Providing References and Pointers in OCaml

- The module of type `Allocator` is called `DLLAllocator` in the file. The type `heap` is a record of four arrays: one array of type `ptr`, one for string values, one for integer values and one bool array, called `trackptr` for keeping track of which positions in the `ptr` array are free.

  The type `ptr` is of `int * int option`, where the first entry in the tuple can be from 0 to 3: where 0 does not refer to any array in heap, 1 indicates that entry contains some address from the `ptr` array, 2 indicates there is some address from the integers array and 3 indicates the strings array. The second entry stores the indices (addresses) as `int option`. The reason why I added `int option` here is to tell apart the null pointer. So each heap has one null pointer where nothing can be assigned. So `Some n` tells you that there might be an index stored while `None` is only for the null pointer.

  Now, I specified the null pointer to always be the first entry in the `ptr` array, and hence the first value of `trackptr` would always be `false` too. This is the reason why when I make a heap, the first value in the pointers array is null and rest are initialised by containing the address of this null pointer. Similarly the first entry in the trackptr array is `false` indicating the spot is not available and initializes the rest of them as `false`.

  The `alloc` functions takes in a heap and an integer `n` and looks if there are `n` consecutive available entries in the heap. It finds the entries in the track array and returns the corresponding entry from the pointers array. Conversely, the `free` function takes in a pointer and an integer `n`, obtains the index of the pointer, goes to the corresponding entry in track array and frees up (converts into true) the next n entries.

  The `deref_as` functions exploits the fact that the first value of each entry in the pointers indicates which array to look in. So based on whether the value is `1, 2` or `3`, it goes to the relevant array and take out the value stored in the index. These functions make sure that if and only if the first value is `1`, it would look for the address in the pointers arrays. If the argument pointer does not refer to the array of the correct type, we get an error.

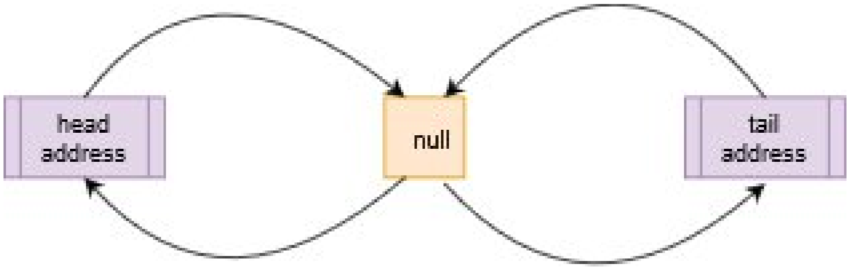
  The `assign_as` function take a pointer offset and value to be stored. The offset tells us in which position to store the address and the element in the relevant array. If the index of the inputted pointer is `p` and the offset is `o` then the address of the item would be stored in the index `p + o` of the pointers array.

- The module functor `Double Linked List` takes in a module of type `Allocator` as a parameter. Each `dll_node` in the module is of type `ptr`. Now, we know that for doubly linked lists, we need to store the value, information of the next and previous node in each node.

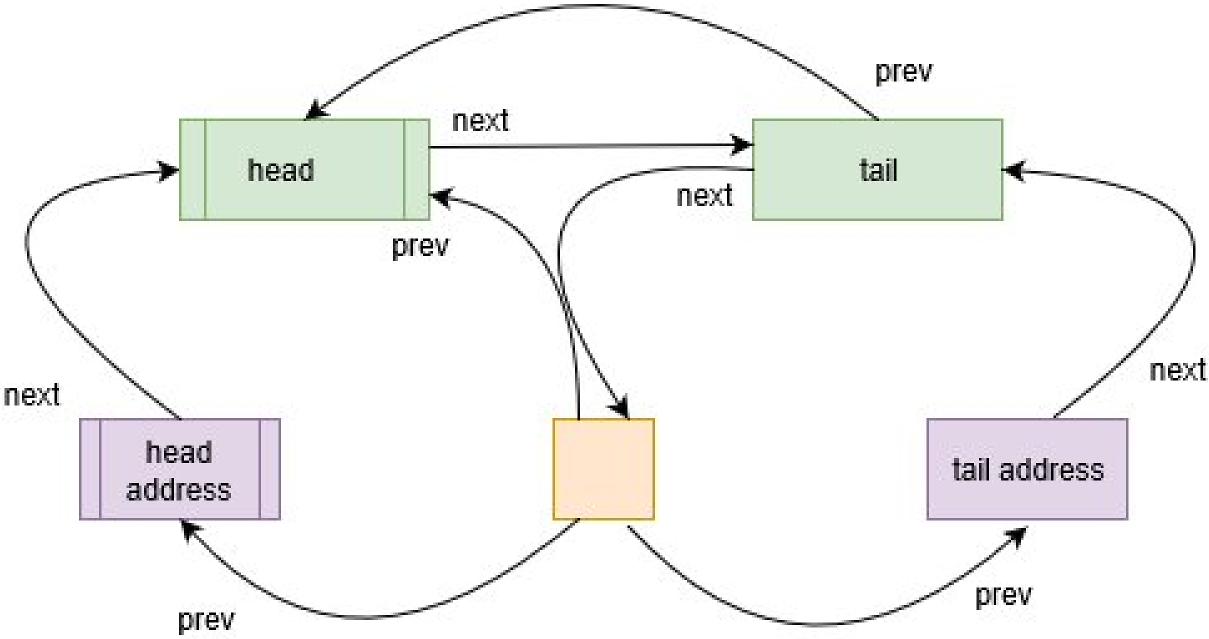
  The `mk_node` function allocates us a segment of size 4 in the heap, assigns the integer to first entry of segment (which has type pointer), string to the second and assigns the null pointer to the third and fourth entry which shows the node does not have any other nodes as a previous node or next node yet. The same format is obeyed by the `int_value`, `string_value`, `prev` and `next` functions. The `remove` functions frees up the space in the heap that was being used by the node and updates the prev and next values of the nodes its previous and next nodes (if any).

  The `print_from_node` function prints the entries represented by the node `n` and then by the node `next n` and so forth all the way to the final node (which has null node as its next node. The function prints all nodes as array of tuples, where each tuple contains the integer and string values of each node.

- Finally, the `Queue` module has the signature of normal queues which have entries of type `int * string`. The `mk_queue` function creates a queue which is of type record, the record consisting of a heap, a `dll_node` for head and a `dll_node` for the tail. So this make function creates a heap and creates two nodes to store the addresses for tail and head node. The `prev` node for these two nodes is at all times the null pointer. But the `next` value of these two nodes keeps changing as these values point to the head and tail of the queue respectively when the queue is not empty. Here is what the the empty queue looks like:



And here is what a queue with two entries look like. Since, there is only two entries, one is head and the next is tail.



The `enqueue` and `dequeue` functions easily deal with tuples of `int * string` and behave the way as expexted. They update the values `prev`, `next`, head and tail where necessary.

Finally, the `queue_to_list` function starts from the head of the queue, and to all the way to the tail, picks the integer and string values of each node and stores them as tuples in a list.