Queues Queues are an essential data structure that are found in vast amounts of software from user mode to kernel mode applications that are core to the system. Fundamentally they honour a first in first out (FIFO) strategy, that is the item first put into the queue will be the first served, the second item added to the queue will be the second to be served and so on. A traditional queue only allows you to access the item at the front of the queue; when you add an item to the queue that item is placed at the back of the queue. Historically queues always have the following three core methods: Enqueue: places an item at the back of the queue; Dequeue: retrieves the item at the front of the queue, and removes it from the queue; Peek: 1 retrieves the item at the front of the queue without removing it from the queue As an example to demonstrate the behaviour of a queue we will walk through a scenario whereby we invoke each of the previously mentioned methods observing the mutations upon the queue data structure. The following list describes the operations performed upon the queue in Figure 6.1: 1. Enqueue(10) 2. Enqueue(12) 3. Enqueue(9) 4. Enqueue(8) 5. Enqueue(3) 6. Dequeue() 7. Peek() 1This operation is sometimes referred to as Front 48 CHAPTER 6. QUEUES 49 8. Enqueue(33) 9. Peek() 10. Dequeue() 6.1 A standard queue A queue is implicitly like that described prior to this section. In DSA we don't provide a standard queue because queues are so popular and such a core data structure that you will find pretty much every mainstream library provides a queue data structure that you can use with your language of choice. In this section we will discuss how you can, if required, implement an efficient queue data structure. The main property of a queue is that we have access to the item at the front of the queue. The queue data structure can be efficiently implemented using a singly linked list (defined in §2.1). A singly linked list provides O(1) insertion and deletion run time complexities. The reason we have an O(1) run time complexity for deletion is because we only ever remove items from the front of queues (with the Dequeue operation). Since we always have a pointer to the item at the head of a singly linked list, removal is simply a case of returning the value of the old head node, and then modifying the head pointer to be the next node of the old head node. The run time complexity for searching a queue remains the same as that of a singly linked list: O(n). 6.2 Priority Queue Unlike a standard queue where items are ordered in terms of who arrived first, a priority queue determines the order of its items by using a form of custom comparer to see which item has the highest priority. Other than the items in a priority queue being ordered by priority it remains the same as a normal queue: you can only access the item at the front of the queue. A sensible implementation of a priority queue is to use a heap data structure (defined in §4). Using a heap we can look at the first item in the queue by simply returning the item at index 0 within the heap array. A heap provides us with the ability to construct a priority queue where the items with the highest priority are either those with the smallest value, or those with the largest. 6.3 Double Ended Queue Unlike the queues we have talked about previously in this chapter a double ended queue allows you to access the items at both the front, and back of the queue. A double ended queue is commonly known as a deque which is the name we will here on in refer to it as. A deque applies no prioritization strategy to its items like a priority queue does, items are added in order to either the front of back of the deque. The former properties of the deque are denoted by the programmer utilising the data structures exposed interface. CHAPTER 6. QUEUES 50 Figure 6.1: Queue mutations CHAPTER 6. QUEUES 51 Deque's provide front and back specific versions of common queue operations, e.g. you may want to enqueue an item to the front of the queue rather than the back in which case you would use a method with a name along the lines of EnqueueFront. The following list identifies operations that are commonly supported by deque's: • EnqueueFront • EnqueueBack • DequeueFront • DequeueBack • PeekFront • PeekBack Figure 6.2 shows a deque after the invocation of the following methods (inorder): 1. EnqueueBack(12) 2. EnqueueFront(1) 3. EnqueueBack(23) 4. EnqueueFront(908) 5. DequeueFront() 6.

DequeueBack() The operations have a one-to-one translation in terms of behaviour with those of a normal queue, or priority queue. In some cases the set of algorithms that add an item to the back of the deque may be named as they are with normal queues, e.g. EnqueueBack may simply be called Enqueue an so on. Some frameworks also specify explicit behaviour's that data structures must adhere to. This is certainly the case in .NET where most collections implement an interface which requires the data structure to expose a standard Add method. In such a scenario you can safely assume that the Add method will simply enqueue an item to the back of the deque. With respect to algorithmic run time complexities a deque is the same as a normal queue. That is enqueueing an item to the back of a the queue is O(1), additionally enqueuing an item to the front of the queue is also an O(1) operation. A deque is a wrapper data structure that uses either an array, or a doubly linked list. Using an array as the backing data structure would require the programmer to be explicit about the size of the array up front, this would provide an obvious advantage if the programmer could deterministically state the maximum number of items the deque would contain at any one time. Unfortunately in most cases this doesn't hold, as a result the backing array will inherently incur the expense of invoking a resizing algorithm which would most likely be an O(n) operation. Such an approach would also leave the library developer CHAPTER 6. QUEUES 52 Figure 6.2: Deque data structure after several mutations CHAPTER 6. QUEUES 53 to look at array minimization techniques as well, it could be that after several invocations of the resizing algorithm and various mutations on the deque later that we have an array taking up a considerable amount of memory yet we are only using a few small percentage of that memory. An algorithm described would also be O(n) yet its invocation would be harder to gauge strategically. To bypass all the aforementioned issues a deque typically uses a doubly linked list as its baking data structure. While a node that has two pointers consumes more memory than its array item counterpart it makes redundant the need for expensive resizing algorithms as the data structure increases in size dynamically. With a language that targets a garbage collected virtual machine memory reclamation is an opaque process as the nodes that are no longer referenced become unreachable and are thus marked for collection upon the next invocation of the garbage collection algorithm. With C++ or any other language that uses explicit memory allocation and deallocation it will be up to the programmer to decide when the memory that stores the object can be freed.