Chapter 4

Heap

A heap can be thought of as a simple tree data structure, however a heap usually employs one of two strategies:

1. min heap; or
2. max heap

Each strategy determines the properties of the tree and its values. If you were to choose the min heap strategy then each parent node would have a value that is $\leq$ than its children. For example, the node at the root of the tree will have the smallest value in the tree. The opposite is true for the max heap strategy. In this book you should assume that a heap employs the min heap strategy unless otherwise stated.

Unlike other tree data structures like the one defined in §3 a heap is generally implemented as an array rather than a series of nodes which each have references to other nodes. The nodes are conceptually the same, however, having at most two children. Figure 4.1 shows how the tree (not a heap data structure) (12 7(3 2) 6(9)) would be represented as an array. The array in Figure 4.1 is a result of simply adding values in a top-to-bottom, left-to-right fashion. Figure 4.2 shows arrows to the direct left and right child of each value in the array.

This chapter is very much centred around the notion of representing a tree as an array and because this property is key to understanding this chapter Figure 4.3 shows a step by step process to represent a tree data structure as an array. In Figure 4.3 you can assume that the default capacity of our array is eight.

Using just an array is often not sufficient as we have to be up front about the size of the array to use for the heap. Often the run time behaviour of a program can be unpredictable when it comes to the size of its internal data structures, so we need to choose a more dynamic data structure that contains the following properties:

1. we can specify an initial size of the array for scenarios where we know the upper storage limit required; and
2. the data structure encapsulates resizing algorithms to grow the array as required at run time

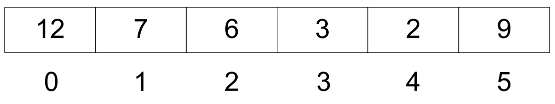

Figure 4.1: Array representation of a simple tree data structure

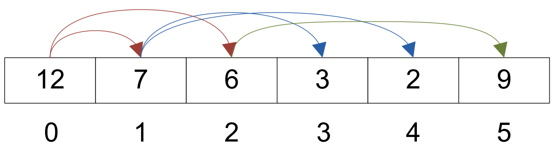

Figure 4.2: Direct children of the nodes in an array representation of a tree data structure

1. Vector
2. ArrayList
3. List

Figure 4.1 does not specify how we would handle adding null references to the heap. This varies from case to case; sometimes null values are prohibited entirely; in other cases we may treat them as being smaller than any non-null value, or indeed greater than any non-null value. You will have to resolve this ambiguity yourself having studied your requirements. For the sake of clarity we will avoid the issue by prohibiting null values.

Because we are using an array we need some way to calculate the index of a parent node, and the children of a node. The required expressions for this are defined as follows for a node at *index*:

1. $(index - 1)/2$ (parent index)
2. $2 * index + 1$ (left child)
3. $2 * index + 2$ (right child)

In Figure 4.4 a) represents the calculation of the right child of 12 $(2 * 0 + 2)$; and b) calculates the index of the parent of 3 $((3 - 1)/2)$.

4.1 Insertion

Designing an algorithm for heap insertion is simple, but we must ensure that heap order is preserved after each insertion. Generally this is a post-insertion operation. Inserting a value into the next free slot in an array is simple: we just need to keep track of the next free index in the array as a counter, and increment it after each insertion. Inserting our value into the heap is the first part of the algorithm; the second is validating heap order. In the case of min-heap ordering this requires us to swap the values of a parent and its child if the value of the child is $<$ the value of its parent. We must do this for each subtree containing the value we just inserted.

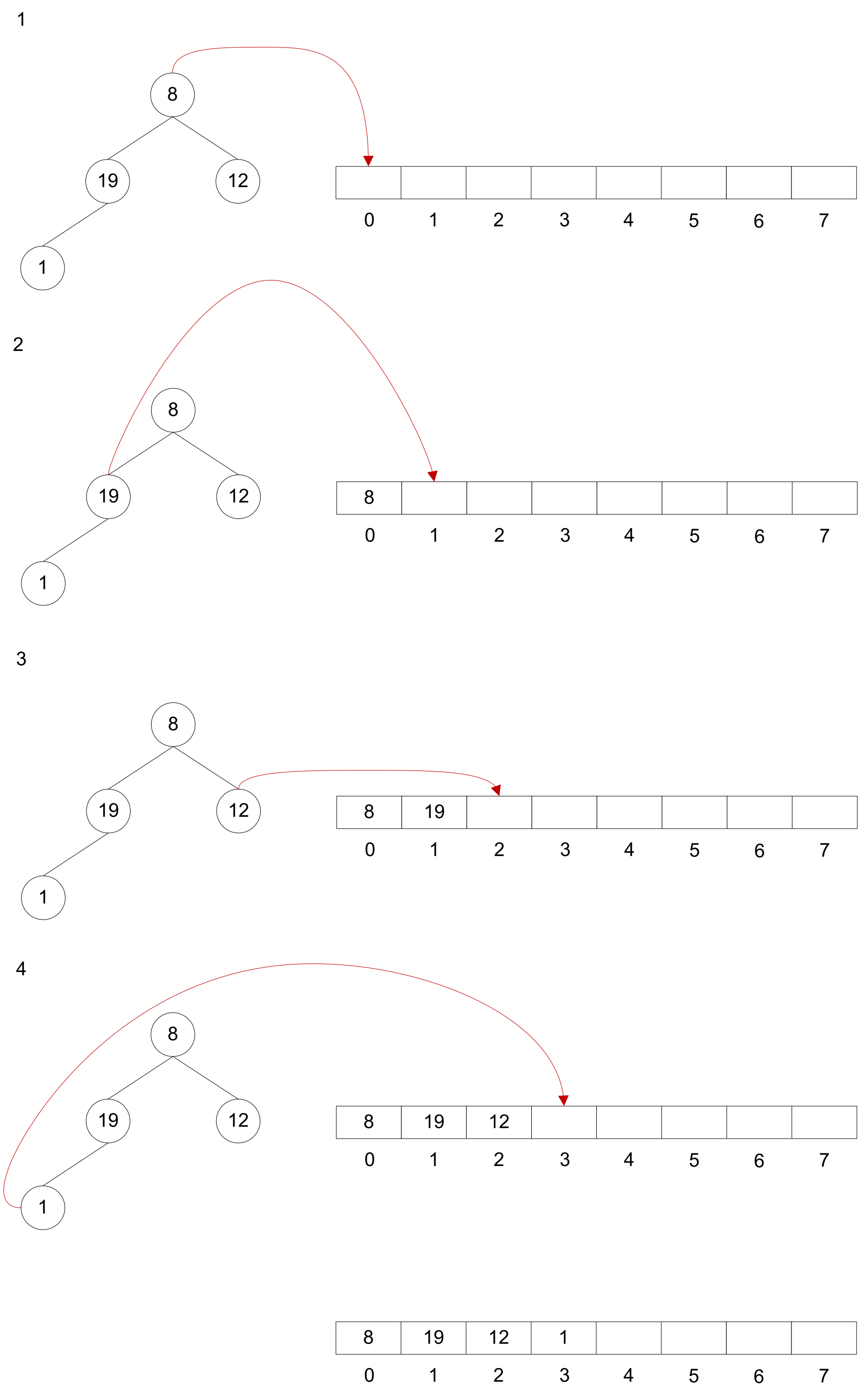

Figure 4.3: Converting a tree data structure to its array counterpart

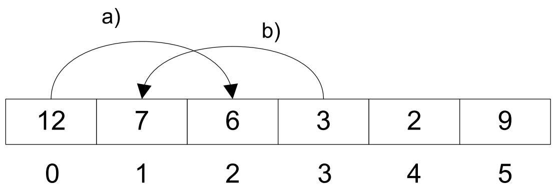

Figure 4.4: Calculating node properties

The run time efficiency for heap insertion is $O(log\ n)$. The run time is a by product of verifying heap order as the first part of the algorithm (the actual insertion into the array) is $O(1)$.

Figure 4.5 shows the steps of inserting the values 3, 9, 12, 7, and 1 into a min-heap.

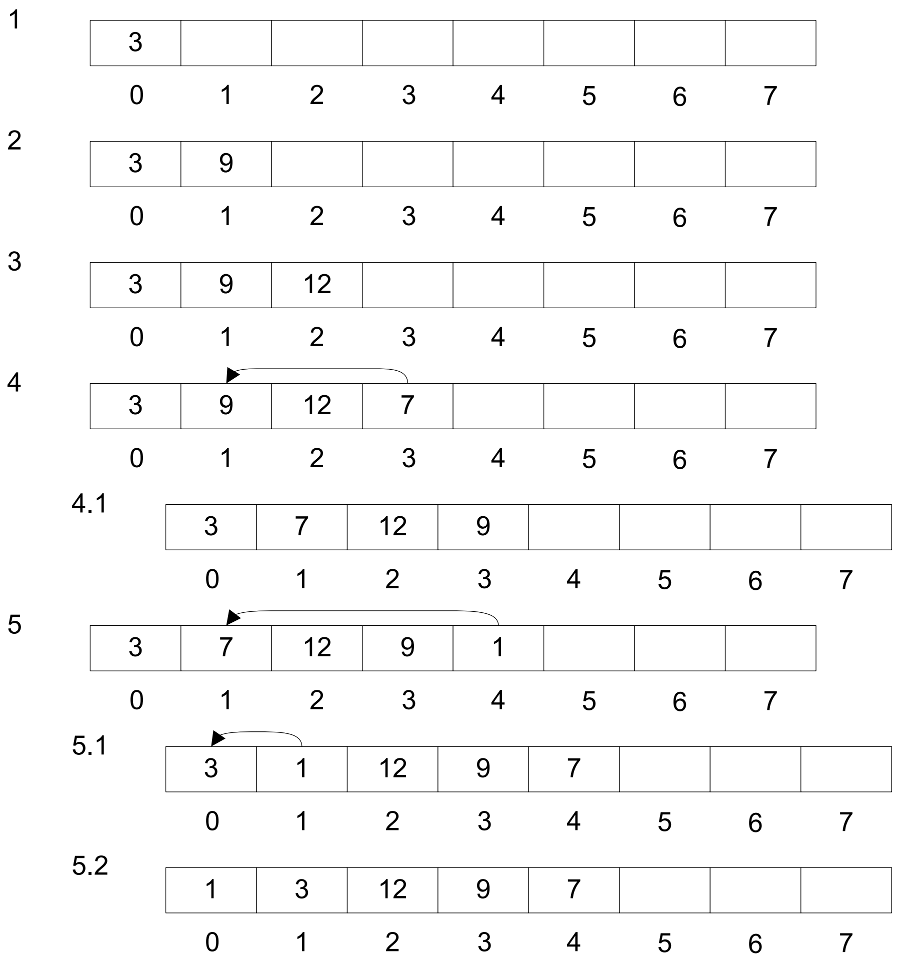

Figure 4.5: Inserting values into a min-heap

```
1) algorithm Add($value$)
2)     Pre: $value$ is the value to add to the heap
3)          Count is the number of items in the heap
4)     Post: the value has been added to the heap
5)     $heap$[Count] $\leftarrow$ $value$
6)     Count $\leftarrow$ Count $+1$
7)     MinHeapify()
8) end Add
```

```
1) algorithm MinHeapify()
2)     Pre: Count is the number of items in the heap
3)          $heap$ is the array used to store the heap items
4)     Post: the heap has preserved min heap ordering
5)     $i \leftarrow$ Count $-1$
6)     while $i > 0$ and $heap[i] < heap[(i-1)/2]$
7)         Swap($heap[i]$, $heap[(i-1)/2]$
8)         $i \leftarrow (i-1)/2$
9)     end while
10) end MinHeapify
```

The design of the *MaxHeapify* algorithm is very similar to that of the *MinHeapify* algorithm, the only difference is that the $<$ operator in the second condition of entering the while loop is changed to $>$.

4.2 Deletion

Just as for insertion, deleting an item involves ensuring that heap ordering is preserved. The algorithm for deletion has three steps:

1. find the index of the value to delete
2. put the last value in the heap at the index location of the item to delete
3. verify heap ordering for each subtree which used to include the value

```
1) algorithm Remove(value)
2)     Pre: value is the value to remove from the heap
3)          left, and right are updated alias' for $2 * index + 1$, and $2 * index + 2$ respectively
4)          Count is the number of items in the heap
5)          heap is the array used to store the heap items
6)     Post: value is located in the heap and removed, true; otherwise false
7)     // step 1
8)     index ← FindIndex(heap, value)
9)     if index < 0
10)       return false
11)    end if
12)    Count ← Count −1
13)    // step 2
14)    heap[index] ← heap[Count]
15)    // step 3
16)    while left < Count and heap[index] > heap[left] or heap[index] > heap[right]
17)       // promote smallest key from subtree
18)       if heap[left] < heap[right]
19)          Swap(heap, left, index)
20)          index ← left
21)       else
22)          Swap(heap, right, index)
23)          index ← right
24)       end if
25)    end while
26)    return true
27) end Remove
```

Figure 4.6 shows the *Remove* algorithm visually, removing 1 from a heap containing the values 1, 3, 9, 12, and 13. In Figure 4.6 you can assume that we have specified that the backing array of the heap should have an initial capacity of eight.

Please note that in our deletion algorithm that we don't default the removed value in the *heap* array. If you are using a heap for reference types, i.e. objects that are allocated on a heap you will want to free that memory. This is important in both unmanaged, and managed languages. In the latter we will want to null that empty hole so that the garbage collector can reclaim that memory. If we were to not null that hole then the object could still be reached and thus won't be garbage collected.

4.3 Searching

Searching a heap is merely a matter of traversing the items in the heap array sequentially, so this operation has a run time complexity of $O(n)$. The search can be thought of as one that uses a breadth first traversal as defined in §3.7.4 to visit the nodes within the heap to check for the presence of a specified item.

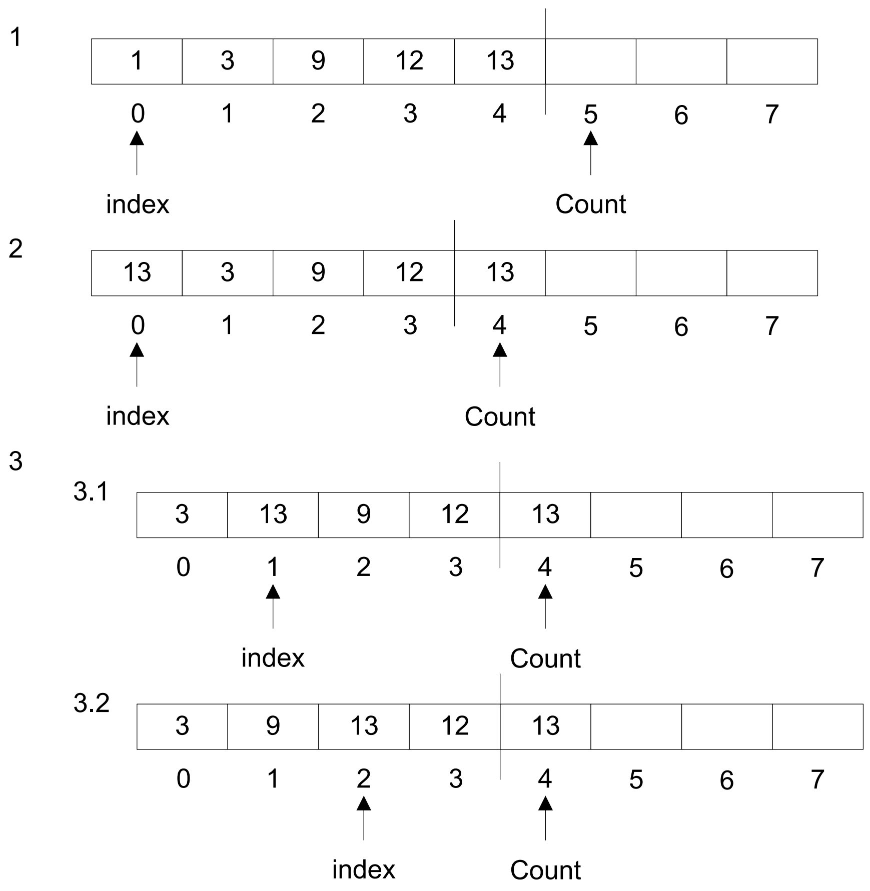

Figure 4.6: Deleting an item from a heap

```
1) algorithm Contains($value$)
2)     Pre: $value$ is the value to search the heap for
3)          Count is the number of items in the heap
4)          $heap$ is the array used to store the heap items
5)     Post: $value$ is located in the heap, in which case true; otherwise false
6)     $i \leftarrow 0$
7)     while $i <$ Count and $heap[i] \neq value$
8)         $i \leftarrow i + 1$
9)     end while
10)    if $i <$ Count
11)        return true
12)    else
13)        return false
14)    end if
15) end Contains
```

The problem with the previous algorithm is that we don't take advantage of the properties in which all values of a heap hold, that is the property of the heap strategy being used. For instance if we had a heap that didn't contain the value 4 we would have to exhaust the whole backing heap array before we could determine that it wasn't present in the heap. Factoring in what we know about the heap we can optimise the search algorithm by including logic which makes use of the properties presented by a certain heap strategy.

Optimising to deterministically state that a value is in the heap is not that straightforward, however the problem is a very interesting one. As an example consider a min-heap that doesn't contain the value 5. We can only rule that the value is not in the heap if $5 >$ the parent of the current node being inspected and $<$ the current node being inspected $\forall$ nodes at the current level we are traversing. If this is the case then 5 cannot be in the heap and so we can provide an answer without traversing the rest of the heap. If this property is not satisfied for any level of nodes that we are inspecting then the algorithm will indeed fall back to inspecting all the nodes in the heap. The optimisation that we present can be very common and so we feel that the extra logic within the loop is justified to prevent the expensive worse case run time.

The following algorithm is specifically designed for a min-heap. To tailor the algorithm for a max-heap the two comparison operations in the **else if** condition within the inner **while** loop should be flipped.

```
1) algorithm Contains(value)
2)    Pre: value is the value to search the heap for
3)         Count is the number of items in the heap
4)         heap is the array used to store the heap items
5)    Post: value is located in the heap, in which case true; otherwise false
6)    $start \leftarrow 0$
7)    $nodes \leftarrow 1$
8)    while $start <$ Count
9)       $start \leftarrow nodes - 1$
10)      $end \leftarrow nodes + start$
11)      $count \leftarrow 0$
12)      while $start <$ Count and $start < end$
13)         if $value = heap[start]$
14)            return true
15)         else if $value >$ Parent($heap[start]$) and $value < heap[start]$
16)            $count \leftarrow count + 1$
17)         end if
18)         $start \leftarrow start + 1$
19)      end while
20)      if $count = nodes$
21)         return false
22)      end if
23)      $nodes \leftarrow nodes * 2$
24)   end while
25)   return false
26) end Contains
```

The new *Contains* algorithm determines if the *value* is not in the heap by checking whether *count* = *nodes*. In such an event where this is true then we can confirm that $\forall$ nodes n at level i : $value >$ Parent(n), $value < n$ thus there is no possible way that *value* is in the heap. As an example consider Figure 4.7. If we are searching for the value 10 within the min-heap displayed it is obvious that we don't need to search the whole heap to determine 9 is not present. We can verify this after traversing the nodes in the second level of the heap as the previous expression defined holds true.

4.4 Traversal

As mentioned in §4.3 traversal of a heap is usually done like that of any other array data structure which our heap implementation is based upon. As a result you traverse the array starting at the initial array index (0 in most languages) and then visit each value within the array until you have reached the upper bound of the heap. You will note that in the search algorithm that we use *Count* as this upper bound rather than the actual physical bound of the allocated array. *Count* is used to partition the conceptual heap from the actual array implementation of the heap: we only care about the items in the heap, not the whole array—the latter may contain various other bits of data as a result of heap mutation.

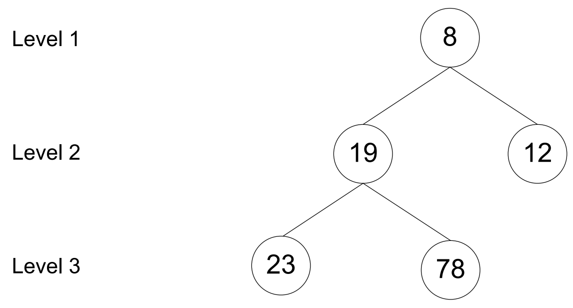

Figure 4.7: Determining 10 is not in the heap after inspecting the nodes of Level 2

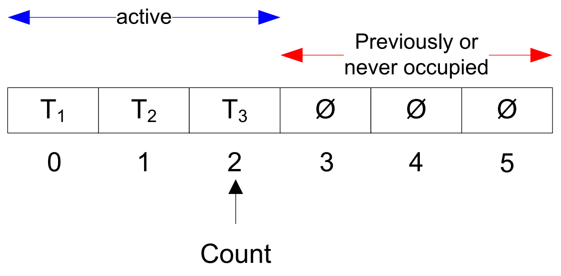

Figure 4.8: Living and dead space in the heap backing array

If you have followed the advice we gave in the deletion algorithm then a heap that has been mutated several times will contain some form of default value for items no longer in the heap. Potentially you will have at most $LengthOf(heapArray) - Count$ garbage values in the backing heap array data structure. The garbage values of course vary from platform to platform. To make things simple the garbage value of a reference type will be simple $\emptyset$ and 0 for a value type.

Figure 4.8 shows a heap that you can assume has been mutated many times. For this example we can further assume that at some point the items in indexes $3 - 5$ actually contained references to live objects of type T. In Figure 4.8 subscript is used to disambiguate separate objects of T.

From what you have read thus far you will most likely have picked up that traversing the heap in any other order would be of little benefit. The heap property only holds for the subtree of each node and so traversing a heap in any other fashion requires some creative intervention. Heaps are not usually traversed in any other way than the one prescribed previously.

4.5 Summary

Heaps are most commonly used to implement priority queues (see §6.2 for a sample implementation) and to facilitate heap sort. As discussed in both the insertion §4.1 and deletion §4.2 sections a heap maintains heap order according to the selected ordering strategy. These strategies are referred to as min-heap, and max heap. The former strategy enforces that the value of a parent node is less than that of each of its children, the latter enforces that the value of the parent is greater than that of each of its children.

When you come across a heap and you are not told what strategy it enforces you should assume that it uses the min-heap strategy. If the heap can be configured otherwise, e.g. to use max-heap then this will often require you to state this explicitly. The heap abides progressively to a strategy during the invocation of the insertion, and deletion algorithms. The cost of such a policy is that upon each insertion and deletion we invoke algorithms that have logarithmic run time complexities. While the cost of maintaining the strategy might not seem overly expensive it does still come at a price. We will also have to factor in the cost of dynamic array expansion at some stage. This will occur if the number of items within the heap outgrows the space allocated in the heap's backing array. It may be in your best interest to research a good initial starting size for your heap array. This will assist in minimising the impact of dynamic array resizing.