Chapter 2

Linked Lists

Linked lists can be thought of from a high level perspective as being a series of nodes. Each node has at least a single pointer to the next node, and in the last node's case a null pointer representing that there are no more nodes in the linked list.

In DSA our implementations of linked lists always maintain head and tail pointers so that insertion at either the head or tail of the list is a constant time operation. Random insertion is excluded from this and will be a linear operation. As such, linked lists in DSA have the following characteristics:

1. Insertion is $O(1)$
2. Deletion is $O(n)$
3. Searching is $O(n)$

Out of the three operations the one that stands out is that of insertion. In DSA we chose to always maintain pointers (or more aptly references) to the node(s) at the head and tail of the linked list and so performing a traditional insertion to either the front or back of the linked list is an $O(1)$ operation. An exception to this rule is performing an insertion before a node that is neither the head nor tail in a singly linked list. When the node we are inserting before is somewhere in the middle of the linked list (known as random insertion) the complexity is $O(n)$. In order to add before the designated node we need to traverse the linked list to find that node's current predecessor. This traversal yields an $O(n)$ run time.

This data structure is trivial, but linked lists have a few key points which at times make them very attractive:

1. the list is dynamically resized, thus it incurs no copy penalty like an array or vector would eventually incur; and
2. insertion is $O(1)$.

2.1 Singly Linked List

Singly linked lists are one of the most primitive data structures you will find in this book. Each node that makes up a singly linked list consists of a value, and a reference to the next node (if any) in the list.

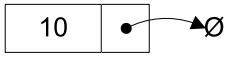

Figure 2.1: Singly linked list node

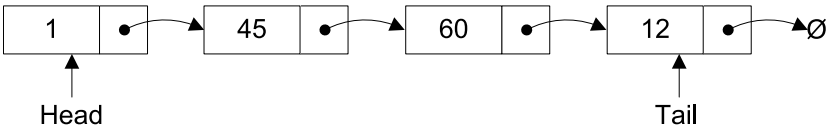

Figure 2.2: A singly linked list populated with integers

2.1.1 Insertion

In general when people talk about insertion with respect to linked lists of any form they implicitly refer to the adding of a node to the tail of the list. When you use an API like that of DSA and you see a general purpose method that adds a node to the list, you can assume that you are adding the node to the tail of the list not the head.

Adding a node to a singly linked list has only two cases:

1. $head = \emptyset$ in which case the node we are adding is now both the *head* and *tail* of the list; or

2. we simply need to append our node onto the end of the list updating the *tail* reference appropriately.

```
1) algorithm Add(value)
2)     Pre: value is the value to add to the list
3)     Post: value has been placed at the tail of the list
4)     n ← node(value)
5)     if head = ∅
6)         head ← n
7)         tail ← n
8)     else
9)         tail.Next ← n
10)        tail ← n
11)    end if
12) end Add
```

As an example of the previous algorithm consider adding the following sequence of integers to the list: 1, 45, 60, and 12, the resulting list is that of Figure 2.2.

2.1.2 Searching

Searching a linked list is straightforward: we simply traverse the list checking the value we are looking for with the value of each node in the linked list. The algorithm listed in this section is very similar to that used for traversal in §2.1.4.

```
1) algorithm Contains(head, value)
2)     Pre:  head is the head node in the list
3)           value is the value to search for
4)     Post: the item is either in the linked list, true; otherwise false
5)     n ← head
6)     while n ≠ ∅ and n.Value ≠ value
7)         n ← n.Next
8)     end while
9)     if n = ∅
10)        return false
11)    end if
12)    return true
13) end Contains
```

2.1.3 Deletion

Deleting a node from a linked list is straightforward but there are a few cases we need to account for:

1. the list is empty; or
2. the node to remove is the only node in the linked list; or
3. we are removing the head node; or
4. we are removing the tail node; or
5. the node to remove is somewhere in between the head and tail; or
6. the item to remove doesn't exist in the linked list

The algorithm whose cases we have described will remove a node from anywhere within a list irrespective of whether the node is the *head* etc. If you know that items will only ever be removed from the *head* or *tail* of the list then you can create much more concise algorithms. In the case of always removing from the front of the linked list deletion becomes an $O(1)$ operation.

```
1) algorithm Remove(head, value)
2)     Pre: head is the head node in the list
3)          value is the value to remove from the list
4)     Post: value is removed from the list, true; otherwise false
5)     if head = ∅
6)         // case 1
7)         return false
8)     end if
9)     n ← head
10)    if n.Value = value
11)        if head = tail
12)            // case 2
13)            head ← ∅
14)            tail ← ∅
15)        else
16)            // case 3
17)            head ← head.Next
18)        end if
19)        return true
20)    end if
21)    while n.Next ≠ ∅ and n.Next.Value ≠ value
22)        n ← n.Next
23)    end while
24)    if n.Next ≠ ∅
25)        if n.Next = tail
26)            // case 4
27)            tail ← n
28)        end if
29)        // this is only case 5 if the conditional on line 25 was false
30)        n.Next ← n.Next.Next
31)        return true
32)    end if
33)    // case 6
34)    return false
35) end Remove
```

2.1.4 Traversing the list

Traversing a singly linked list is the same as that of traversing a doubly linked list (defined in §2.2). You start at the head of the list and continue until you come across a node that is ∅. The two cases are as follows:

1. $node = \emptyset$, we have exhausted all nodes in the linked list; or
2. we must update the *node* reference to be *node*.Next.

The algorithm described is a very simple one that makes use of a simple *while* loop to check the first case.

```
1) algorithm Traverse(head)
2)     Pre: head is the head node in the list
3)     Post: the items in the list have been traversed
4)     n ← head
5)     while n ≠ 0
6)         yield n.Value
7)         n ← n.Next
8)     end while
9) end Traverse
```

2.1.5 Traversing the list in reverse order

Traversing a singly linked list in a forward manner (i.e. left to right) is simple as demonstrated in §2.1.4. However, what if we wanted to traverse the nodes in the linked list in reverse order for some reason? The algorithm to perform such a traversal is very simple, and just like demonstrated in §2.1.3 we will need to acquire a reference to the predecessor of a node, even though the fundamental characteristics of the nodes that make up a singly linked list make this an expensive operation. For each node, finding its predecessor is an $O(n)$ operation, so over the course of traversing the whole list backwards the cost becomes $O(n^2)$.

Figure 2.3 depicts the following algorithm being applied to a linked list with the integers 5, 10, 1, and 40.

```
1) algorithm ReverseTraversal(head, tail)
2)     Pre: head and tail belong to the same list
3)     Post: the items in the list have been traversed in reverse order
4)     if tail ≠ ∅
5)         curr ← tail
6)         while curr ≠ head
7)             prev ← head
8)             while prev.Next ≠ curr
9)                 prev ← prev.Next
10)            end while
11)            yield curr.Value
12)            curr ← prev
13)        end while
14)        yield curr.Value
15)    end if
16) end ReverseTraversal
```

This algorithm is only of real interest when we are using singly linked lists, as you will soon see that doubly linked lists (defined in §2.2) make reverse list traversal simple and efficient, as shown in §2.2.3.

2.2 Doubly Linked List

Doubly linked lists are very similar to singly linked lists. The only difference is that each node has a reference to both the next and previous nodes in the list.

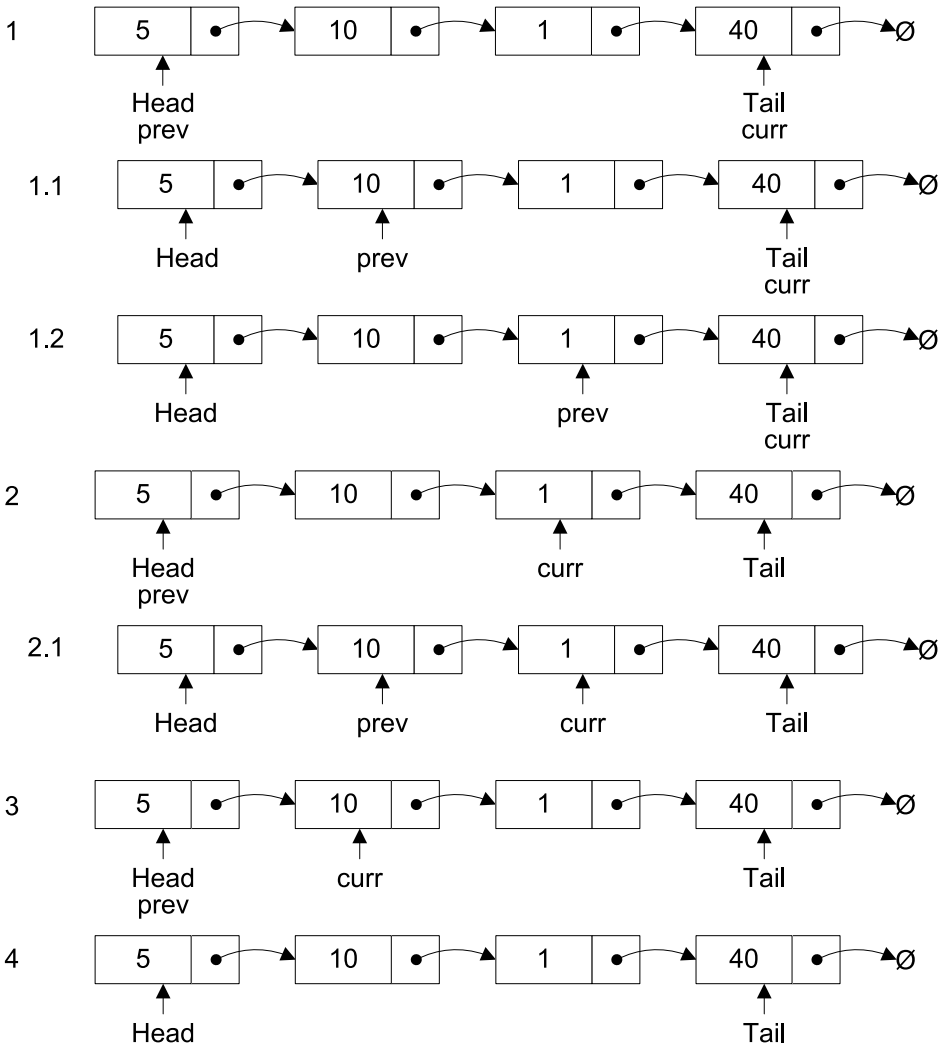

Figure 2.3: Reverse traveral of a singly linked list

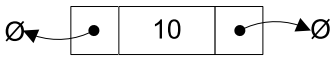

Figure 2.4: Doubly linked list node

The following algorithms for the doubly linked list are exactly the same as those listed previously for the singly linked list:

1. Searching (defined in §2.1.2)
2. Traversal (defined in §2.1.4)

2.2.1 Insertion

The only major difference between the algorithm in §2.1.1 is that we need to remember to bind the previous pointer of n to the previous tail node if n was not the first node to be inserted into the list.

```
1) algorithm Add(value)
2)     Pre: value is the value to add to the list
3)     Post: value has been placed at the tail of the list
4)     n ← node(value)
5)     if head = ∅
6)         head ← n
7)         tail ← n
8)     else
9)         n.Previous ← tail
10)        tail.Next ← n
11)        tail ← n
12)    end if
13) end Add
```

Figure 2.5 shows the doubly linked list after adding the sequence of integers defined in §2.1.1.

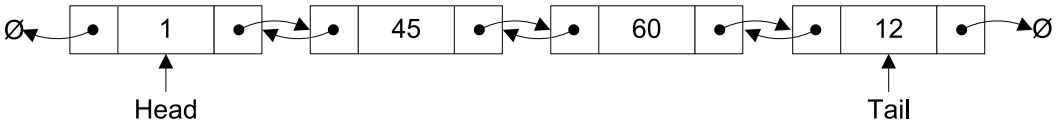

Figure 2.5: Doubly linked list populated with integers

2.2.2 Deletion

As you may of guessed the cases that we use for deletion in a doubly linked list are exactly the same as those defined in §2.1.3. Like insertion we have the added task of binding an additional reference (*Previous*) to the correct value.

```
1) algorithm Remove(head, value)
2)     Pre: head is the head node in the list
3)          value is the value to remove from the list
4)     Post: value is removed from the list, true; otherwise false
5)     if head = ∅
6)         return false
7)     end if
8)     if value = head.Value
9)         if head = tail
10)            head ← ∅
11)            tail ← ∅
12)        else
13)            head ← head.Next
14)            head.Previous ← ∅
15)        end if
16)        return true
17)    end if
18)    n ← head.Next
19)    while n ≠ ∅ and value ≠ n.Value
20)        n ← n.Next
21)    end while
22)    if n = tail
23)        tail ← tail.Previous
24)        tail.Next ← ∅
25)        return true
26)    else if n ≠ ∅
27)        n.Previous.Next ← n.Next
28)        n.Next.Previous ← n.Previous
29)        return true
30)    end if
31)    return false
32) end Remove
```

2.2.3 Reverse Traversal

Singly linked lists have a forward only design, which is why the reverse traversal algorithm defined in §2.1.5 required some creative invention. Doubly linked lists make reverse traversal as simple as forward traversal (defined in §2.1.4) except that we start at the tail node and update the pointers in the opposite direction. Figure 2.6 shows the reverse traversal algorithm in action.

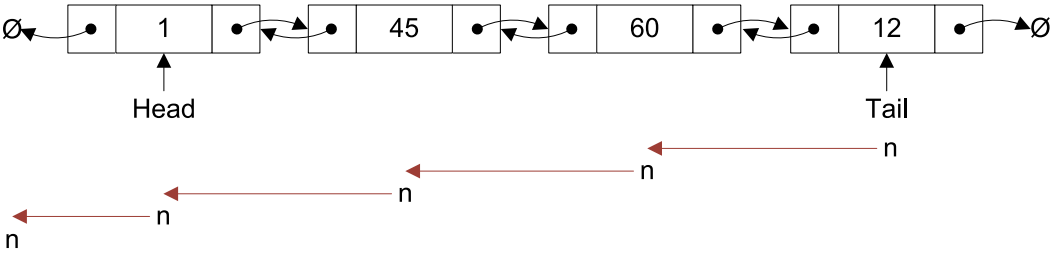

Figure 2.6: Doubly linked list reverse traversal

```
1) algorithm ReverseTraversal(tail)
2)     Pre: tail is the tail node of the list to traverse
3)     Post: the list has been traversed in reverse order
4)     n ← tail
5)     while n ≠ ∅
6)         yield n.Value
7)         n ← n.Previous
8)     end while
9) end ReverseTraversal
```

2.3 Summary

Linked lists are good to use when you have an unknown number of items to store. Using a data structure like an array would require you to specify the size up front; exceeding that size involves invoking a resizing algorithm which has a linear run time. You should also use linked lists when you will only remove nodes at either the head or tail of the list to maintain a constant run time. This requires maintaining pointers to the nodes at the head and tail of the list but the memory overhead will pay for itself if this is an operation you will be performing many times.

What linked lists are not very good for is random insertion, accessing nodes by index, and searching. At the expense of a little memory (in most cases 4 bytes would suffice), and a few more read/writes you could maintain a *count* variable that tracks how many items are contained in the list so that accessing such a primitive property is a constant operation - you just need to update *count* during the insertion and deletion algorithms.

Singly linked lists should be used when you are only performing basic insertions. In general doubly linked lists are more accommodating for non-trivial operations on a linked list.

We recommend the use of a doubly linked list when you require forwards and backwards traversal. For the most cases this requirement is present. For example, consider a token stream that you want to parse in a recursive descent fashion. Sometimes you will have to backtrack in order to create the correct parse tree. In this scenario a doubly linked list is best as its design makes bi-directional traversal much simpler and quicker than that of a singly linked

Chapter 6

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()

[1] This operation is sometimes referred to as Front

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.

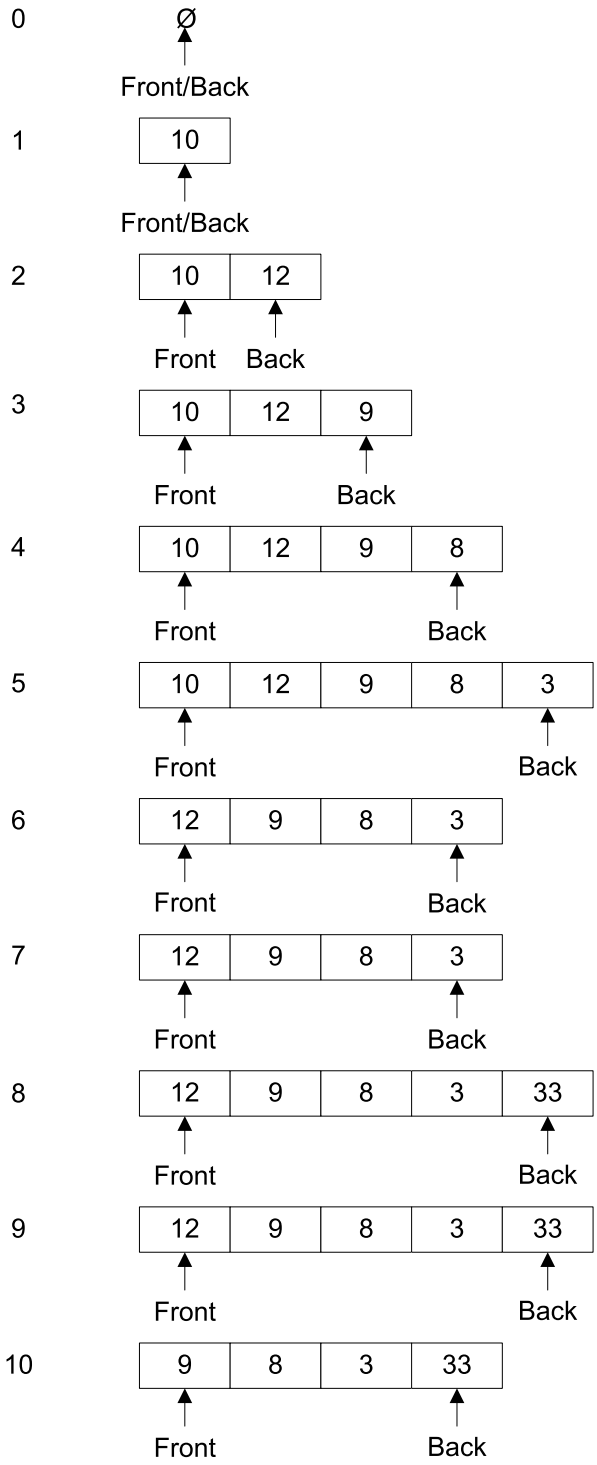

Figure 6.1: Queue mutations

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 (in-order):

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

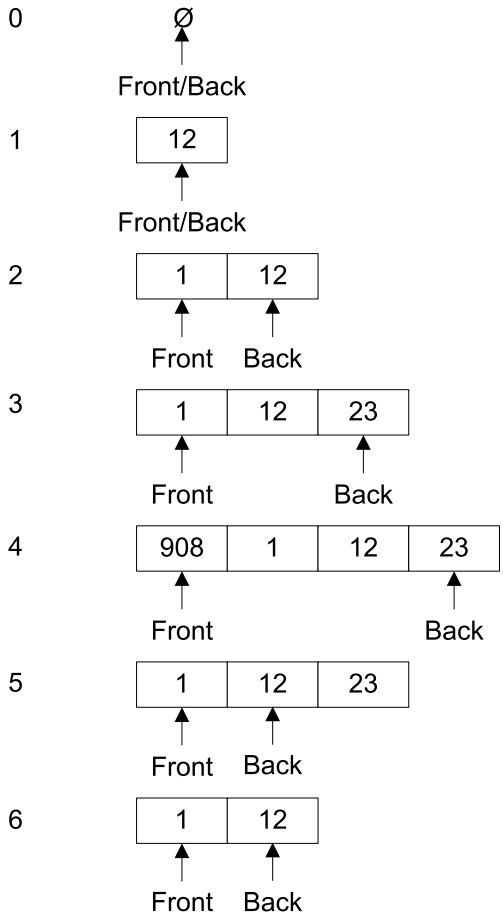

Figure 6.2: Deque data structure after several mutations

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.

6.4 Summary

With normal queues we have seen that those who arrive first are dealt with first; that is they are dealt with in a first-in-first-out (FIFO) order. Queues can be ever so useful; for example the Windows CPU scheduler uses a different queue for each priority of process to determine which should be the next process to utilise the CPU for a specified time quantum. Normal queues have constant insertion and deletion run times. Searching a queue is fairly unusual—typically you are only interested in the item at the front of the queue. Despite that, searching is usually exposed on queues and typically the run time is linear.

In this chapter we have also seen priority queues where those at the front of the queue have the highest priority and those near the back have the lowest. One implementation of a priority queue is to use a heap data structure as its backing store, so the run times for insertion, deletion, and searching are the same as those for a heap (defined in §4).

Queues are a very natural data structure, and while they are fairly primitive they can make many problems a lot simpler. For example the breadth first search defined in §3.7.4 makes extensive use of queues.