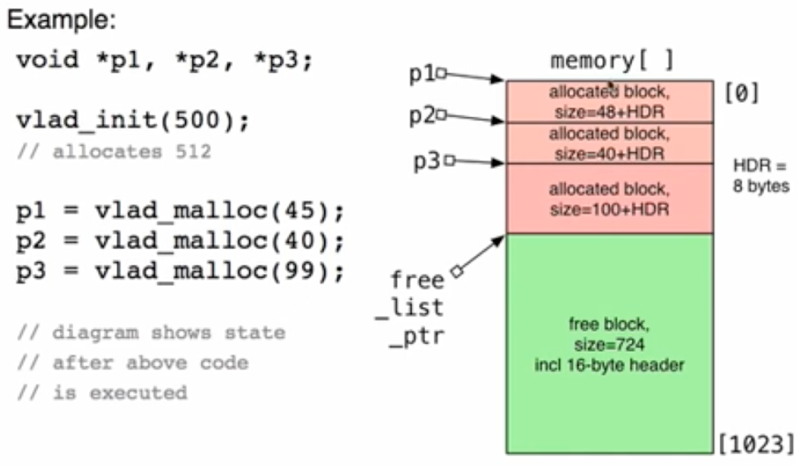
**Assignment 1**

* Implement a memory allocator
* Dynamic data structure
  + You don’t know how big to start with, so you dynamically add data
* Aims of assignment:
  + Learn about how memory is managed
  + Doubly linked lists
  + More practice with C and data structures
* DUE **Monday 29th August at 9:00am**
* What to do?

1. vlad\_init(vsize\_t MAX)

* Initialises allocator
* (size = power of 2, min = 1024)
* If it is not, then round up to power of 2

1. vlad\_malloc(vsite\_t N)

* Reserves block N bytes inside initial allocated  
  chunk, returns address as void \*
* Returns N bytes + Header HDR
* (size = multiple of 4)
* If it is not, then round up to multiple of 4

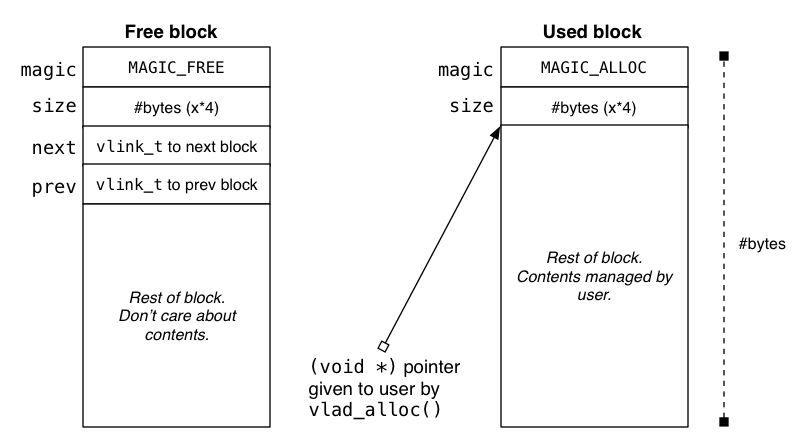
where vsize\_t = unsigned int type

The header tells you that:

* It is an allocated block
* How big the block is

The blocks aren’t AREN’T allocated are in a FREE LIST.

* 1st block in the free list = always the one with the LOWEST MEMORY ADDRESS
  + It is pointer to by the FREE LIST POINTER
* There can be multiple free blocks

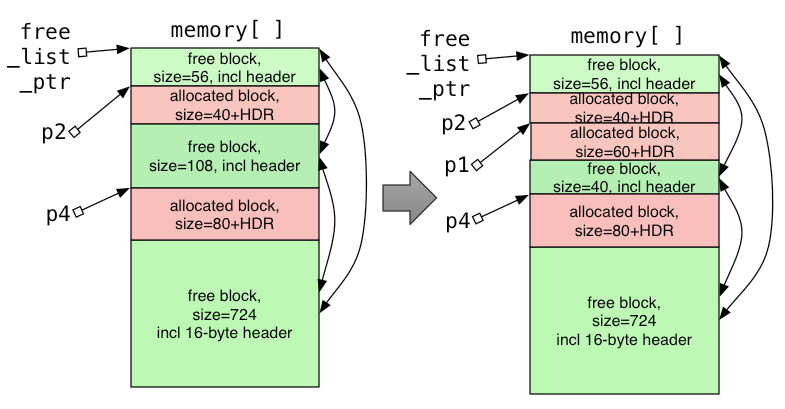


**Vlad\_free function**

* User wants to free up specific allocated memory block, pointed to by pointer P3
* Function will:
  + know where P3 is pointing
  + know how large the block is
  + be able to make the block “look like a free block” and link in to the FREE LIST
  + Because it is newly allocated to the free block, it becomes the new start of the free list (the 1st free block)
  + I.e. New free block will point to old free block, then old free block points back to the new free block.  
    (therefore connecting the free blocks together into a “single block”)
* If user requests for memory size:
  + SOMEWHAT LESS or EQUAL than the available free block, then it can use that specific free block (even if its not exact size).
  + SIGNIFICANTLY LESS, then it will only use partial free block, then link the rest back to the old free block
  + LARGER than the available block, it will use up the older, larger free block.

**According to the project spec, when allocated memory (vlad\_alloc), we should choose the free block that is LARGER and CLOSEST in size** **to use up**.

Diagram: Vlad\_malloc(60) function, before allocation + after allocation



The allocated block size = **user memory alloc request + 8byte header**

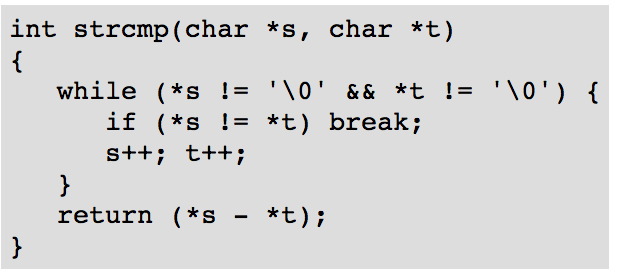
**Vlad\_merge** function

* Instead of having many small fragments of free blocks which would be unusable, the free block should be merged into a larger block of memory ready for use.

Different approaches to allocating:

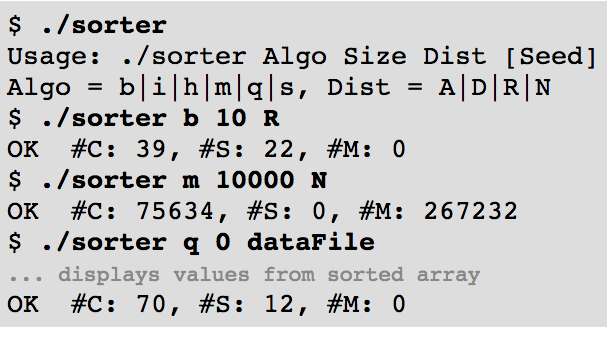
* You can allocate using the largest size block
* You can allocate using the most fit size block
* Any way you want (read the specs though)

Hints:

* **Void \*** … real memory address (C pointer)
* **Byte \*** … real memory address (C pointer)
* **Vaddr\_t** … offset within **memory[]** chunk
  + Just references memory addresses
* **Vlink\_t** … offset within **memory[]** chunk
  + Inside headers with the prev / next nodes are all done with **vlink\_t**
* From outside the ADT, the user only sees **void \*** pointer values
* **Void\***, **vaddr\_t**, **vlink\_t** all refer to locations in **memory[]**
* Need ways to map between **void \*** and **vlink\_t** values.
* In C, we can do arithmetic on pointer values
  + Consider: **Type \*p; … p = p + n;**
  + New value of **p** is calculated as **p + n\*sizeof(Type)**  
    (numeric input x size of type that you’re pointing at)
  + Applies to ints, chars, etc.
    - E.g. Int = 4 bytes / char = 1 byte
  + A use for pointer arithmetic: **string comparison**
    - 
* Another use for pointer arthimetic: **fast array access**
  + In theory, accessing memory via. Pointers than Arrays are faster because:  
    POINTERS: Directly access memory locations  
    ARRAYS: Indices are processed, then memory locations calculated / accessed

**Sortlab Program**

A program that runs sorting algorithms and analyses them. How to use sortLab program:



**Bubble Sort**

How it works

* Starts at one end of array.
* Looks at adjacent elements that are out of order, then swaps
* Keeps moving down the array until it reaches the end.
* Goes back to the start and repeats process until it reaches “sorted section”
* Goes back to start and repeats process until entire array is sorted

Cost Analysis

* Cost of *i*th iteration
  + n – I comparisons, ?? swaps
  + S depends on “sortedness”, best = 0, worst = n – i
* How many iterations? Depends on sortedness
  + Best case: 1 iteration
  + Worst case: n – 1 iterations
* CostBEST = n (sorted)
* CostWORST = n – 1 + … + 1 (reverse sorted)
* **Complexlity = O(n2)**

Adaptability: **BUBBLE SORT IS ADAPTIVE**

**Insertion Sort**

How it works

* Does moves rather than swaps
  + Movement behaviour: **SORTED PORTION | 🡨 UNSORTED PORTION**
* Take the first element and treat it as a sorted array (length 1)
* Take next element along the list, look through the sorted part of array and moves existing elements in the sorted part up, until you reach an element that is smaller than the one you want to insert
* You insert that element into the sorted list (length now = 2)
* Take the next element along the list and again go through the sorted part of array, inserting the element into the correct place while moving the larger elements up the list.
* Repeat until sorted.

Cost Analysis

* Cost for inserting element into sorted list of length l
  + C = ??, depends on sortedness, best = 1, worst = i
  + S = 0 (no swap operations, only moves)
  + M = ??, depends where **value** fits, best = 1, worst = i
* Always has N iterations
* CostBEST = 1 + 1 + … + 1 (sorted)
* CostWORST = 1 + 2 + … + N = N\*(N+1)/2 (reverse)
* **Complexity = O(n2)**

**Quicksort**

How it works

* 2 phases:
  + Partitioning Phase = Have all small elements at the start of the array (not necessarily sorted) and large elements at the end of the array (not necessarily sorted)
  + Merging Phase = Sort the LEFT PARTITION + Sort the RIGHT PARTITION.
* In the PARTITIONING PHASE:
  + We pick a **pivot element X**
  + We have 2 indexes, **i** and **j**
  + (assuming the pivot is the 1st element):

One starts at one end and the other starts at the other end (they move towards each other)

* + As they are moving, as we see an element that is less than Pivot X, we keep moving forward as they are in the correct partition.
  + We stop once we reach an element that is greater than Pivot X.
  + Similar for the top part, we move down the list and stop at an element that is smaller than Pivot X.
  + We swap the two elements over once they have reached the stopping point.
  + We continue the previous operation until **i** and **j** reach somewhere in the middle.
  + The point that they meet = the point where we put the Pivot X into the array (depending on if the point is smaller or larger than Pivot X)
  + The two partitions are now created, with:
    - Unsorted Partition < X
    - Pivot X
    - Unsorted Partition > X

NOTE: This will work horribly if the Pivot X chosen is the SMALLEST ELEMENT, which means it will stop very frequently while the top element goes down the list the whole way.

Cost Analysis

* CaseBEST = O(nlogn) comparisons
  + Choice of pivot gives two equal sized partitions
  + Same happens at every recursive level
  + Each level requires approx. n comparisons
  + Halving at each level 🡪 log2n levels
* CaseWORST = **O(n2)** comparisons
  + Always choose lowest / highest value for pivot
  + Partitions size 1 and n – 1
  + Each level requires approx. n comparisons
  + Partitioning to 1 and n – 1 🡪 n levels

**Mergesort**

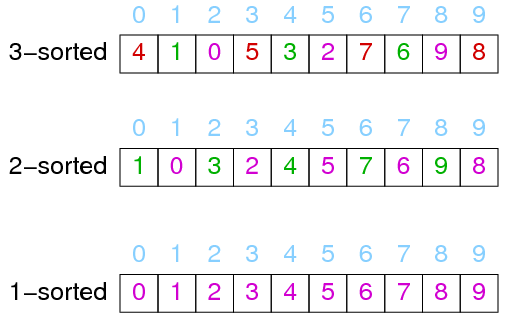
How it works

* Split the array in half, take the midpoint / middle index
* Recursively sort the first half
* Recursively sort the second half
* Take the two sorted halves and put them together into NEW sorted array.
* Compare the first elements of both partitions and then insert the SMALLER one back into the original array
* Repeat the comparisons and merging back into the original list, until:
  + Both lists are empty OR
  + One list is empty and another list still has a few elements left. You can append the rest of these elements to the end of the original list (as they are already sorted).

Cost Analysis

* CaseBEST = O(nlogn) comparisons
  + Split array into equal-sized portions
  + Same happens at every recursive level
  + Each “level” requires ≤N item comparisons
  + Halving at each level 🡪 Log2N levels
* CaseWORST = **O(nlogn)** comparisons
  + Partitions are exactly interleaved
  + Need to compare all the way to end of partitions
* Disadvantage over quicksort: need extra storage O(N)

**Shell Sort**

How it works (H-Sorted Arrays)

* Multiple passes doing multiple long-distance  
  insertion sorts. Example:
* 1st iteration = 3-sorted
  + Every 3 elements are in sorted order
* 2nd iteration = 2-sorted
  + Every 2 elements are in sorted order
* 3rd iteration = 1-sorted
  + Every element is in sorted order
* For each value of h, we do multiple INSERTION SORTS

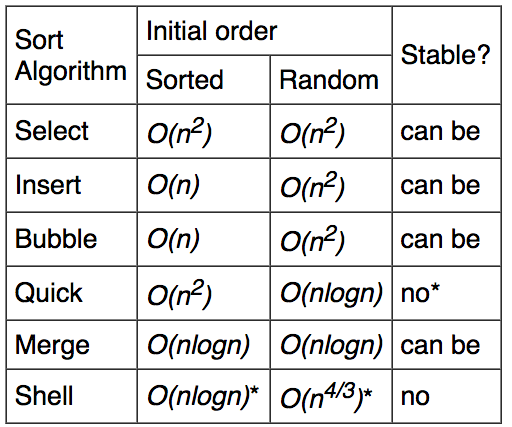
Questions

* How to choose the starting h?
* How much to reduce h on each iteration?

Efficiency of Shell Sort

* Depends on the sequence of *h* values
* Shell Sort has still not yet been fully analysed, therefore we don’t know its cost complexity
* **Knuth** suggests **O(N3/2) e.g. O(N^1.5)** comparisons for the above sequence

**Summary of Sorting Algorithms**



* Merge Sort is arguably the best sorting algorithm + stable.
* Most algorithms are stable, but some are inherently unstable  
  because of the way they shift things around.
  + E.g. Shell Sort:  
    As part of the h-sorting process, there is no guarantee  
    that the two items with the same key will end up in  
    the same order. (as they may be separated due to  
    the sorting keys, then moved back into an incorrect  
    relative position)
  + E.g. Quick Sort:  
    With partioning, relative positioning can become  
    incorrect after swaps.
* “Can be” stable = algorithms can be stable, as long as you  
  only stick with using **<** **(less than)** or **> (greater than)** rather  
  than less / greater than and equal to.
* The **choice of comparison test** generally affects the stability of an algorithm.

**Sorting in GENERAL**

It is simple to think of sorting as using integers in an array. However, sorting items can be:

* **Structs** with multiple values associated with it, where **one of those values are a** **unique** **key field**.

**Structs: Writing Comparison Functions**

Example: Struct with data = int ID, char NAME, int PROGRAM.

Write comparison functions that could be used to:

* Sort by **ID** only
* Sort by **Name** only
* Sort by **Program**, then by **name**

See **sortCompare.c** file

How to make use of these functions?

* You have a sorting function

**External Sorting**

Previous sorts assume

* **Item a[n]** , efficient access to **a[i]**
* Which suggests that data is in arrays IN memory
* Which limits sortable data to what fits IN memory

Data in disk = sequential access

Data in memory = can be random access

When data is in disk files

* Random access is inefficient (files are sequential access)
* But max data size is far less constrained (much larger data size on disks compared to in memory)

**EXTRA NOTE: If you want to pass a pointer to a function in C, you can simply write the name of the function.**

* **C understands what we’re talking about.**
* **No need to write it as:** *function(par1,par2,&functionName)*  
  🡪 The ampersand in front of functionName you want to use
* **As soon as you write a function with parameters, it assumes you’re making a function call.**

MergeSort

* A sequential sort
* Always going to have two pointers, adjacent to each other.
* 1st pass = simply merge adjacent elements (sorting 2 elts)
  + If 1st > 2nd , it writes 1st elt to output file
  + If 2nd> 1st, it writes 2nd elt to output file
* **We end up with an array with sorted pairs in it**
  + **A whole sequence of sorted pairs**
  + **Maybe the last pair only has one elt (if array = odd length)**
* In the next pass, we scan through array again.
  + We look at the 1st pair and the 2nd pair and we do a merge on those.