalloc(heap,1);
```

The heap itself will simply be an array of Memory.

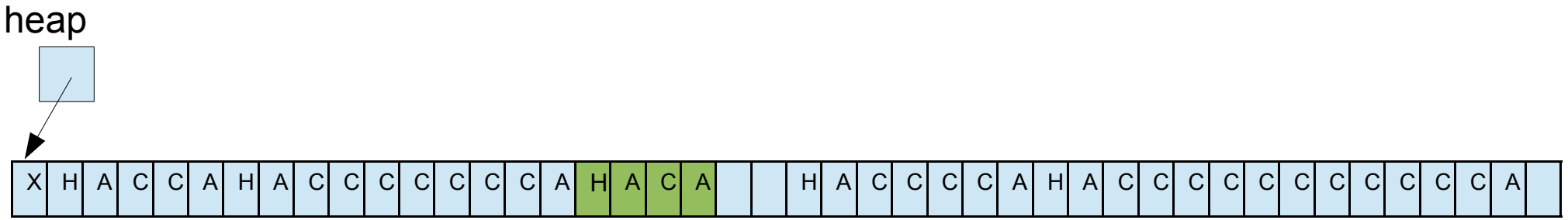


If we are just allocating blocks of memory (without any frees), the process is very Straightforward. We keep adding blocks to the immediate right of the previously allocated block.

Once we can free blocks, it becomes more complicated. Esentially, it is like removing a node from a doubly linked list. You need to make some modifications to the memory blocks immediately before an after the block you are freeing.

The problem becomes a little more complex when we have reached the end of the heap trying to find a place to allocate a block of memory. Once we have wrapped around from the end to the front of the heap, we are essentially adding a node to the interior of a double linked list. Again, we need to modify several allocated blocks of memory to account for this.

The heap itself will simply be an array of Memory.



As mentioned in class (13 March), you will need to keep at least two additional data structures to manage your heap.

1) You'll need some kind of list or table that allows you to keep track of which memory locations have been allocated by a call to charMalloc(). You also need to know if you free this location later.

For this, do not simply make a second array that mirrors the heap and says what each memory cell is (allocated or free; if allocated a head cell, address or data cell). You should only keep the minimal amount of information that is needed.

2) You'll need to keep a stack of malloc requests, so that you know the last block that was allocated, or the one before that (if that was freed), or the one before that one if last two have been freed, etc.