Datum Mechanics

David A. Hendrix

May 5, 2026

Contents

1 Space and Time **4**
- 1.1 Time Complexity . . . 4
  - 1.1.1 Big-O . . . 4
  - 1.1.2 Big-Omega . . . 5
  - 1.1.3 Big-Theta . . . 6
  - 1.1.4 Average-case, Best-case, and Worst-case scenarios . . . 6
- 1.2 Searching . . . 6
  - 1.2.1 Linear Search . . . 6
  - 1.2.2 Binary Search . . . 7
- 1.3 Sorting . . . 9
  - 1.3.1 Bogo Sort . . . 10
  - 1.3.2 Bubble Sort . . . 10
  - 1.3.3 Merge Sort . . . 12
  - 1.3.4 Quick Sort . . . 14
- 1.4 Space Complexity . . . 16
  - 1.4.1 The Building Blocks of Python . . . 18
- 1.A Geometric Series . . . 19

2 Lists **20**
- 2.1 Data Structures and Abstract Data Types . . . 20
- 2.2 Lists as an Abstract Data Type . . . 21
- 2.3 Static Arrays . . . 22
  - 2.3.1 Advantages and Disadvantages of the Static Array . . . 23
  - 2.3.2 Dunder Methods . . . 23
- 2.4 Dynamic Arrays . . . 23
- 2.5 Implementing a List with Dynamic Arrays . . . 23
  - 2.5.1 Declaring a Dynamic Array . . . 23
  - 2.5.2 Defining New Objects . . . 24
  - 2.5.3 Resizing a Dynamic Array . . . 25
- 2.6 Dynamic Arrays: Space and Time . . . 26
  - 2.6.1 Space Complexity of Dynamic Arrays . . . 26
  - 2.6.2 Time Complexity of Dynamic Arrays . . . 27
  - 2.6.3 Time Complexity of Appending an Element . . . 27
  - 2.6.4 The Time Complexity of Inserting to a Dynamic Array . . . 27
  - 2.6.5 The Time Complexity of Deleting from a Dynamic Array . . . 28
  - 2.6.6 Summary of the Time Complexity for Dynamic Arrays . . . 28
  - 2.6.7 Time complexity of Finding the Largest Element . . . 28
- 2.7 Linked Lists . . . 29
- 2.8 Implementing a List with Linked Lists . . . 29
  - 2.8.1 Declaring a Linked List . . . 29
  - 2.8.2 Declaring a Doubly Linked List . . . 30
- 2.9 Linked Lists: Space and Time . . . 31

2.9.1 Space Complexity of Linked Lists . . . 31
2.9.2 Time Complexity of Traversing a Linked List . . . 31
2.9.3 Time Complexity of Clearing a Linked List . . . 32
2.9.4 Time Complexity of Appending to a Linked List: `insert_last` . . . 33
2.9.5 Time Complexity of `insert_first` with a Linked List . . . 33
2.9.6 Time Complexity of Insertion into a Linked List . . . 34
2.9.7 Time Complexity of Reversing a Linked List . . . 36
2.9.8 Summary of the Time Complexity of Linked Lists . . . 37

3 Stacks and Queues **39**
3.1 Stacks . . . 39
3.1.1 Stacks as an ADT . . . 39
3.1.2 A simple idea of a stack . . . 40
3.1.3 Stacks Implemented as Linked Lists . . . 40
3.1.4 Stacks Implemented as Dynamic Arrays . . . 42
3.2 Queues . . . 43
3.2.1 Queues as an ADT . . . 44
3.2.2 Implementing a Queue with a Linked List . . . 44
3.2.3 Implementing a Queue with a Dynamic Arrays . . . 44
3.3 Implementing a Queue Using Two Stacks . . . 45
3.3.1 Implementing `enqueue()` with stacks . . . 45
3.3.2 Implementing `dequeue()` with stacks . . . 45
3.4 Implementing a Stack Using Two Queues . . . 46
3.4.1 Implementing Stack `push()` with Two Queues. . . . 47
3.4.2 Implementing stack `pop()` with Two Queues. . . . 48
3.5 Circular Arrays . . . 48
3.5.1 Circular Buffers . . . 50
3.5.2 Remainder operator . . . 50
3.5.3 Implementing `enqueue()` and `dequeue()` Operations with a Circular Array . . . 51
3.5.4 Resizing and Reindexing Circular Arrays . . . 52
3.6 Deques: Double-ended Queues . . . 53
3.6.1 Implementing a Deque with a Doubly Linked List . . . 53
3.6.2 Front and Back Sentinels . . . 53

4 Dictionaries **60**
4.1 Implementing Dictionaries with Hash Tables . . . 60
4.1.1 Hash Functions . . . 61
4.1.2 The virtues of a good hash function . . . 61
4.1.3 Hash function collisions . . . 61
4.2 Hash Tables: Space and time. . . . 62
4.2.1 Resolving hash table collisions with “chaining” . . . 63
4.2.2 The load factor for a hash table using chaining . . . 63
4.2.3 Hash table with chaining: worst-case time complexity . . . 64
4.2.4 Hash Table with chaining: average time complexity . . . 64
4.2.5 Adjusting the load factor with Dynamic Hash Tables . . . 64
4.2.6 Resolving hash table collisions with “open addressing” . . . 64
4.2.7 Time-complexity of Open Addressing . . . 65
4.2.8 Modeling Collisions as a Bernoulli Process . . . 65
4.2.9 Modeling Collisions with a Geometric Distribution . . . 66
4.3 Implementing a Dictionary with Binary Search Trees . . . 67
4.3.1 Storing and Retrieving Keys in a Binary Search Trees . . . 69
4.4 Binary Search Tree: Space and Time . . . 71
4.4.1 Full and perfect binary trees . . . 72
4.4.2 The Time Complexity of Searching a BST . . . 73

4.4.3 The Time Complexity of `insert` with a BST . . . 74
4.4.4 The Time Complexity of Deleting from a BST . . . 74
4.5 Tree Traversal . . . 79
4.5.1 Depth-first traversal of a binary tree . . . 79
4.6 Implementing a Dictionary with AVL Trees . . . 81
4.6.1 Balance and binary search trees . . . 81
4.6.2 Balance Factor . . . 81
4.6.3 Maintaining height balance with AVL Trees . . . 82
4.6.4 Rotations with AVL Trees . . . 82
4.6.5 Situations when the AVL Tree is rebalanced . . . 84
4.7 AVL Trees: Space and Time . . . 85
4.7.1 AVL Trees: Space Complexity . . . 85
4.7.2 AVL Trees: Time Complexity . . . 85

5 Priority Queues **86**
5.1 Implementing Priority Queues with Heaps . . . 86
5.1.1 Heaps correspond to complete trees . . . 87
5.1.2 insert(P,x) . . . 88
5.2 Heaps: Space and Time . . . 88
5.2.1 Time Complexity of Insert to a Heap . . . 88
5.2.2 $max(P)$ and $max\_dequeue(P)$ . . . 89
5.2.3 The Time Complexity of Fixing a Node . . . 89
5.2.4 Building a max-heap from an unordered array. . . . 90
5.2.5 The Time Complexity of `build_heap()` . . . 90
5.2.6 Sorting a list with heap-sort . . . 91

Chapter 1

Space and Time

1.1 Time Complexity

Time complexity involves computing the number of steps, or reasonable bounds on the number of steps, for a particular task. We'll refer to this number of steps as $T(n)$ to describe the work involved, or the "time" it takes in terms of the number of operations. We often use the number of steps because different hardware can take a different amount of actual time to compute the same thing. When working with different data structures, we will be able to compute $T(n)$ explicitly for many examples. But before we do that, let's review some of the fundamental properties of Big-O notation.

Time complexity is usually defined by a particular set of functions that describe the order of growth. The following table provides a list of some of the common functions:

Table 1.1: Common Time Complexity Functions Ordered by Growth Rate

Big O Notation	**Common Name**
$O(1)$	Constant time
$O(\lg n)$	Logarithmic time
$O(n)$	Linear time
$O(n \lg n)$	Log-linear or Quasilinear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(n^k)$	Polynomial time (where $k > 1$)
$O(2^n)$	Exponential time
$O(n!)$	Factorial time

1.1.1 Big-O

Big-O notation describes the upper bound of the number of computational steps $T(n)$ that it takes to complete a particular task. To say that $f(n)$ is $O(g(n))$ is to say that $f(n)$ will not exceed $g(n)$, within a factor of a constant for some large n. We say that a function $f(n)$ is $O(g(n))$ if there exists a positive constant c and an integer n_0 such that:

$$0 \leq f(n) \leq c \cdot g(n), \text{ for all n} \geq n_0$$

For a given function $f(n)$, our goal would be to derive a constant c and a starting point, usually $n_0 = 1$, such that the rule above is satisfied.

Example 1.1. Show that $f(n) = 3n^2 + 5n + 8$ is $O(n^2)$

Solution. First, we can create an inequality in the direction that we want, by making every term in the function the same order as the highest order term

$$f(n) = 3n^2 + 5n + 8 \leq 3n^2 + 5n^2 + 8n^2$$

The reason for this is that we know that for $n \geq 1$ this inequality will be true, and the right-hand side will always be larger because n is growing.

Next, we group together all the terms so that there is one coefficient:

$$f(n) = 3n^2 + 5n + 8 \leq (3 + 5 + 8)n^2 = 16n^2$$

We are pretty much done at this point. We have found a constant $c = 16$ such that for $n \geq 1$ the inequality is true. Therefore, the original function $f(n)$ is $O(n^2)$.

Example 1.2. Show that $f(n) = \log_{10}(n^3)$ is $O(\lg(n))$

Solution. In this example, we are comparing two bases of the logarithm, log-based 10 and log-based 2, which we denote by lg() for brevity. Furthermore, the n is raised to the third power for $f(n)$. First, we note the properties of logarithms, in that $\lg(x^a) = a \lg(x)$. This means that:

$$f(n) = \log_{10}(n^3) = 3 \log_{10}(n)$$

Then note another property of logarithms, in that we can switch between bases with a multiplicative factor: $\log_b(x) = \lg(x)/\lg(b)$. This means that we can do the following:

$$f(n) = 3 \log_{10}(n) = \frac{3}{\lg(10)} \lg(n)$$

We have found a constant $c = \frac{3}{\lg(10)}$ such that for $n \geq 1$ the inequality is true. Therefore, the original function $f(n)$ is $O(\lg(n))$. In fact, you will note that the equality is true, but this satisfies the inequality as well. This type of derivation is therefore applicable to other measures of time complexity.

1.1.2 Big-Omega

Big-Omega describes the lower bound on the number of steps required to compute a particular task. Big-Omega is, in some sense, the complement to Big-O because it describes the lower bound in contrast to Big-O as the upper bound. We say that a function $f(n)$ is $\Omega(g(n))$ if there exists a positive constant c and an integer n_0 such that:

$$0 \leq c \cdot g(n) \leq f(n), \text{ for all n} \geq n_0$$

Example 1.3. Show that $f(n) = (\frac{n+1}{n}) \lg(n)$ is $\Omega(\lg(n))$

Solution. First, we might want to find a common denominator for the fractional coefficient term. For large n, this would seem to approach 1, so let's show that.

$$f(n) = (\frac{n+1}{n}) \lg(n) = (1 + \frac{1}{n}) \lg(n)$$

Our goal is to establish a lower bound, and the expression $1 + \frac{1}{n}$ is always greater than 1.

$$f(n) = (\frac{n+1}{n}) \lg(n) = (1 + \frac{1}{n}) \lg(n) \geq \lg(n)$$

We are pretty much done at this point. We have found a constant $c = 1$ such that for $n \geq 1$ the inequality is true. Therefore, the original function $f(n)$ is $\Omega(\lg(n))$. Note that a similar approach could also be used to show that the function is $O(\lg(n))$.

1.1.3 Big-Theta

There is perhaps something missing with upper and lower bounds on their own, because they do not give a tight bound on the time complexity. Big-Theta describes the exact growth rate, and similarly describes a tight bound of a function or time complexity. We say that a function $f(n)$ is $\Theta(g(n))$ if there exists positive constants c_1 and c_2 and an integer n_0 such that:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \text{ for all n} \geq n_0$$

For $f(n)$ to be $\Theta(g(n))$ it must be the case that it is $O(g(n))$ and $\Omega(g(n))$. It says that $f(n)$ behaves like $g(n)$ for large n.

If we know that a function is $\Theta(g(n))$, then we know that it is a tight bound. From this, we can conclude that it is also $O(g(n))$ and $\Omega(g(n))$. Our derivations could also demonstrate that a function is $\Theta(g(n))$ by showing that it is $O(g(n))$ and $\Omega(g(n))$. In other words, a function is $\Theta(g(n))$ if and only if it is also $O(g(n))$ and $\Omega(g(n))$.

1.1.4 Average-case, Best-case, and Worst-case scenarios

In the average-case scenario, we are using some probability or statistics to describe the number of steps required to complete the task on average. This can give a real-world expectation of the time complexity. The best-case scenario provides the situation where the setup is lucky, and we can take as few steps as possible. The worst-case scenario is where the setup is as bad as possible, and the number of steps is as large as can be. We can use Big-O, Big-Omega, and Big-Theta to describe each of the worst-case, best-case, and average-case scenario time complexities.

There is perhaps an important distinction to be made. Many people describe Big-O as the worst-case time complexity, and in many cases, this is true; however a more precise statement is that because Big-O describes the upper bound, and the number of steps cannot do worse than that upper bound. However, when we talk about "worst-case scenarios," we can also apply Big-O, Big-Omega, and Big-Theta to the worst-case scenario, hence there is often a distinction between "worst-case" and Big-O in the context of time complexity. With that in mind, let's review some searching and sorting algorithms.

1.2 Searching

For searching algorithms, we have a situation where we have a particular value x and we want to search for it in an array of n elements. It can be valuable to return the index of the element of the array.

1.2.1 Linear Search

A linear search algorithm would explicitly traverse the array, one element at a time, until it finds the desired element, and it would return the index. Here is some python code that would perform this task:

```
def linear_search(arr, target):

    for i in range(len(arr)):
        # Check if target is at index i
        if arr[i] == target:
            return i

    # Target not present in array
    return -1
```

Let's consider the different scenarios explicitly, and examine time complexity for each scenario.

Worst-case Scenario

In the worst-case scenario for linear search, you would have to visit all n elements. This can happen if the element being searched for is at the end of the array, or is not there at all. In this case, we can say that $T(n) = n$. Because we have an explicit function for this scenario, we can say that the time complexity is $\Theta(n)$. We could also show that the situation is $O(n)$ and $\Omega(n)$ using a derivation similar to above, and these two must be true in order for the tight bound to be $\Theta(n)$.

Best-case Scenario

In the best-case scenario, the element is the first element in the array, and therefore only one comparison is made, so $T(n) = 1$. We could say that this situation is $\Theta(1)$.

Average-case Scenario

For the average case, we might expect that the element we are searching for would be in the array at some probability p. In the case when the element we are searching for is in the array, we would assume it is equally likely to be in any one of the n positions with probability $\frac{1}{n}$. So the expected value for the number k of steps would be:

$$E[k] = \sum_{k=1}^{n} k \cdot \frac{1}{n} = \frac{1}{n}\left(\sum_{k=1}^{n} k\right) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

The element would not be in the array with probability, $1 - p$, in that case one would have to search n steps. If we combine these two situations into one expression, we get the following:

$$T_{avg}(n) = p\left(\frac{n+1}{2}\right) + (1-p)n = (1 - \frac{p}{2})n + \frac{p}{2}$$

We can then show that this expression is $O(n)$ using the methods as above, because this is a polynomial in n with the largest term being linear in n. We can also show that this expression is $\Omega(n)$ by a similar derivation. Combining these shows that the expression is $\Theta(n)$ for the average case.

1.2.2 Binary Search

For binary search, we have a requirement that the elements of the array are sorted. When this is the case, we always know if the element with the desired value is above or below a given index because of the condition of being sorted. We can implement this algorithm with the following python code:

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        # Calculate the middle index
        mid = (low + high) // 2

        # Check if target is at mid
        if arr[mid] == target:
            return mid
        # If target is greater, ignore left half
        elif arr[mid] < target:
            low = mid + 1
        # If target is smaller, ignore right half
        else:
            high = mid - 1
```

```
    # Target not present in array
    return -1
```

Because the binary search algorithm has a divide-and-conquer strategy, and splits the search space in two at each iteration, we have fewer and fewer remaining terms to examine by a factor of two. This can be visualized by the following binary tree structure:

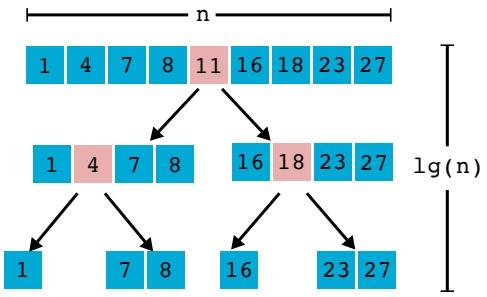


Figure 1.1: A representation of the tree-like structure of binary search.

Best-case Scenario

The best-case scenario is that we select the element we are searching for on the first try. This would correspond to the value we are searching for residing exactly in the middle of the array. The time complexity $T(n)$ of this kind of scenario would be $\Theta(1)$.

Worst-case Scenario

The worst-case scenario would involve iteration of the loop of the search as far as possible in order to find the desired element, which would be $T(n) = \lg(n)$ terms. This will happen when the element being searched for is at the beginning or end of the array, because the search checks the middle element first, and continues to do so; therefore, the outer elements get visited last. This time complexity would be the case even if the element is not in the array, because it would take $\Theta(\lg(n))$ to make that determination.

Average-case Scenario

The average case for binary search can be computed explicitly. Let's assume we have a sorted array of n elements, and for simplicity, we can assume that the number n is such that $n = 2^h - 1$, which corresponds to the exact number of elements that would fit in a perfect binary tree of height h. If this is not the case, we are dealing with correction terms that could get messy, but should not change the overall time complexity. Since we can visualize the search as traversing this binary tree in a depth-first approach, going one layer or "depth" of the tree at each step. The depths of the tree exponentially grow in terms of the number of nodes, so that we have 2^d nodes at each depth d, with $d = 0$ corresponding to the root. We can assume that when the desired element is at depth d, we have $d + 1$ steps in the computation to find it, because we have one comparison when it is at depth $d = 0$ (the root), and two comparisons when the depth is $d = 1$, which correspond to checking the root, and then checking node we visit at $d = 1$.

Our average time complexity would need to sum up all the number of operations to find each element, and divide by n to get the average. The sum of all the number of operations we would do would be given by:

$$S = \sum_{d=0}^{h-1} (d+1)2^d$$

And our average time complexity is then given by:

$$T_{avg}(n) = \frac{S}{n} = \frac{1}{n}\sum_{d=0}(d+1)2^d$$

To solve the sum S into a compact form, we can do several things, but one relatively straightforward way is to compute $S - 2S$. We will see another way later on in the context of heaps.

$$S - 2S = -S = \sum_{d=0}^{h-1}(d+1)2^d - \sum_{d=0}^{h-1}(d+1)2^{d+1}$$

We can do a change of variables after recognizing a pattern, letting $d' = d + 1$. This change of variables will also change the bounds of the sum (for example, while d goes up to $h-1$, d' will go up to h.

$$-S = \sum_{d=0}^{h-1}(d+1)2^d - \sum_{d=1}^{h}(d')2^{d'}$$

This form reveals that the coefficients in front of the 2^d terms subtract away, leaving just $(d+1) - d = 1$ for all the coefficients except the last one in the second sum. Let's cancel those terms out, and include the last term from the second sum, and multiply everything by -1:

$$S = h2^h - \sum_{d=0}^{h-1} 2^d$$

The second term on the right-hand side is a standard geometric series apart from the sum going up to $h-1$. The most common form of this sum is (see Appendix 1.4.1):

$$\sum_{k=0}^{n} ar^k = \frac{a(1-r^{n+1})}{1-r} \tag{1.1}$$

In this sum, the max is n, and the highest power in the numerator of the result is $n+1$. So in our case, $a = 1$ and the sum is going to $h-1$, so the power of the corresponding term in the numerator will be h. This gives an expression for S as:

$$S = h2^h - \frac{1-2^h}{1-2} = h2^h + 1 - 2^h = (h-1)2^h + 1$$

Using our initial assumption that $n = 2^h - 1$, this gives for our average-case time complexity the following:

$$T_{avg}(n) = \frac{(h-1)2^h + 1}{n} = \frac{(n+1)(\lg(n+1) - 1) + 1}{n} = \frac{(n+1)\lg(n+1) - n}{n} = \frac{n+1}{n}\lg(n+1) - 1$$

For sufficiently large n, the $\frac{n+1}{n}$ approaches 1, and the -1 term becomes negligible, which gives us the final result of $O(lg(n))$. With our expression for $T_{avg}(n)$ we can also show that this expression is $\Omega(\lg(n))$ and $\Theta(\lg(n))$.

1.3 Sorting

Sorting algorithms provide some great examples for considering time and space complexity. Let's review some sorting algorithms that exemplify some considerations for complexity analysis. In each case, we are provided an array of length n, and our task is to return the values sorted in ascending order.

1.3.1 Bogo Sort

Bogo Sort is an intentionally inefficient sorting algorithm that illustrates time complexity extremes. This algorithm sorts an array by essentially trying every possible permutation of the elements.

```
import random

def is_sorted(arr):
    # Check if the array is sorted in ascending order.
    for i in range(len(arr) - 1):
        if arr[i] > arr[i + 1]:
            return False
    return True

def bogosort(arr):
    # Shuffle the array randomly until it is sorted.
    attempts = 0
    while not is_sorted(arr):
        random.shuffle(arr)
        attempts += 1
    return arr, attempts
```

Best-case Scenario

In the best case scenario, the array that is input is already sorted. For an array of length n, it takes n steps to confirm whether it is sorted or not, because each element would have to be visited and compared to its neighbors, and this is the function of the `is_sorted()` function defined first. The best case time complexity is $O(n)$ for Bogo Sort. We see right away that $O(n)$ for best case for a sorting algorithm is not a high bar, and will apply to most sorting algorithms.

Worst-case Scenario

The worst case for the above algorithm is going to be something like $O(\infty)$, because there is no guarantee that the correct sorting will ever be found, particularly for a large array, making the solution unbounded. Even though there is a finite probability of finding the sorted array with a random search, it is small and it is theoretically possible for this program to run forever, making the worst-case infinite time. Because this is a non-deterministic algorithm, calculating Big-Theta seems impossible.

Average-case Scenario

As stated, it takes n steps to confirm whether an array is sorted or not, and there would also be $n! = n(n-1)(n-2)\ldots 3 \cdot 2 \cdot 1$ different permutations of an array of n elements. This can be understood as there are n choices for the first element, $n-1$ choices for the second element after the first one is selected, and so on. When we combine these, we get an expression like $\Theta(n \cdot n!)$ for the average case time complexity. This implies that the algorithm is also $O(n \cdot n!)$ and $\Omega(n \cdot n!)$ for the average case because these are guaranteed by the $\Theta(n \cdot n!)$.

1.3.2 Bubble Sort

While Bubble Sort is not practical in real-world use, it does provide a nice illustration of time complexity. Here is an implementation of the algorithm in Python:

```
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
```

```
    # to check if already sorted/no swaps
       swaps = 0

       # loop to compare neighbors, skipping last i sorted elements
       for j in range(0, n - i - 1):
           if arr[j] > arr[j + 1]:
               # Swap elements using Python's simultaneous assignment
               arr[j], arr[j + 1] = arr[j + 1], arr[j]
               swaps += 1

       # If no elements were swapped, the array is sorted
       if swaps == 0:
           break

   return arr
```

Best-case Scenario

The best case scenario for Bubble Sort is going to be the same as Bogo Sort, and will be $O(n)$ when the array is already sorted. This implementation counts the number of swaps in the first pass, and if there are no swaps, then the array is already sorted. We can explicitly see that a nice feature of Bubble Sort is this checking mechanism is built into the core sorting procedure, and is not a separate function like Bogo Sort.

Worst-case Scenario

In the worst-case scenario for Bubble Sort, we will have $O(n^2)$ because of the nested loops that are never exited. The outer loop defines i, the number of elements that will be sorted at the end of the array at each iteration, so that those elements can be skipped. The inner loop ranges from the beginning up to but not including $n-i-1$. The variable j defines the elements to be compared, and goes up to but not including $n-i-1$ but is also compared to $j+1$ which allows the value indexed by $n-i-1$ to be compared. So at each iteration of the outer loop, fewer elements are compared at each step. In the worst-case scenario, we will never see a step where the number of swaps is zero, so each step of the nested for loops will have to be applied. There will be $n-1$ comparisons for the first case (pass 1), $n-2$ for the second case (pass 2), and so on, leaving 1 comparison for the last iteration, pass $n-1$ of the inner loop. Writing this out explicitly, gives:

$$T_{worst}(n) = \sum_{i=0}^{n-2}(n-i-1) \tag{1.2}$$

Note that the outer for loop goes from 0 up to but not including n, hence $n-1$, but the sum in Equation 1.2 goes up to $n-2$. The reason for this is when $i = n-1$, the inner loop will range from 0 to $n-(n-1)-1 = 0$, so there will be no steps when $i = n-1$, which leaves it out of the sum. Our sum can be written out like this:

$$T_{worst}(n) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 = \frac{n(n-1)}{2} \tag{1.3}$$

Here we have used the fact that $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. We can expand this out to be $T_{worst}(n) = \frac{1}{2}n^2 - \frac{1}{2}n$, which will give us a tight bound of $\Theta(n^2)$.

Average-case Scenario

The average case is a much more complicated calculation, but stems from the fact that we must be able to estimate the number of swaps that will happen for a random array. For a very crude approximation of this calculation, let's consider the number of pairs of elements that would need to be swapped. A pair of

elements at indexes i and j such that $i < j$ needs to eventually be swapped if $arr[i] > arr[j]$. For a random distribution, we would expect that half of the $\binom{n}{2}$ pairs would be out of order. This would give:

$$\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}$$

pairs that are out of order. While each pair of elements that is out of order could take multiple swaps to correct, Bubble Sort is more efficient than that, and often times each swap will fix an inverted pair. If we assume that on average, each swap removes one pair that is out of order, then we get that the average case $T_{avg}(n) \approx \frac{n^2}{4}$, which is $\Theta(n^2)$. This ends up being the case, despite the oversimplification here.

1.3.3 Merge Sort

Merge Sort is a much more practical method of sorting, and also is intuitive. It operates by splitting the array recursively, until only one element is left, and then merges elements into sorted subarrays until the whole array is merged back and sorted.

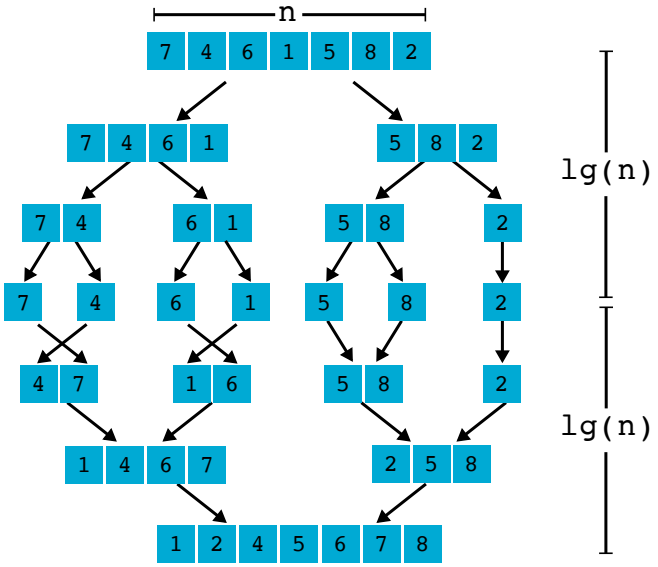


Figure 1.2: A representation of the tree-like structure of Merge Sort.

Let's look at a Python implementation of `merge_sort`. The first thing we need to define is the `merge` function, which will take two sorted arrays and then merge them into one array that preserves the ordering:

```
def merge(left, right):
    result = []
    i = j = 0

    # append to the result array while preserving order
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
```

```
    result.extend(right[j:])
    return result
```

We can see that this function creates a temporary array for the result, which will come into play in the context of space complexity. In terms of time complexity, this step takes $O(n)$ time for two input arrays that are of length n, but will vary depending on the lengths of the arrays and whether they are already sorted. Let's now look at the main sorting code to examine how this function is used.

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Slicing creates new memory (auxiliary space)
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)
```

We can see that Merge Sort is a divide-and-conquer algorithm that splits the input array into two, and again for each half, until we are left with single values that can easily be merged into sorted arrays. As each result gets popped off the recursion stack, we eventually combine the whole thing into a sorted array.

Best-case Scenario

For this implementation, and the standard implementation, there is no check to determine whether the array is already sorted. Adding this step would lead to $\Theta(n)$ in the best case. However, the standard implementation does not do that. In a perfect scenario, each time the array is split, we would have all the left half elements be in order and less than the right half, and have all the right half in order as well. This would result in $n/2$ actual comparisons, and the rest would be taken care of by the `extend()` method at the end where all the remaining elements would just be appended at once. Let's keep this at $T_{merge}(n) = n/2$ for the merge step in the best-case scenario.

We can visualize the recursion as a tree-like structure, seen in Figure 1.2. For the root, we have one merge of size n, with a work of n. For the second level, we have two merges of size $n/2$, which would combine to form a work of n, for the next level we have four merges of size $n/4$, which would combine to be n. At each level, there is a cost of n. There are $\lg(n)$ levels in a tree structure like this, which gives a total time of $T(n) = n \lg(n)$. As a result of this, the time complexity would be $\Theta(n \lg(n))$.

Worst-case Scenario

In the worst-case scenario, we would have $T_{merge}(n) = n - 1$ operations for each merge involving an input array of length n because we would have to do $n - 1$ comparisons. But we still have our tree-like structure. The result of this would be $\Theta(n \lg(n))$ because the added cost of the merges only affect the result by a multiplicative factor.

Average-case Scenario

For the average case, we are still going to have $\Theta(n)$ for our merge steps, so let's assume we have $T_{merge}(n) = cn$ for some constant c. We can apply a recursion relation for the calculation, which would be the following:

$$T(n) = 2T(n/2) + cn$$

We can interpret this as whatever $T(n)$ is, we will have to do the same computation on the left and right subarrays of length $n/2$, which means in addition to the merge step, we will also have to do the work $2T(n/2)$ for each of these subarrays. At the next step of the recursion, we would have:

$$T(n/2) = 2T(n/4) + c(n/2)$$

If we substitute this back into our first equation, we would get:

$$T(n) = 2[2T(n/4) + c(n/2)] + cn = 4T(n/4) + 2cn$$

We could work this out one more time, with $T(n/4) = 2T(n/8) + c(n/4)$, and plugging into the recursion gives:

$$T(n) = 4[2T(n/8) + c(n/4)] + 2cn = 8T(n/8) + 3cn$$

If we examine each step of the sorting procedure, we get something like this:

$$\begin{aligned} T(n) &= 2T(n/2) + cn \text{ (step 1)} \\ T(n) &= 2(2T(n/4) + c(n/2)) + c(n/2) = 4T(n/4) + 2cn \text{ (step 2)} \\ T(n) &= 2(4T(n/8) + \Theta(n/4)) + 2cn = 8T(n/8) + 3cn \text{ (step 3)} \\ &\dots \\ T(n) &= 2^k T(\frac{n}{2^k}) + kcn \text{ (step k)} \end{aligned}$$

This recursion would stop when $\frac{n}{2^k} = 1$, which corresponds to $k = \lg(n)$. If we plug this back into the recursion result, we get the following:

$$T(n) = nT(1) + cn\lg(n)$$

It should be the case that $T(1)$ is $O(1)$ given the problem of sorting one element, which gives the final result as $\Theta(n \cdot lg(n))$.

1.3.4 Quick Sort

Let's first look at an auxiliary function that is defined, known as the partition function.

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            # Swap elements within the original array/list
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

A critical piece of this is the pivot, which corresponds in this implementation to the value at the right bound of the subarray under consideration. The values of the array are compared to this pivot value, and swapped if they are less than the pivot. Let's now look at the `quick_sort` sorting function itself:

```
def quick_sort(arr, low, high):
    if low < high:
        # Partition happens in-place; no new arrays created
        pivot_index = partition(arr, low, high)

        # Space is only used by the recursion stack
        quick_sort(arr, low, pivot_index - 1)
        quick_sort(arr, pivot_index + 1, high)
```

The `quick_sort` function takes in two integers, `low` and `high`. When the `quick_sort` function is first called, it takes in 0 and $n-1$ for these values, and the function is recursively called after updating these bounds using the pivot value returned by the `partition()` function.

Best-case Scenario

The best case scenario for Quick Sort is when the pivot is always the median value of each subarray. In this scenario, the work required to sort the array will follow a recurrence relation that is the same as the average case of Merge Sort. We would have divided the problem into two equal subproblems that are half the size of the original array of length n, plus a linear term corresponding to comparing each element to the pivot. This would be:

$$T_{best}(n) = 2T(n/2) + cn$$

Here we can assume that the constant c is different than before, but in each case they require a much more detailed calculation to get correct. For our purposes, we'll recognize that this is not the exact calculation, but gives us an expression that allows us to categorize the time complexity. As before, we will resolve to this being $\Theta(n \lg(n))$.

Worst-case Scenario

For the worst case for our implementation, we would have a sorted or reverse sorted array as the input. Under this situation, the pivot would be either the minimum or the maximum value in the subarray. Under this situation, our sorting method degenerates into nested for loops, and we end up with partitions that contain 0 and $n-1$ elements. The time complexity ends up being similar to Bubble Sort and is $\Theta(n^2)$ because our work function ends up being $T(n) = n + (n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$.

Average-case Scenario

The average case for Quick Sort is interesting. First, you will note that the `partition()` function will take $n-1$ steps for an input of length n, because each value in the array is compared to the pivot, which is one of the original values, leading to $n-1$ comparisons. In addition, there is a recursive cost of applying the same function to the two subarrays such that whatever the total cost $T(n)$ is, it will include a contribution from $T(i)$ and $T(n-i-1)$ for a pivot index at i. The pivot index can be anywhere in theory, so for the average case, we can assume that it is selected with uniform probability $p = 1/n$. This results in the recurrence relation:

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}[T(i) + T(n-i-1)]$$

Here, in addition to the partition cost of $n-1$ we are computing a weighted average over all the possible pivot indices i. However, we could split up the sum over the two $T(\cdot)$ terms, and they would sum to the same value by symmetry. This would give us:

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

Now we need to remove the sum to solve for $T(n)$. We can subtract this equation from itself in such a way to remove the sum, similar to what we did for the average case of Binary Search. To do this, we will need to remove the n term in the denominator outside of the sum. We can do this by multiplying this equation by n:

$$nT(n) = n(n-1) + 2\sum_{i=0}^{n-1} T(i) \tag{1.4}$$

Now the equation that we are going to subtract is this same thing, but with $n-1$ in place of n. In other words, putting $n-1$ in place of the n everywhere in the equation above, which would give this expression:

$$(n-1)T(n-1) = (n-1)(n-2) + 2\sum_{i=0}^{n-2} T(i) \tag{1.5}$$

If we subtract Equation 1.5 from Equation 1.4, we should get:

$$nT(n) - (n-1)T(n-1) = (n-1)(n-(n-2)) + 2\sum_{i=0}^{n-1} T(i) - 2\sum_{i=0}^{n-2} T(i)$$

Recognizing that $n-(n-2) = 2$ and for the sums, only the last term of the first sum will be left, we will get:

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1)$$

We can isolate $T(n-1)$ onto one side and clean up to give:

$$nT(n) = (n+1)T(n-1) + 2(n-1)$$

Maybe not expected, but we can divide both sides by $n(n+1)$ to get similar terms like $T(n)/(n+1)$ on both sides (with the right-hand side having n replaced with $n-1$):

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

At this point, we are getting close. We can see that to compute $\frac{T(n)}{n+1}$ from $\frac{T(n-1)}{n}$ we need to add $\frac{2(n-1)}{n(n+1)}$, which would correspond to computing $T(n)$ from $T(n-1)$ to some degree. That term that we add will be close to $\frac{2}{n}$ for large n because $\frac{n-1}{n+1} \approx 1$ for large n. In other words, we have shown that:

$$\frac{T(n)}{n+1} \approx \sum_{i=0}^{n} \frac{2}{i}$$

This would be the result of building up the value of $\frac{T(n)}{n+1}$ from starting at $T(1)$ and adding terms to compute the final value. However, we can approximate this, because this sum is approximately equal to (two times) the natural logarithm, $\ln(n)$, because:

$$\ln(n) = \int_1^n \frac{1}{x} dx \approx \sum_{i=1}^{n} \frac{1}{i}$$

Finally, with this approximation, we have the result that:

$$T(n) \approx 2(n+1)\ln(n)$$

We can therefore conclude that in the average case, the time complexity is $\Theta(n \cdot lg(n))$. Going from natural log to log-based-2 is just a multiplicative constant, so we can use our standard lg() function.

1.4 Space Complexity

Space complexity describes how much memory a data structure occupies, and is typically described using Big-O notation, which gives an upper bound of memory usage. In particular, it describes how the maximum required memory scales with the number of elements being considered. Space complexity can have a more straightforward component. For example, lists, stacks, queues and similar abstract data types that we will visit in coming chapters require $O(n)$ to store n elements memory for most implementations. However, in other contexts, we will have to consider the recursion stacks of an algorithm to evaluate the memory usage.

In some situations, how we represent the data matters, and there are pros and cons related to each representation. For something like a graph, which is a network of V vertices (nodes) connected by E edges, the space complexity can range from $O(V+E)$ to $O(V^2)$ depending on how we represent it. In this situation, the more costly $O(V^2)$ may have the advantage of $O(1)$ access to an edge in the graph, while the smaller representation $O(V+E)$ may require a small look-up cost to getting information about edges. This example brings up an important consideration in space complexity: there can often be a trade-off with time complexity. In some cases, you may sacrifice some amount of space complexity to ensure better time complexity.

Another important consideration for space complexity is that we typically consider the cost to be related to the maximum memory usage throughout the runtime of the code. With Python, this makes garbage collection important because whenever a variable falls out of scope, it is freed up for other memory usage by Python's garbage collection. Let's compare `merge_sort` and `quick_sort` implementations.

First, let's look back at `merge_sort`. The first thing we need to define is the merge function, which will take two sorted arrays and then merge them into one result that preserves the ordering. We can see that this function creates a temporary array for the result. Let's look at the sorting code now to examine how this function is used.

Notice that at each call of this function, there are temporary arrays that are created for the two halves of the array, and then the function is called recursively. We would first call the function on the left half of the input array, and recursion would follow the left half only until the left half is sorted, then it would proceed on the right half. As recursion follows the left half, a copy of the left half of the array is created with size $n/2$. Then an array storing the left half *of that half* (one quarter of the original array) is created, which would take up $n/4$ size of memory. Then we would take another half of this quarter, resulting in $n/8$ elements, and so on. At the highest point, we also have the final result array, which would be n terms. If we assume large n that can be divided many times, we get the following series:

$$M(n) = n + n/2 + n/4 + n/8 + \dots$$

If we write this series in sigma notation we get:

$$M(n) = \sum_{k=0}^{\infty} n(\frac{1}{2})^k = n\sum_{k=0}^{\infty}(\frac{1}{2})^k$$

This sum is a geometric series, which will come up numerous times throughout this course, and we have seen for binary search. For Equation 1.1, we can note that in the limit that $n \to \infty$ the numerator will go to 1 if and only if the magnitude of r is such that $|r| < 1$. Therefore, for the infinite series, we have:

$$\sum_{k=0}^{\infty} r^k = \frac{1}{1-r}, \text{ when } |r| < 1$$

Therefore, the memory function $M(n)$ has a finite sum. Returning back to that, we get:

$$M(n) = n\sum_{k=0}^{\infty}(\frac{1}{2})^k = 2n$$

Because $r = 1/2$ in this example, we have $\frac{1}{1-1/2} = 2$ as the result for our sum. So in the end, we still have $O(n)$ memory usage despite a potentially large recursion stack.

A key difference with `quick_sort` is that the array is sorted in place. When we consider the auxiliary function `partition` that is defined, note that it does not make any copies of the array, but only uses the original array reference and two indices. In terms of space, each call to this function would be $O(1)$ because it is not creating new memory usage.

As we can see, `quick_sort` is a recursive function, and each new call within the sorting function will subdivide the data around the pivot index. Because of this recursion, there will be a recursion stack, in which consecutive calls will be added to memory, with each call adding $O(1)$ of memory. The key question is: how many calls will there be on this recursion stack? When `quick_sort` is originally called, it will be using `quick_sort(arr,0,n-1)` as the first call. Assuming that the data is uniformly random, then we can expect

that the partition index will be uniformly distributed among the previous `low` and `high` values. Therefore, our recursive calls to the function might proceed something like this on average:

```
quick_sort(arr,0,n-1)
quick_sort(arr,0,n/2-1)
quick_sort(arr,0,n/4-1)
quick_sort(arr,0,n/8-1)
...
```

Because `quick_sort` is not making copies, we are adding up this number of $O(1)$ terms of the stack. We can formally calculate how much memory this is using a recursion relation. The memory would be allocated something like this:

$$M(n) = 1 + M(n/2)$$

Because we are adding $O(1)$ memory at each call, and then recursively calling on the half-array. We can consider a series of these calls:

$$\begin{aligned} M(n) &= 1 + M(n/2) \text{ (step 1)} \\ M(n) &= 1 + (1 + M(n/4)) = 2 + M(n/4) \text{ (step 2)} \\ M(n) &= 2 + M(n/4) = 2 + (1 + M(n/8)) = 3 + M(n/8) \text{ (step 3)} \\ &\cdots \\ M(n) &= k + M(\frac{n}{2^k}) \text{ (step k)} \end{aligned}$$

The stopping point for this will be when we cannot divide further, which happens when $\frac{n}{2^k} = 1$, which means that $n = 2^k$, or $k = \lg(n)$. When all is said and done, we have the following:

$$M(n) = \lg(n)$$

The result is that because `quick_sort` does not allocate memory for subarrays during the recursion, it wins in the context of space complexity. However, `quick_sort` has an Achilles heel, in that if the data is already sorted, and because the pivot point is always chosen to be the value indexed by `high` (or `low` if it's sorted in descending order), this could end up being $O(n)$ space complexity in the worst-case scenario.

1.4.1 The Building Blocks of Python

We will be building data structures using the Python programming language, and evaluating the time and space complexity of the data structures we create. Python is a high-level language, which can make memory considerations more complex. For example, while a low-level language like C might represent an integer as four bytes (32 bits) of data, in Python it is much larger because it stores other information as well.

Within the module `sys`, there is a function called `getsizeof()` that returns the shallow size in bytes for any input variable or data type. Shallow, meaning it would not give the total memory of objects referenced by the variable. For a variable `x`, the value returned by `sys.getsizeof(x)` quantifies the size of the variable. Another important function is `type()`, which can return the data type of a variable. If we define an integer variable `x` we get the following results:

```
import sys
x = 42
print(sys.getsizeof(x))
print(type(x))

28
<class 'int'>
```

We can see that the variable takes up 28 bytes of data, and is of type `int`. The reason for the large size compared to a C int variable, which is 4 bytes of information, is because in Python, each variable is an object, and there are other attributes to the variable storing the data type, and information for garbage collection. In particular, there is a **reference count**, which takes up 8 bytes of information and stores how many pointers reference the object, the **type pointer**, which takes up 8 bytes and points to the class definition, the **object size**, which takes up 8 bytes and stores the number of digits in the integer, and the **value** itself, which is 4 bytes. The object size can grow to accommodate larger values, which is different from C, which can have overflow errors.

The bottom line here in the context of space complexity is that while there is a larger size to each variable, the addition is $O(1)$ because it is a constant addition. However, the reference count allows for memory to be freed-up and reused in the code once the variable falls out of scope. This is why `merge_sort` is not as bad as it could be, and the arrays created do not add up for the entire algorithm, and only add up for a given recursion stack.

There is no address operator like the C address operator, `&`, to determine where a variable is stored in Python. This is because Python uses virtual memory and manages memory through this garbage collection process, and one does not interact with physical memory in Python. However, in Python there is an `id()` function, which provides a uniquely identifying integer that remains constant for the duration of a variable's existence. However, in CPython, which is the standard version of python, the value returned by `id()` is the virtual memory address of the variable cast as a Python integer. If you would like to see hexadecimal virtual memory addresses, you could do so with a function like `hex(id(x))`.

1.A Geometric Series

The general form is something like this:

$$S = \sum_{k=0}^{n} r^k = 1 + r + r^2 + r^3 + \cdots + r^n$$

One way to solve this is to multiply both sides by $1 - r$. If we distribute this term throughout the sum, we get the following:

$$(1-r)S = (1-r)(1 + r + r^2 + r^3 + \cdots + r^n) = 1 + r + r^2 + r^3 + \cdots - (r + r^2 + r^3 + r^4 + \cdots + r^{n+1})$$

But we see that with the minus sign, every term will cancel out, except the first and last term. This effect is known as a telescoping sum because this large sum collapses down to the two terms:

$$(1-r)S = 1 - r^{n+1}$$

Which means for the original sum, we have the following:

$$\sum_{k=0}^{n} r^k = \frac{1 - r^{n+1}}{1 - r}$$

This is a really useful sum that comes up in a lot of time complexity calculations. In the case when $r = 2$, the denominator becomes $1 - 2 = -1$, which just adds a minus sign, so we can flip the order of the terms in the numerator:

$$\sum_{k=0}^{n} 2^k = 2^{n+1} - 1$$

Because powers of 2 occur so frequently, this sum has a lot of utility.

Chapter 2

Lists

2.1 Data Structures and Abstract Data Types

Data is a representation of a record or measurement stored on a computer, and is typically combinations of words, numbers, images, and video. Videos and images can be represented as a complex arrangement of numbers as well. Video is a time series of images, and images can be broken down into a matrix of pixels, with the pixels corresponding to numbers that could represent RGB or HSV values. So for now, we can think of data as being a collection of text and numbers. It is notable that the word **data** is a plural word, and the singular form is **datum**. Therefore it is actually correct to say *these data* rather than *this data*.

A **data type** is a programmatic way of modeling and representing basic data, like numbers and text. Programming languages typically have rudimentary data types such as numbers, characters, and text. There is only so much you can do with basic data types, so we would like to build more complex arrangements of data to solve different tasks.

A **data structure** is something that we build to organize data. Data structures are often compared by the efficiency that various operations can be performed, such as sorting and retrieving specific values. While some data structures are well-suited for certain tasks, this may come with a trade-off reduction in efficiency at another task.

The organization of data is deeply connected to how we design and think about algorithms. The speed and efficiency with which we can sort and retrieve information from large complex data sets is directly related to how they are organized into data structures.

Similar to the algorithm/program distinction, an **abstract data type (ADT)** is an idealized representation or concept of what our desired object should do. A data structure is a particular solution or implementation of an ADT. While a data structure is a concrete, realized implementation, an ADT is an abstract, idealized concept. We could also think of the ADT as the interface that we use, and that we specify the desired operations and how they should work. We often call this the Application Programming Interface, or API. When we use an API we don't often know what is going on behind the scenes, we just know how the interface works. Theoretical computer scientists think about ADTs as abstract mathematical objects, and write proofs about ADTs much like a mathematician will think about and write proofs about other mathematical objects. By working with ADTs, we can make more general statements that go beyond any specific programming language or implementation.

Consider the rational numbers. We might have an idea of how to add two rational numbers like $\frac{a}{b} + \frac{c}{d}$,

Program	$\leftrightarrow$	Algorithm
Programming Language	$\leftrightarrow$	Pseudocode, Formal language
Data Type, Data Structure	$\leftrightarrow$	Abstract Data Type

Table 2.1: Abstract data types (ADTs) are idealized concepts, like algorithms, and they are implemented with data structures. This concrete/abstract dichotomy is seen throughout computer science.

and expect them to behave within rules governing these types of numbers. The abstract mathematical idea of a rational number would be the ADT in this case, but a programmatic implementation of rational numbers would need to be a defined data type or data structure that satisfies the algebraic rules governing rational numbers.

2.2 Lists as an Abstract Data Type

A list is a fundamental abstract data type. The list ADT is defined as a sequence of elements $L = e_1, \ldots, e_n$ such that each element e_i is of the same data type. Here we are using 1-based indexes for the abstract data type for simplicity. We can assume that $n \geq 0$, where we say we have an empty list in the case when $n = 0$. We might refer to the value i for each e_i as its position or index.

The list ADT is defined such that it can expand and contract as needed to accommodate elements. Lists also have a collection of operations that we should be able to do, such as insert new elements to the list, and delete elements from the list. We also expect to be able to insert to and delete from the list at an arbitrary index i.

Here are some basic operations we might want to consider. As we evaluate particular implementations of a list, we would want to evaluate the complexity of these operations. These operations are all fairly "static" in the sense that they do not require any dynamic sizing or resizing of the list. Once it is created, specific values can be updated.

1. `create(X)` - this function would allocate memory for and create an empty list.
2. `length()` - this function would return the length of the list, defined as the current number of elements n.
3. `get(i)` - this function would return the value of the list at position `i`.
4. `set(i,x)` - this function would set value `x` at position `i`. It would replace the current value at `i`.

Next the **dynamic operations**. These require more dynamic size of our list, and may require additional steps to resize the list.

1. `insert_first(x)` - this function would insert the value `x` to list at the beginning.
2. `delete_first()` - this function would delete the value at the beginning.
3. `insert_last(x)` - this function would insert the value `x` to list at the end.
4. `delete_last()` - this function would delete the value at the end.
5. `insert_at(i,x)` - this function would insert the value `x` to list at the position `i` and shift everything over.
6. `delete_at(i)` - this function would delete the value at position `i` and shift everything over

These operations are dynamic because the changing size n demands that at some point we need to reallocate memory for our list. For example, consider a list

$$L = e_1, \ldots, e_n$$

The function or method `get()` will return the value at index i, so something like this:

$$x = L.get(i)$$

would assign the value e_i to the variable x. We are foreshadowing the Python object-oriented syntax here. If we were to set the value x to the index i, we would do something like this:

$$L.set(i, x)$$

The set method would overwrite any value currently at that index. Alternatively, we might want to insert the value at a particular index, and then shift the other values over to accommodate the new element, and not overwrite anything.

$$L.insert(i, x)$$

This would result in the list being such that $L = e_1, e_2, \ldots, e_{i-1}, x, e_{i+1}, \ldots, e_n$. If we were to insert the item x at index i. The resulting list would be

$$L = e_1, e_2, \ldots, e_{i-1}, x, e_{i'}, e_{i'+1}, \ldots, e_{n'}$$

And here we have the new positions $i' = i + 1$ and $n' = n + 1$. The indices that we use for a particular value have now changed when the original index is greater than or equal to i, and we have changed our length n, which could result in a reallocation of memory of the underlying data structure that is implementing the list.

2.3 Static Arrays

The first implementation of a list we will consider is the data structure called a static array. An array is a contiguous series of adjacent blocks of the same data type, and in some sense is our first true "data structure". The Static Array allows us to add and remove elements, getting and setting items, but does not allow us to resize the array without creating a new instance. Here is the implementation in Python:

```
import ctypes
from typing import Any, Type

class StaticArray:
    def __init__(self, capacity: int, dtype: Type[Any] = ctypes.c_int) -> None:
        # Allocates a fixed-size C-style array in memory.
        self._size: int = capacity
        # Create a new array type: (type * length)
        self._ArrayType: Type[Any] = dtype * capacity
        # Instantiate the type to allocate the actual memory buffer
        self._data: ctypes.Array = self._ArrayType()

    def __len__(self) -> int:
        # Returns the fixed capacity of the array.
        return self._size

    def __getitem__(self, index: int) -> Any:
        # Retrieves an element with bounds checking.
        if not 0 <= index < self._size:
            raise IndexError("StaticArray index out of range")
        return self._data[index]

    def __setitem__(self, index: int, value: Any) -> None:
        # Sets an element with bounds checking.
        if not 0 <= index < self._size:
            raise IndexError("StaticArray index out of range")
        self._data[index] = value

    def __repr__(self) -> str:
        # Visualizes the array by converting to a standard list.
        return f"StaticArray({list(self._data)})"
```

2.3.1 Advantages and Disadvantages of the Static Array

An advantage of the Static Array is that you have random access in constant time to items in the array using an integer index that ranges from `0` to `size-1`. By constant time, we mean that the number of operations needed to access the element does not vary with a changing size of the array. We can access and alter the terms of an array pretty easily.

The major disadvantage of static arrays for many people more familiar with Python lists is the fact that the number of elements is restricted to the size defined at the declaration. The size of an array does not automatically expand unless we explicitly reallocate the expanded memory.

2.3.2 Dunder Methods

This example of a Static Array, being our first data structure implemented in Python is the use of the double-underscore (dunder) method definitions. These "Dunder Methods" help define standard operations and give access to the syntax of standard python objects. For example, the method `__len__` gives the use of the `len()` function in python as you might expect how it would be used: to return the length of the object. The `__getitem__` method allows us to use indices for our array to access elements of the array, as in `A[i]` for an array `A`.

2.4 Dynamic Arrays

In order to use dynamic memory allocation, we need to allocate contiguous memory, and resize dynamically as new data is added and we exceed the defined size. We'll take the starting point of the Static Array, and add that dynamic functionality.

2.5 Implementing a List with Dynamic Arrays

The **dynamic array** is a data structure created to address the limitations of ordinary arrays. The dynamic array has a `length`, which is the current number of elements in the list, but it also has a term called `capacity`, which is the total number of elements that have currently been allocated. Our dynamic array-based implementation of lists would grow and shrink according to our needs by adjusting `capacity` whenever the number of terms grows to exceed it.

2.5.1 Declaring a Dynamic Array

Our implementation will be built on our implementation of the Static Array, but we will add `length` and `capacity` attributes as well as the array to store the data. We would need to allocate memory for the `array` field in a separate statement. We would include all the same functions as the Static Array, but with the following updates:

```
import ctypes
from typing import Any, Type, List

class DynamicArray:
    def __init__(self, dtype: Type[Any] = ctypes.c_int) -> None:
        self._length: int = 0   # Actual number of elements stored
        self._capacity: int = 1   # Current size of the allocated buffer
        self._dtype: Type[Any] = dtype   # Store the C-type
        self._ArrayType: Type[Any] = self._dtype * self._capacity  # Define the type
        self._data: ctypes.Array = self._ArrayType()   # Allocate initial buffer

    # __len__() and other functions will be the same

    def insert_last(self, value: Any) -> None:
        if self._length == self._capacity:
            self._resize(2 * self._capacity)

        self._data[self._length] = value
        self._length += 1

    def __repr__(self) -> str:
        elements: List[Any] = [self._data[i] for i in range(self._length)]
        return f"DynamicArray({elements}, capacity={self._capacity})"
```

We could visualize our dynamic array that stores `int` values like this:

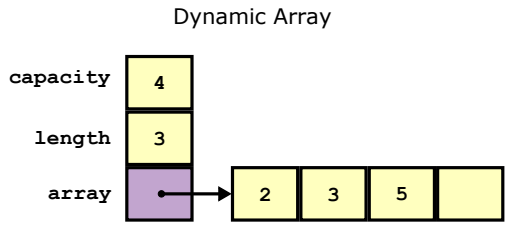


Figure 2.1: A visual representation of a dynamic array.

In this example, our array is defined to be a contiguous block of the defined dtype, which is provided, but with the default value of `int` values. This format for our `__init__` method allows us to easily construct an array of user-defined data type.

2.5.2 Defining New Objects

One feature of this implementation is that it allows us to work with any object, not just integers, for example. In all our sorting algorithms, we just assumed an array of integers, but they can all be written more generally than that.

Let's say we are running a store, or an e-commerce website. We could define an object to store information about the products we are selling. We can define a class like the following:

```
class Product:

    def __init__(self, name: str, brand: str, stock: int, price: float) -> None:
        self.name: str = name
        self.brand: str = brand
        self.stock: int = stock
```

```
        self.price: float = price

    def __str__(self) -> str:
        # Implementation for displaying name, price, and inventory
        return f"{self.name} from {self.brand}: ${self.price:.2f} ({self.stock} in stock)"
```

In this example, we are creating an object that has the product name, the inventory, or amount that we have in stock as an integer, and the price as a floating point number. Let's say we wanted to add to our stock an electric toothbrush called the "Electric Mayhem" from the brand "Dr. Teeth". We could do so with the following command (here applied using the python terminal):

```
>>> from product import Product
>>> toothbrush = Product('Electric Mayhem','Dr. Teeth',100,5.99)
>>> print(toothbrush)
Electric Mayhem from Dr. Teeth: $5.99, (100 in stock)
```

The print statement here is using our `__str__` method, which knows how to convert the object to a string using this f-string method.

If we wanted to add instances of our `Product` object to a dynamic array, we could do the following:

```
dynarray: DynamicArray = DynamicArray(dtype=ctypes.py_object)
new_product: Product = Product(name, brand, stock, price)
dynarray.insert_last(new_product)
```

We are able to append the new product that we have defined into the dynamic array. The `__init__` method has a few steps to create the array. The first step is to set the data type, which is the main input to the object constructor method with the command:

```
self._dtype: Type[Any] = dtype
```

Then, we use this to define the array type, which includes both the capacity and the defined data type:

```
self._ArrayType: Type[Any] = self._dtype * self._capacity
```

This doesn't allocate the memory yet, but a set of parentheses are used to actually allocate the memory using the defined `_ArrayType` variable. In other words, we are sort of defining a method for allocating memory, and then applying it in this line:

```
self._data: ctypes.Array = self._ArrayType()
```

This essentially gives us access to the C function `calloc` to allocate the memory for a contiguous block of a particular size.

2.5.3 Resizing a Dynamic Array

The difference between an ordinary array and a dynamic array is that we have a built-in functionality to adjust the size of our dynamic array as the number of elements increases. After the number of elements `_length` becomes equal to the `_capacity`, we increase the `_capacity` to accommodate new elements. In practice, the `_capacity` is doubled whenever the `_length` exceeds the current capacity. Consider a situation where both the `_length` and `_capacity` are 4, and a new element is added.

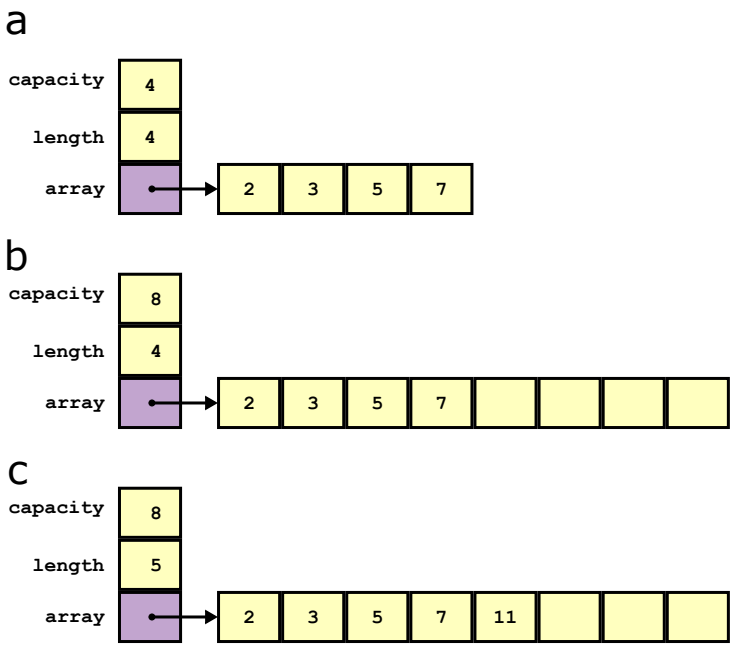


Figure 2.2: Resizing a dynamic array when the number of elements exceeds capacity. **a.** First, the number of elements is equal to the capacity. **b.** If a new element is to be added, the capacity needs to be doubled, and the allocated memory also needs to be doubled. **c.** Now the additional element can be added.

Consider a simple implementation of resize, which would update the length to be an input `new_capacity`, which is typically double the current capacity, and updates the object in place.

```
def _resize(self, new_capacity: int) -> None:
    # Internal utility to resize the internal array.
    new_array_type: Type[Any] = self._dtype * new_capacity
    new_data: ctypes.Array = new_array_type()

    # Copy old data to new buffer
    for i in range(self._length):
        new_data[i] = self._data[i]

    self._data = new_data
    self._capacity = new_capacity
```

In this example, the function first checks if the current total is equal to the capacity. If so, the capacity is doubled and memory for the array is updated to match the current capacity. The data in the old array is copied to the new array, and then the new array is stored in the `_data` field.

2.6 Dynamic Arrays: Space and Time

One of the advantages of dynamic arrays is space considerations are very explicit in their definition. Let's take a look at the space and time complexity of this data structure. Time complexity can be a little less obvious for this data structure, and we will see that it is very efficient because it only rarely updates the size of the array.

2.6.1 Space Complexity of Dynamic Arrays

We therefore say that the memory allocation scales linearly with the length of the input n, and might say that the space complexity is $O(n)$. As with many data structures, there are other values that are stored,

including the size of the array and the capacity of the array, but these are all considered $O(1)$ additions that do not change the overall space complexity. Even while doubling the array size, the space complexity is still $M(n) = 2n$, which is still $O(n)$.

2.6.2 Time Complexity of Dynamic Arrays

Time complexity is always a consideration when evaluating data structures, and we always want to think in terms of the list of operations that we defined for the list ADT. First, let's consider how much time it takes to allocate memory for a dynamic array. If we evaluate our `create_dynarray()` function we defined earlier, it should be clear that the dynamic array is created a few regular steps. Such a situation is said to be "constant time", because we only allocate a length of 1 initially.

We would say that constant-time operations like this are $O(1)$, even if they take two or three steps. The point is that the initialization of the array does not depend on the total number of elements that it could contain, because this is unknown. This could be contrasted with the ordinary array, where we must know the maximum number of elements before hand, and the cost of initialization could take $O(n)$ if the values are initialized to some value (like the behavior of `calloc()`).

2.6.3 Time Complexity of Appending an Element

The time complexity of `insert_last` or in other words appending an element to the end of the list depends on whether we have to resize. In case 1, when we don't have to resize, the cost of appending a value is just $O(1)$. In case 2, when we have to resize, the cost would involve copying over the values. The resizes would be done at the lengths of $2^0, 2^1, 2^2, 2^3, \ldots, 2^k$.

In other words, the number of operations (time) that it takes to dynamically append to the dynamic array up to length n will be $T(n)$, and this depends on the current size n. The cost will be the sum of the cost of reallocating the memory, and copying over the values for to the resized array. This will take $O(2^k)$ for the k-th resize operation.

For a dynamic array that is of size n it would have been resized at the positions the lengths of $2^0, 2^1, 2^2, 2^3, \ldots, 2^k$. At these lengths, there would be a copy step of that length.

On average, this would require $\frac{T(n)}{n}$. But

$$T(n) = \sum_{k=0}^{\lg(n)} 2^k = 2^{\lg(n)} - 1 = n - 1$$

On average, this requires $\frac{T(n)}{n} = \frac{n-1}{n} \approx 1$. Therefore, the average-time resize and copy cost is $O(1)$.

2.6.4 The Time Complexity of Inserting to a Dynamic Array

To insert a value into an arbitrary location of the array, we not only have to resize, but we also have to shift the values over by one. In the worst-case scenario this would involve $O(n)$. Consider an implementation of this using our previously defined Dynamic Array data structure. Note that it is not a dunder method as there is no such defined operation for insert:

```
def insert(self, i: int, value: Any) -> None:
    # Insert value at index i, shifting subsequent elements right.

    # 1. Check if we need more space
    if self._length == self._capacity:
        self._resize(2 * self._capacity)

    # 2. Shift elements to the right
    for j in range(self._length, i, -1):
        self._data[j] = self._data[j-1]
```

```
    # 3. Insert the new value
    self._data[i] = value
    self._length += 1
```

The work done for inserting an element on average would be $T(n) = n/2$ on average, if we assume a uniform distribution of positions. If at each step, a value is inserted at an arbitrary position, we would incur a cost of shifting the values of $O(n)$ for each value, leading to a time-averaged cost of $O(n)$.

2.6.5 The Time Complexity of Deleting from a Dynamic Array

To delete a value from an arbitrary location of the array, we only have to shift the values over by one. In the worst-case scenario this would involve $O(n)$. Consider an implementation of the delete operation:

```
def __delitem__(self, k: int) -> None:
    # Remove element at index k, shifting subsequent elements left.
    if not 0 <= k < self._length:
        raise IndexError('Index out of bounds')

    # Shift elements to the left to fill the gap
    for i in range(k, self._length - 1):
        self._data[i] = self._data[i + 1]

    # Clean up the last reference and decrement size
    self._data[self._length - 1] = None
    self._length -= 1
```

The work done for deleting an element on average would be $T(n) = n/2$ on average, if we assume a uniform distribution of positions. If at each step, a value is deleted at an arbitrary position, we would incur a cost of shifting the values of $O(n)$ for each value, leading to a time-averaged cost of $O(n)$.

2.6.6 Summary of the Time Complexity for Dynamic Arrays

In summary, we can compare the benefits of the dynamic array to the static array with the following table. Clearly, operations like `insert_last` improve time in the time-averaged analysis giving us $O(1)$, and the `create` operation is also $O(1)$ because we initialize to only one element and add as we go.

Operation	Static Array	Dynamic Array
`create()`	$O(n)$	$O(1)$
`length()`	$O(1)$	$O(1)$
`get(i)`	$O(1)$	$O(1)$
`set_at(i,x)`	$O(1)$	$O(1)$
`insert_first(x)`	$O(n)$	$O(n)$
`insert_last(x)`	$O(n)$	$O(n)$, $\Theta(1)*$
`insert_at(i,x)`	$O(n)$	$O(n)$
`delete_first()`	$O(n)$	$O(n)$
`delete_last()`	$O(n)$	$O(1)$
`delete_at(i)`	$O(n)$	$O(n)$
`resize()`	$O(n)$	$O(n)$, $\Theta(1)*$
`copy()`	$O(n)$	$O(n)$, $\Theta(1)*$

* time averaged/amortized

2.6.7 Time complexity of Finding the Largest Element

What would the time complexity be of finding the largest element in the array? This procedure would involve traversing the list at least once, and evaluating the size of each item. Here is an example of finding the maximum value of an ordinary array:

```
def find_max(self) -> Any:
    if self._length == 0:
        raise ValueError("Array is empty")

    # Initialize with the first element
    max_val = self._data[0]

    # Iterate through the actual stored elements
    for i in range(1, self._length):
        if self._data[i] > max_val:
            max_val = self._data[i]

    return max_val
```

Because of the necessity to traverse all the elements of `_data`, this process must be at least $O(n)$. An alternative to this cost would be a list that maintains sorting as you add to it. We will explore data structures for this idea in Chapter 5.

2.7 Linked Lists

2.8 Implementing a List with Linked Lists

The second implementation of a list we will consider is the data structure called a linked list. A linked list is composed of simple nodes that contain a value (or pointer to data) and a pointer to the next node. For the singly linked list case, we have only one link, defined as a pointer or reference for each node, and in the doubly linked list we can have two links.

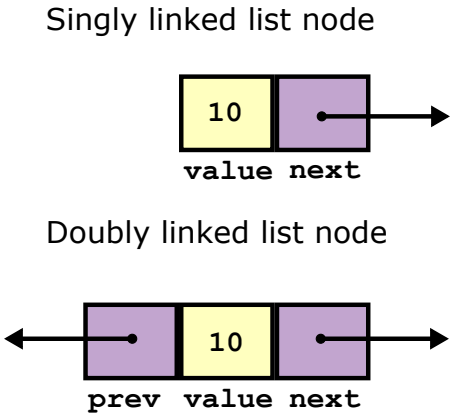


Figure 2.3: A representation of linked list nodes. These nodes can either contain one pointer for the singly linked list, or contain two pointers in the case of the doubly linked list.

There is only so much one can do with a single node, but with several nodes we could build some pretty complex and useful machinery. When several of these relatively simple nodes are linked together, we have a list-like structure.

2.8.1 Declaring a Linked List

If we were to define a Python class to declare a linked list node called `node` that stores integer values, it might look something like this.

```
from typing import Any, Optional

class Node:
    def __init__(self, value: Any, next_node: Optional["Node"] = None):
        self.value = value
        self.next = next_node

    def __repr__(self) -> str:
        return f"Node({self.value})"
```

This example would be for storing integer values, but other data types could be defined. We also want to keep track of both the first and last node of the list with additional pointers that we will call `head` and `tail`. The `tail` pointer is an important component so that appending to the end of the list cost $O(1)$ instead of having to traverse the list to find the end. Creation of a linked list might include the creation of `head` and `tail` pointers, but also a length variable to keep track of how many nodes we have. We could make this declaration define a "control panel" with different associated variables to act as knobs to our data structure. Let's keep it simple here and define something like this:

```
from typing import Optional, Any
from node import Node

class LinkedList:
    def __init__(self):
        self.head: Optional[Node] = None
        self.tail: Optional[Node] = None  # Track the end of the list
        self.length: int = 0

    def __iter__(self):
        # Standard generator for traversal.
        current = self.head
        while current:
            yield current.value
            current = current.next

    def __repr__(self) -> str:
        # Note: join still works here if you have __iter__ defined
        return " -> ".join(map(str, self)) + " -> None"
```

Putting this all together, our singly linked list would then look something like this Figure 2.4

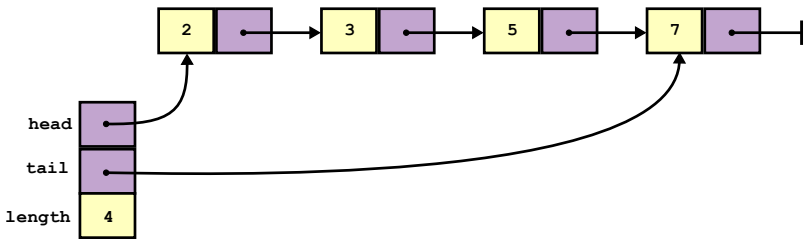


Figure 2.4: A representation a linked list with `head` and `tail` pointers.

2.8.2 Declaring a Doubly Linked List

You could define a similar class to declare a doubly linked list node. For simplicity, let's use the same name in this example, but we could name something else if defining singly and doubly linked list nodes in the same code.

```
from typing import Any, Optional

class Node:
    def __init__(self, value: Any, next_node: Optional["Node"] = None, prev_node: Optional["Node"] = None):
        self.value = value
        self.next = next_node
        self.prev = prev_node
```

The doubly linked list has two pointers, referencing the next node and the previous node.

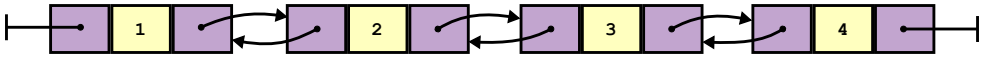


Figure 2.5: A representation a doubly linked list.

The doubly linked list has an advantage that it can be reversed relatively quickly. Doing so would involve swapping the `next` and `prev` pointers, but would require a temporary variable to do so. However, our control panel or interface to the doubly linked list would be essentially the same.

Our linked lists would start off with a length of 0 and no nodes. The `head` and `tail` pointers would just both point to `None`. This would of course all change once we add nodes to our list.

2.9 Linked Lists: Space and Time

Let's take a look at the space and time complexity of linked lists.

2.9.1 Space Complexity of Linked Lists

Like our dynamic array, we should be able to calculate how much memory our linked list occupies for n elements. In our example, we have one `int` variable and one or two pointers, which can be considered a constant addition $O(1)$ for each node. Since we have n nodes, we would say that space complexity is $O(n)$. Furthermore, it can be said that any efficient implementation of lists would occupy $O(n)$ memory. Simply storing the values would always include this penalty.

2.9.2 Time Complexity of Traversing a Linked List

Consider the function that would loop through all the nodes, and print the value at each node. We have this already with our `__iter__` and `__repr__` methods defined above, but we could also define a more simplified method to better consider the mechanics of traversing the linked list.

```
    def display(self):
        """Traverses the list and prints each node's value."""
        current = self.head

        while current:
            # Print the value; use end=" -> " to keep it on one line
            print(current.value, end=" -> ")
            current = current.next

        print("None")
```

Clearly, this would cost $O(n)$ for traversal of the linked list. This is an important consideration when evaluating operations involving a linked list, because many operations involve list traversal.

2.9.3 Time Complexity of Clearing a Linked List

To free a linked list in Python, we don't need to traverse the list, because simply setting the head and tail variables to `None` would clear the linked list. This happens because by setting the `head` attribute to `None`, we would set the reference count to the first node to 0, and then there would be a cascade of garbage collection to the node after, and so on, until the entire linked list is reclaimed in memory.

```
def clear(self) -> None:
        # Efficiently clears the entire list.
        self.head = None
        self.tail = None
        self.length = 0
```

However, while this is an $O(1)$ operation in terms of the variables we manipulate, the total work by the Python interpreter would be $O(n)$ steps in terms of time complexity, because of the steps taken in the cascading garbage collection. This might be more efficient for small lists because of deferred garbage collection, it might not free the whole thing at once, and would clean up during background collection when the CPU isn't as busy. In the case of a large list, it may actually be better to explicitly remove each node through list traversal. This is because the garbage collector in Python can have a recursion limit that prevents full removal of long series of connected objects like a massive linked list.

```
def clear(self):
        # Manually breaks links to prevent stack overflow on huge lists.
        current = self.head
        while current:
            next_node = current.next
            # Break the reference to the next node
            current.next = None
            # If it's a doubly linked list:
            # current.prev = None

            # Move to the next node
            current = next_node

        self.head = None
        self.tail = None
```

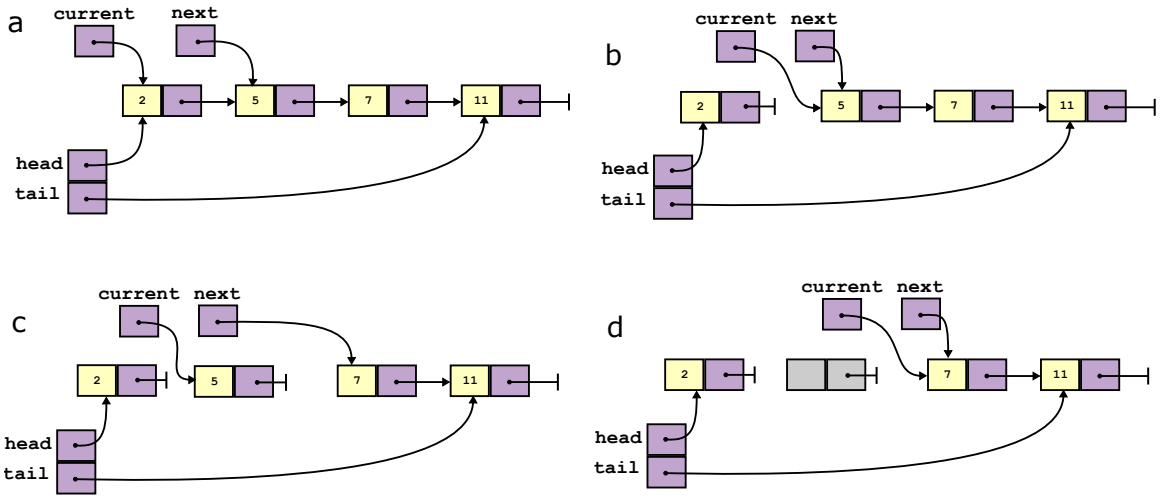


Figure 2.6: A representation of freeing the memory of a linked list. The head node is not freed until the last step, so technically the second node is freed first.

2.9.4 Time Complexity of Appending to a Linked List: `insert_last`

A function for appending to a linked list would be given a value for the new node, and then add this node to the end of the list.

```
def insert_last(self, value: Any) -> None:
    # adds a new node with the provided value
    new_node = Node(value)

    # Case 1: Linked List is empty
    if not self.head:
        self.head = new_node
        self.tail = new_node
        return

    # Case 2: Linked List has items. Point the current tail to the new node, then move the tail poi
    self.tail.next = new_node
    self.tail = new_node
    self.length += 1
```

We can visualize the action of this function with Figure 2.7. This can be thought of as a multi-step process. First, we create the new node and add its value or data. Next, if the list is not empty, we would make the pointer to our new node the value stored in the `next` variable of the node pointed to by `tail`. Lastly, we would update our `tail` pointer to point to the new node. We would also need to update the `length` variable to account for the new node.

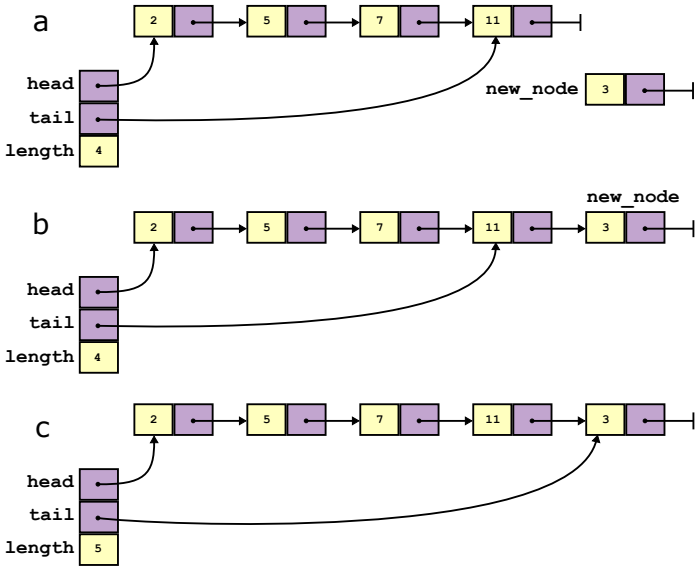


Figure 2.7: A representation appending a node into a linked list. **a.** In the first step, we have our original linked list, and a new node. **b.** Using the pointer to the last node, we can make the `next` pointer of the last node point to our new node. **c.** In the last step, we can update our `tail` pointer to point to our new node.

2.9.5 Time Complexity of `insert_first` with a Linked List

One of the advantages of a linked list is the efficiency with which nodes can be inserted. Insertion to an arbitrary location takes $O(1)$ time for a linked list assuming we have a pointer to the desired location, while it would take $O(n)$ time for an ordinary dynamic array due to the need to shift all elements over by one to make room for the inserted value. However, for a linked list it also takes $O(n)$ time to find the location to insert the value.

Prepending a value, which is to say adding a value with `insert_first`, is a special case of this that illustrates the point. For a linked list, this can be completed in one step. A function for `insert_first` with a linked list implemented in Python would look something like this:

```
def insert_first(self, value: Any) -> None:
    new_node = Node(value)
    new_node.next = self.head
    self.head = new_node

    # if linked list is empty
    if not self.tail:
        self.tail = self.head

    self.length += 1
```

We can visualize our procedure for prepending to a linked list as process shown in Figure 2.8. We first define our new node. We next need to point the `next` pointer of our new node to the same node that `head` points to. In other words, we point `new_node.next` to `head`. Lastly, we update our `head` pointer to point to our new node.

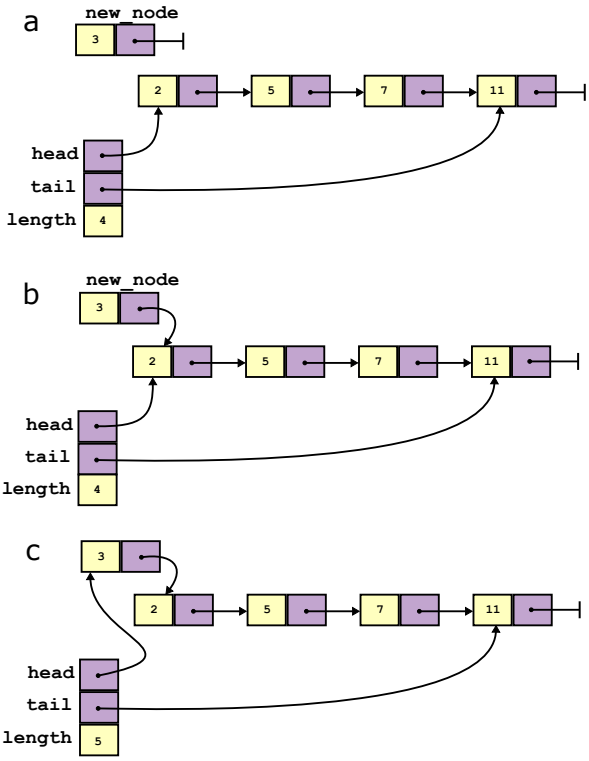


Figure 2.8: A representation prepending a node into a linked list. **a.** In this first step, we begin with our original linked list and a new node. **b.** Next, we make the `next` pointer of our new node point to the current `head` node. **c.** Finally, we update our `head` pointer to have the value of the pointer to the new node.

2.9.6 Time Complexity of Insertion into a Linked List

More generally speaking, a node can be inserted into an arbitrary position very efficiently. The problem with linked lists is we need to find the location to put the node. If we do an insertion operation by index, defined by the number of nodes in the linked list, we would need to compute some work with average time $T_{avg}(n) = n/2$ in order to find the place to put the node, while the work to insert the node itself is $O(1)$ once

we have that location. The process of insertion is similar to prepending, but we carry out this action relative to another pointer in our list, rather than relative to `head`. This process can be visualized as a similar series of steps depicted in Figure 2.9.

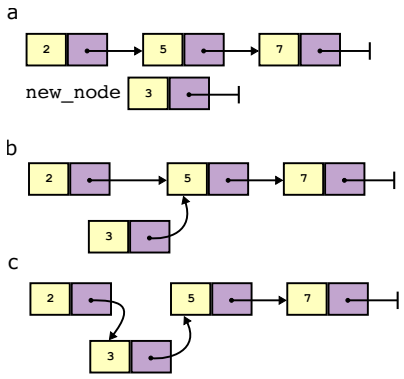


Figure 2.9: A representation inserting a node into a linked list. This does not depict the work to find the place to put the new node.

We can implement this procedure with the following code that separates the two actions of finding the index and the actual insertion:

```
def insert_after_node(self, prev_node: Node, value: int) -> None:
    # O(1) complexity: Links a new node and increments length
    new_node = Node(value)
    new_node.next = prev_node.next
    prev_node.next = new_node
    self.length += 1

def insert_at_index(self, index: int, value: int) -> None:
    # O(n) logic: Reuses insert_after_node for index-based insertion.
    # Quick O(1) bounds check using our length attribute
    if index < 0 or index > self.length:
        raise IndexError("Index out of bounds")

    # Special case: Insert at head
    if index == 0:
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node
        self.length += 1
        return

    # Traverse to the node right BEFORE the target index
    current: Optional[Node] = self.head
    for _ in range(index - 1):
        if current:
            current = current.next

    # Use our helper to perform the insertion and update length
    if current:
        self.insert_after_node(current, value)
```

2.9.7 Time Complexity of Reversing a Linked List

Reversing a linked list can be computed in $O(n)$. With this type of list reversal, we would want to start with a situation as presented in Figure 2.10 and achieve the same outcome.

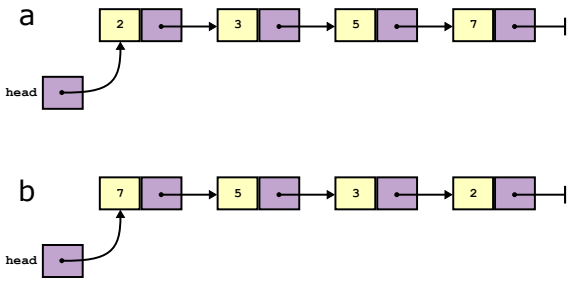


Figure 2.10: A representation of the desired outcome of reversing a linked list.

To compute the reversed list, we need to carefully update each node without "letting go" of any node. In other words, we must maintain a pointer to each node throughout this process.

First, we begin by defining three nodes, or more precisely, we keep track of pointers to the three nodes `prev`, `current`, and `next`. We will iterate these three nodes across our list.

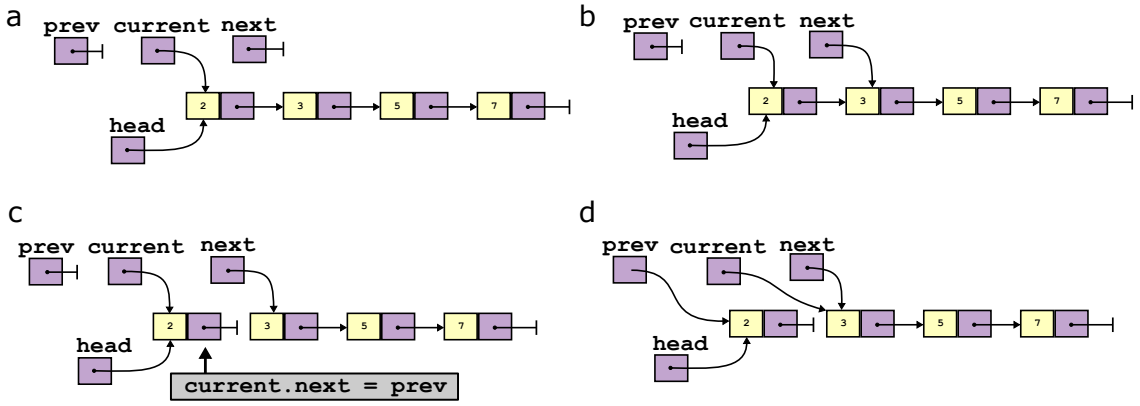


Figure 2.11: A representation of the first steps of reversing a linked list. **a.** Initially, we just define the nodes with `current` pointing to `head` and the others `None`. **b.** Next we update our `next` pointer to point to `current->next`. **c.** Next we update our `current->next` to be whatever `prev` was pointing to. **d.** Then we update our `previous` pointer to have the value of whatever `current` was pointing to, and then update`current` pointer to have the value of whatever `next` was pointing to. This final update moves our pointers further down the list.

Initially, two of the three pointers are `None`, but we update them as we go. The same operations are described in Figure 2.11 as in Figure 2.12, but more action happens as it moves further across the list.

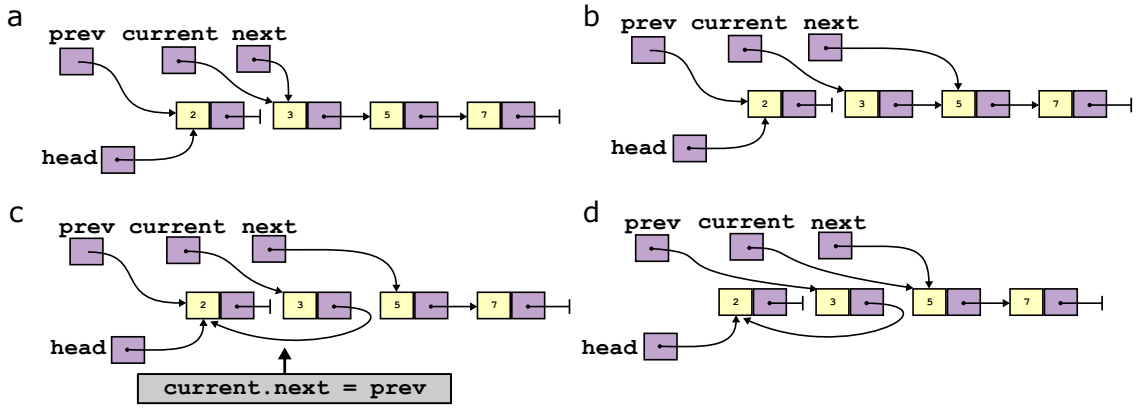


Figure 2.12: A representation of the next steps of reversing a linked list. **a.** Initially, our nodes are the same place as they ended in Figure 2.11. **b.** Next we update our `next` pointer to point to `current->next`. **c.** Next we update our `current->next` to be whatever `prev` was pointing to. **d.** Then we update our `previous` pointer to have the value of whatever `current` was pointing to, and then update`current` pointer to have the value of whatever `next` was pointing to. This final update moves our pointers further down the list.

This process is repeated until the end of the list, resulting in the desired reversed linked list. Putting this all together we get the following function:

```
def reverse(self):
    # O(n) time complexity, O(1) space complexity.
    prev = None
    current = self.head

    while current:
        # 1. Save the next node (so we don't lose the rest of the list)
        next = current.next

        # 2. Reverse the pointer
        current.next = prev

        # 3. Move 'previous' and 'current' one step forward
        prev = current
        current = next

    # 4. Update the head to the last node we processed
    self.head = prev
```

2.9.8 Summary of the Time Complexity of Linked Lists

Single operations for a list of size n:

Operation	Time Complexity
create()	$O(1)$
length()	$O(1)$*
get(i)	$O(n)$
set(i,x)	$O(n)$
insert_first()	$O(1)$
insert_last()	$O(1)$
insert_at()	$O(n)$
delete_first()	$O(1)$
delete_last()	$O(n)$
delete_at()	$O(n)$
resize()	Not applicable, no resize needed!
copy()	$O(n)$, but don't often need to, no resize

Table 2.2: A summary of the work for basic tasks applied to a linked list. *Assuming a length attribute.

If we were to grow a linked list of length n, we would get an average of similar to the individual case, because most cases are $O(1)$.

Note that there is no method for resizing a linked list. You could say that the resize is $O(1)$ if you consider the addition of each node a resize. However, the meaning here is that the linked list does not allocate a block of memory to store the data, hence no such method exists for the linked list.

Operation	Time Complexity
Time Average for Append	$O(1)$
Time Average for Insert	$O(1)$

Chapter 3

Stacks and Queues

In many real-world applications, we have restrictions on our lists. For example, software for graphic design would want to store edits to drawings in the order that they are made, so that "undo" undoes them in order. In other situations, like a software application that processes orders may want to ensure that the orders are processed in the order that they are received. These two ideas are motivation for stacks and queues.

3.1 Stacks

A stack is a special case of a list, and is an abstract data type like a list. A stack is an abstract idea with defined rules of operation.

We can imagine this idea as a stack of books or a stack of plates. We can add plates to the top of the stack, we can retrieve plates from the top of the stack, but never to or from the bottom of the stack. When you take a dinner plate from the cupboard, you are always taking the plate that was most recently put away! This restriction placed on stacks can be thought of as **last in first out (LIFO)**.

Figure 3.1: A stack can be thought of as a stack of plates. You would never want to add or retrieve a plate form the bottom of the stack!

3.1.1 Stacks as an ADT

Since a stack is a special case of a list, it is also an ADT, and can be defined as a sequence of elements $L = e_1, \ldots, e_n$.

For the stack ADT, we have two primary operations, `push()` to add to the top of the stack, and and *pop*() to remove from the top of the stack and return the value. In addition, a `peek()` operation is also common to view the element at the top of the stack without removing it.

The *push*() operation would take the list L and an new element x. To add to the top of the stack, such that an expression like `L.push(x)` would result in an updated list $L = x, e_1, \ldots, e_n, x$

Similarly, the `pop()` operation would be defined such that $e_n = pop(L)$

and L would be modified to become $L = e_2, e_3, \ldots, e_{n-1}$ with its top term removed.

3.1.2 A simple idea of a stack

Let's consider a stack of blocks, and we'll also consider two operations, `push()` and `pop()`.

- `push(stack,block)` will add a block to the top of the stack
- `block=pop(stack)` will remove a block from the top of the stack

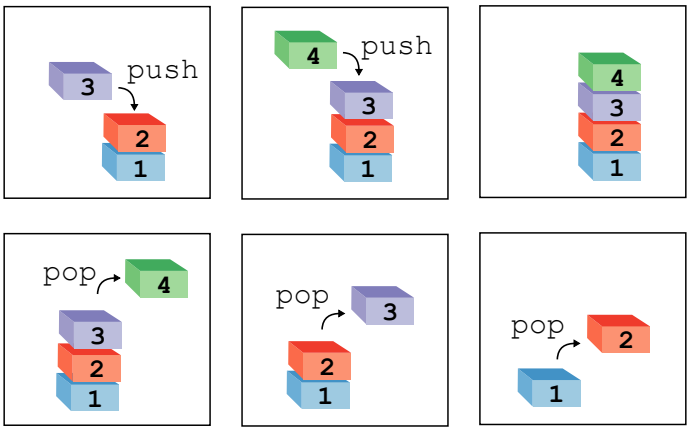


Figure 3.2: A stack of blocks with push and pop visualized

There are several examples of stacks in everyday life. A word processor application has undo/redo functionality such that each action (typing words, formatting) is a block on the stack, the "undo" operation pops from this stack, redo pushes it back on the stack. A web browser history is a stack such that the currently viewed webpage is the top of the stack, Clicking a link pushes to the stack, back button pops from the stack, and the forward button would push back the last visited webpage to the top of the stack.

3.1.3 Stacks Implemented as Linked Lists

Since a stack is an abstract idea, we'll need to implement it programmatically using one of our data structures.

One such approach to implement as a linked list. In this model, the first term of the linked list, which we have represented as the `head` pointer, is the top of the stack. The `tail` pointer refers to the element of the list at the bottom of the stack.

The reasoning for this choice of the top of the stack corresponding to the head node, as opposed to implementing the top of the stack as the tail node, is because the operation of removing the last node for a singly linked list is actually an $O(n)$ operation, because to update the tail pointer requires traversing the nodes of the linked list. Additionally, in many cases it is more intuitive to work with `head` as the main interface to the list. Since we never add or remove from the `tail` term, we can simplify and only visualize with the `head` pointer, but assume the `tail` pointer is also there. That being said, this is not an issue with a doubly linked list, which can remove elements from both ends of the chain with $O(1)$ cost.

Stack Push Implemented as `insert_first()` for a Linked List

Building off of our previously defined `Node` class, we can define a stack and a method for prepending to a linked list to define the push operation like this:

```
class Stack:
    def __init__(self):
        self.top = None  # Initially, the stack is empty

    def push(self, value):
        # 1. Create a new node (equivalent to create_node(value))
        new_node = Node(value)

        # 2. Push to the stack: point new node's next to current top
        new_node.next = self.top

        # 3. Make new node the new stack top
        self.top = new_node
```

Note that for simplicity, we have used the term `top` to refer to what is the top of the stack, but corresponds to the head node. For simplicity, we have removed access to the tail node, which would correspond to the bottom of the stack, which we should not normally have access to anyway. Furthermore, we do not have a `count` or `total` attribute to store the number of elements on the stack, but this could be easily added and might be a good idea, because accessing the number of elements would be $O(n)$ work. Similarly, we can also define the pop and peek operation as follows:

```
def pop(self):
        if self.top is None:
            return "Stack is empty"

        popped_value = self.top.value
        self.top = self.top.next  # Move the top pointer to the next node
        return popped_value

    def peek(self):
        return self.top.value if self.top else "Stack is empty"
```

Note that we are using Python's garbage collection to remove the node, and returning the value stored in that node. By updating the `top` pointer to what was the second node, the reference count to the original top node goes to zero, and the element should be cleared from memory.

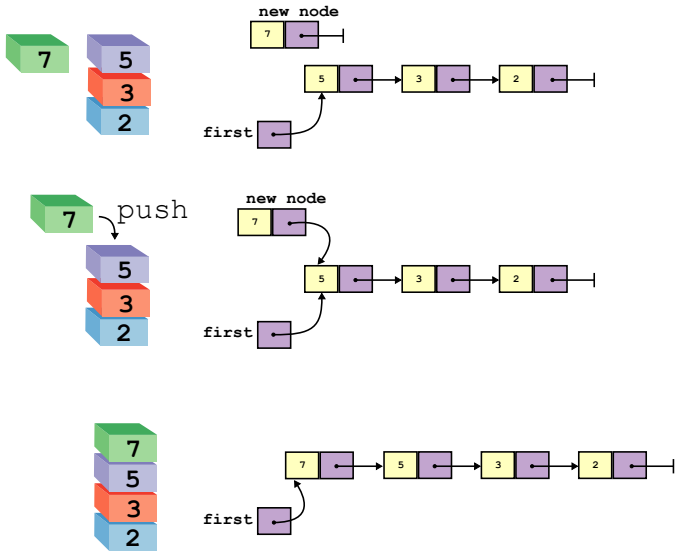


Figure 3.3: A representation of pushing to a stack as analogously prepending a node into a linked list.

3.1.4 Stacks Implemented as Dynamic Arrays

We could also implement our stack as a dynamic array. In this model, the last element of the array could be viewed as the top of the stack.

Under this model, the `push` operation could be implemented as adding a value to the last term of the array. This operation would potentially involved resizing the array.

The `pop` operation would be implemented as removing the last term of the array and returning it as a value.

Push as appending to a dynamic array

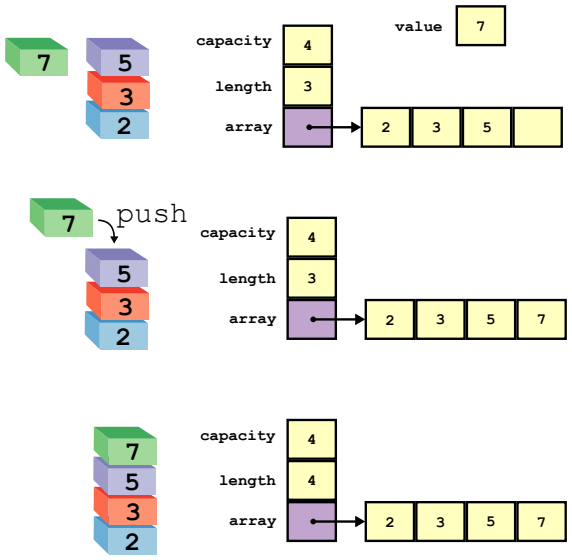


Figure 3.4: A representation of pushing to a stack as adding to a dynamic array.

If we were to implement this in Python using our Dynamic Array class, we would define the push operation as follows:

```
import ctypes

class Stack:
    def __init__(self, dtype=ctypes.c_int):
        self._array = DynamicArray(dtype)

    def push(self, value):
        # will call _resize if full, then add to end
        self._array.insert_last(value)

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty stack")

        # Access the last element
        top_val = self._array._data[self._array._length - 1]

        # Decrease length (removing it from the stack view)
        self._array._length -= 1

        # Return the top value
        return top_val

    def peek(self):
        if self.is_empty():
            return None
        return self._array._data[self._array._length - 1]

    def is_empty(self):
        return self._array._length == 0
```

One surprising piece of this implementation is that it does not actually remove the top element and only decrements the length of the underlying dynamic array. However, this is fine because any push operations after this will overwrite the value, and any other pop operations will access the correct value because the `length` is used to access it.

3.2 Queues

A queue is a special case of a list, and is an ADT like a list. When working with a queue, new items are added, and items that are removed are always the first item to have been added to the queue. Like a stack, a queue is a special case of list because a restriction is placed on the queue so that you can only add items to the back of the queue, and remove from the front of the queue. In other words, the restriction is **first in first out (FIFO)**.

Here are some examples of queues in everyday life:

- **Store deliveries** Items are delivered in the order in which they are received.
- **Waiting line** such as a telephone customer service line. “your call will be answered in the order it was received.”
- **Short-order cook at a restaurant** orders are made as they come in

3.2.1 Queues as an ADT

The queue ADT is defined as a sequence of elements $L = e_1, \ldots, e_n$.

For queues, we have two main operations `enqueue()` to add to the back of the queue, and `dequeue()` to remove from the front of the queue.

For the `enqueue()` operation, we would have something like this:

$$e_1, \ldots, e_n, x = enqueue(L, x)$$

So that it adds to the back of the queue. Under this system, the first term is the first item in the series. The `dequeue()` operation would remove and return the first element of the queue such that $e_1 = dequeue(L)$, and the list is updated to be $L = e_2, e_3, \ldots, e_n$.

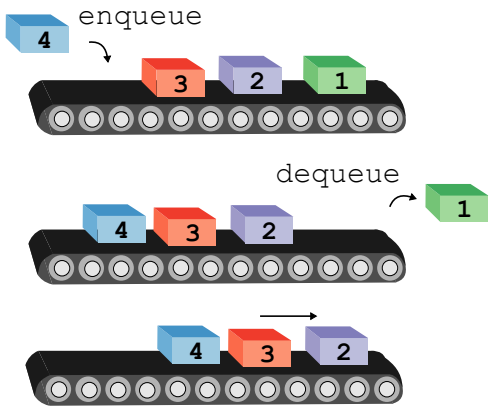


Figure 3.5: A queue can be thought of blocks on a conveyer belt. As items are removed, firstst added item will be the first out.

Let's consider a queue of blocks, and we'll also consider two primary operations, `enqueue()` and `dequeue()`.

- `enqueue(queue,block)` will add a block to the back of the queue
- `block=dequeue(queue)` will remove a block from the front of the queue

3.2.2 Implementing a Queue with a Linked List

Since a queue is an abstract idea, we'll need to implement it programmatically using one of our data structures, and we will first consider linked lists and dynamic arrays as our implementations. For the linked list, we must first define what the back and front of the queue will be.

The only good option for a singly linked list is that the tail node will correspond to the back of the queue, and the head node will correspond to the front of the queue. This method provides $O(1)$ work to perform the enqueue and dequeue operations.

If we implemented the other way, with the head node corresponding to the back of the queue and the tail node corresponding to the front, we would incur some added costs. While the enqueue operation is efficient, with $O(1)$ work to add a node, the dequeue operation requires $O(n)$ work because there is no easy way with a singly linked list to update the tail pointer and set the second to last node's next pointer to `None` without traversing the list to access this node again.

3.2.3 Implementing a Queue with a Dynamic Arrays

For a dynamic array, we quickly realize that it is not an ideal candidate for implementing a queue. While the enqueue operation is efficient, the dequeue operation is costly. For the enqueue, we can treat the last element as the back of the queue, and we can add elements in $O(1)$ time with amortized time complexity.

However, the dequeue operation will be involve something like `delete_first()` to remove the first element after, which will require $O(n)$ work to shift all the other values over by one.

We actually do not get more efficient doing the other way, because defining the first element of the dynamic array to be the back of the queue will result in similar $O(n)$ shifting cost to enqueue new elements. The conclusion here is that the linked list performs the fundamental operations of the queue more efficiently.

3.3 Implementing a Queue Using Two Stacks

Stacks and Queues are two completely different paradigms, so we would expect that implementing one using the machinery of the other would be counterproductive. However, this exercise can help illustrate the limitations of each method in other contexts. A point of clarification here, recognizing that both queues and stacks are ADTs, we cannot "implement" one with the other as they are both abstract ideas. More precisely, we will be using the implementations for stacks to implement a queue and vice versa, but will describe them more succinctly with "implementing a queue with two stacks."

First, to implement a queue using stacks, we need to define the implementation with two stacks.

```
class QueueFromStacks:
    """
    FIFO queue implemented using two LIFO stacks:
    - s1 collects enqueued elements,
    - s2 serves dequeues/fronts by reversing s1 when needed.
    """

    def __init__(self) -> None:
        # initialize the queue from stacks
        self.s1 = Stack()
        self.s2 = Stack()
```

As noted in the comments, one stack will serve as the back of the queue, and the other will serve as the front. We will enqueue by pushing onto stack 1, and dequeue by removing from stack 2. We will of course need to develop a way to move elements from one stack to another and maintain the correct ordering.

3.3.1 Implementing `enqueue()` with stacks

Under this two-stack implementation, when you add to a queue, you would push to the top of the first stack `s1`. So in this framework, the first stack represents the back of the queue.

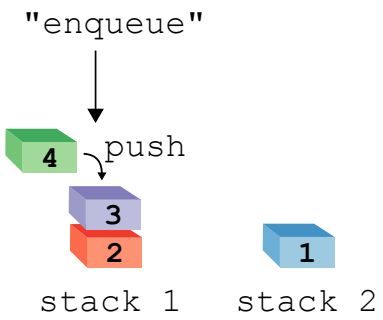


Figure 3.6: We can think of stack 1 as the back of the queue, so `push()` on stack 1 corresponds to `enqueue`

3.3.2 Implementing `dequeue()` with stacks

Under this two-stack implementation, the front of the queue is stored in `s2`. When you want something from the front of the queue, you would return the top of stack 2 using a `pop` operation on this stack.

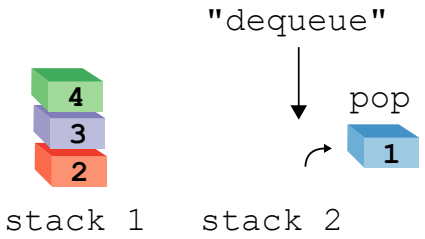


Figure 3.7: We can think of stack 2 as the front of the queue, so that `pop()` on stack 2 corresponds to `dequeue()`

But what if stack 2 is empty? You could pop the terms off of stack 1 and push them to stack 2.

The key idea here is that the terms on stack 2 need to be in the reverse order from stack 1. Meaning, the most recently added block from stack 1 needs to be at the bottom of stack 2. To achieve this, when stack 2 is empty we need to move the back of our queue, which is represented by stack 1, forward to the front. You would need to loop through and pop each block of stack 1, and push all of them onto stack 2 one at a time. Programmatically this can be achieved with a while-loop.

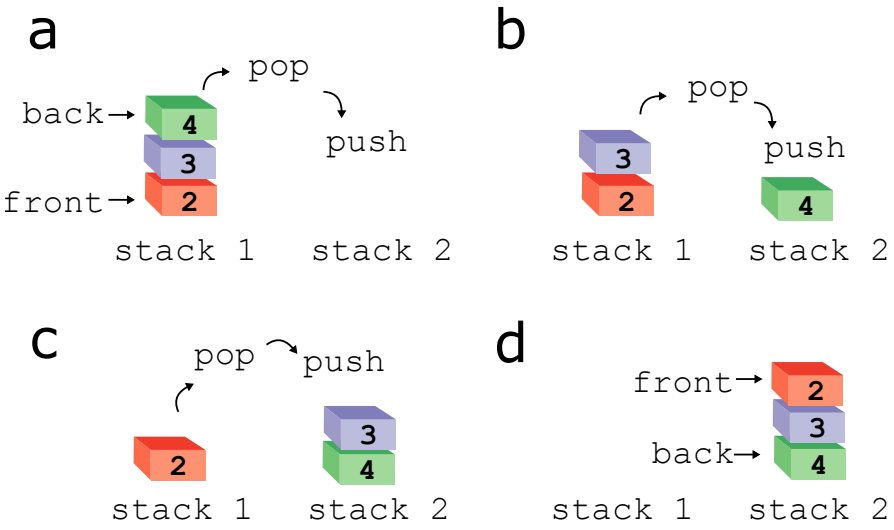


Figure 3.8: When the stack 2 is empty, we loop through stack 1, and transfer to stack 2. **a.** pop 4 off the top of `s1` and push onto `s2` **b.** pop 3 off the top of `s1` and push onto `s2` **c.** pop 2 off the top of `s1` and push onto `s2` **d.** At this point, after `s1` is empty, `s2` is now non-empty and the top of `s2` corresponds to the front of the queue.

The dequeue operation using two stacks will incur some work that is $O(1)$ in the best case, and $O(n)$ in the worst-case scenario.

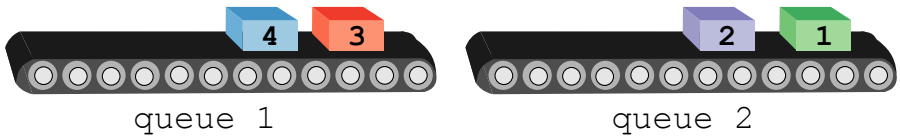
3.4 Implementing a Stack Using Two Queues

Similar to before, we will use two queues to implement a stack. We could create a class similar to before like so:

```
class StackFromQueues:
    """
    A LIFO stack implemented using two FIFO queues.
```

```
    At any time, one of q1 or q2 holds all the elements; the other is empty.
    Push appends to the non-empty queue. Pop to remove the last-pushed value.
    """

    def __init__(self) -> None:
        # initialize the stack from queues
        self.q1 = Queue()
        self.q2 = Queue()
```

We will always need to keep all the items in the order in which they were added, but we can use the two queues to pass values back and forth to (1) keep them in order and (2) return the most recently added item (the top of the "stack").



The top of our stack can be modeled dynamically, so that it switches between our two queues. We can consider the last item added to queue 2 as our default stack top, except when stack 2 is empty. This might be easier to explain this in terms of `push()`, so let's do that.

3.4.1 Implementing Stack `push()` with Two Queues.

Under this implementation, we would push to the top of the stack by using the `enqueue` operation. We can think of the last element added to queue 2 as our stack top, but when it is empty, we can think of the last element added to queue 1 as our stack top. This way, when queue 2 is empty, we can `enqueue()` to `q1` to push to the top of the stack. When `q2` is not empty, we can `enqueue()` to it to push to the top of the stack.

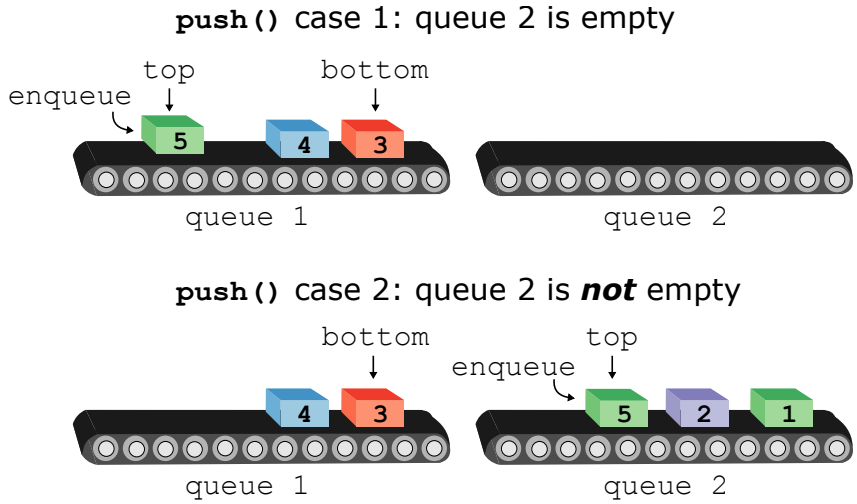


Figure 3.9: We can implement `push()` by considering the stack top to vary between the two queues. When queue 2 is empty, we can think of `enqueue()` to queue 1 as `push()` to the stack top, and when queue 2 is not empty, we can consider `enqueue()` to queue 2 as the `push()` to thde top of the stack.

3.4.2 Implementing stack `pop()` with Two Queues.

It is perhaps more difficult to implement `pop()` with two queues. The problem is that as noted before, the top of the stack is in some sense the least accessible element in our queues. Queues are built to have the first item added the most accessible. Therefore, to get to the top of the stack, we need to move items off the queue, and move them to the other queue.

Because we also considered two possible stack tops, we need two scenarios. Let's first consider when `q2` is not empty, since it is our default stack top. We would need to `dequeue()` items from `q2` and `enqueue()` them to `q1` one at a time.

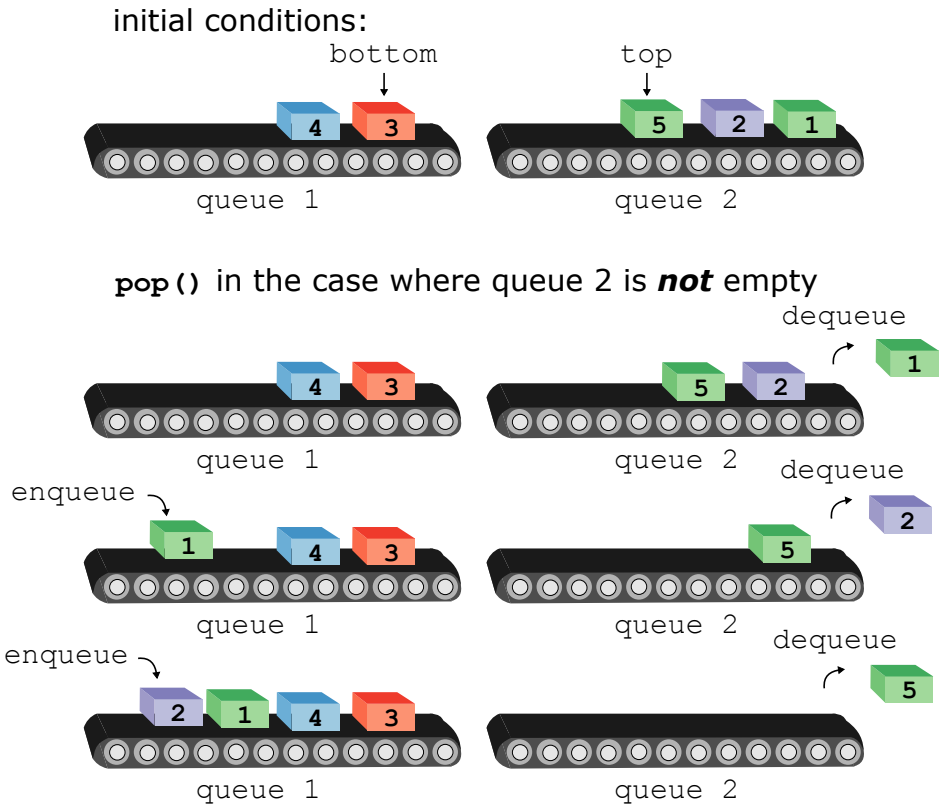


Figure 3.10: When `q2` is not empty, the last element to be added to it is the top of the stack. Therefore, we need to `dequeue()` each element and `enqueue()` them to `q1` one at a time. After all the elements on `q2` have been removed, the last element is the desired stack top.

In the other case, when `q2` is empty, we want the last item added to `q1`, which corresponds to the back of that queue. To get this item, we would carry out a similar procedure, by looping through the elements of `q1`. We would need to `dequeue()` items from `q1` one at a time, and `enqueue()` them to `q2`. The last element dequeued from `q1` would be the top of our stack and we would return that value.

3.5 Circular Arrays

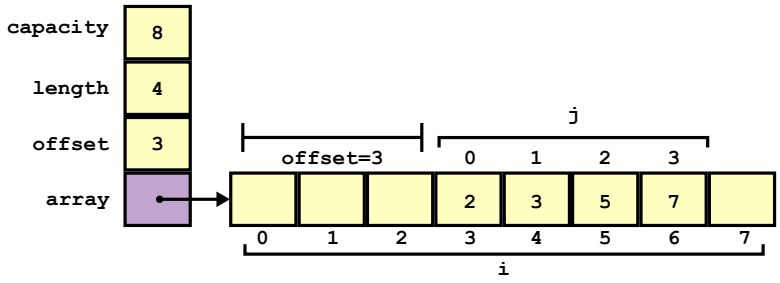
A key limitation of using dynamic arrays to implement queues was the fact that we would have to shift over values by one every time we `dequeue()` from the queue.

One solution is called a **circular array**. In a circular array, allow the data in our array to shift around, and not necessarily all be at the beginning of the array. We could keep track of an additional variable `offset` to account for this shift. We also allow the values of the array to be circular, so that when the go over the edge, they are placed on the other side.

Our declaration might look something like this.

```
import ctypes
from typing import Any, Type, List

class CircularArray:
    def __init__(self, capacity: int = 1, dtype: Type[Any] = ctypes.c_int) -> None:
        self._length: int = 0
        self._capacity: int = capacity
        self._offset: int = 0  # Represents the 'front' of the array
        self._dtype: Type[Any] = dtype

        # Initial allocation
        self._ArrayType = self._dtype * self._capacity
        self._data = self._ArrayType()

    def insert_last(self, value: Any) -> None:
        if self._length == self._capacity:
            self._resize(2 * self._capacity)

        # The i index is (offset + length) % capacity
        i = (self._offset + self._length) % self._capacity
        self._data[i] = value
        self._length += 1

    def remove_first(self) -> Any:
        if self._length == 0:
            raise IndexError("Cannot remove from an empty array")

        value = self._data[self._offset]

        # Logical removal: move the offset forward
        self._offset = (self._offset + 1) % self._capacity
        self._length -= 1

        return value
```

Note that we have not defined the `_resize()` method yet. We'll come back to that because the circular array demands special consideration. Before we get there, let's visualize the role of the offset.



In this example, we have defined an additional index `j` to correspond to the shifted index relative to the array indices `i`. You can think of `j` as the shifting index of the circular array, and it is different from the original index of the array we allocated, represented by `i`. To get the actual index in the stored array:

```
i = offset + j
```

This isn't the complete equation, just the beginning of the idea, because what do we do when `offset+j` exceeds the capacity of the array?

3.5.1 Circular Buffers

The circular nature of the array, means we could add and remove elements from the beginning and end of the array more efficiently than a traditional dynamic array, and would not require "shifting". As long as we have the capacity, circular buffers means that values that go over the edge of the circular array will be placed at available positions at the beginning. This fact that the elements can wrap around the array is what makes it "circular".

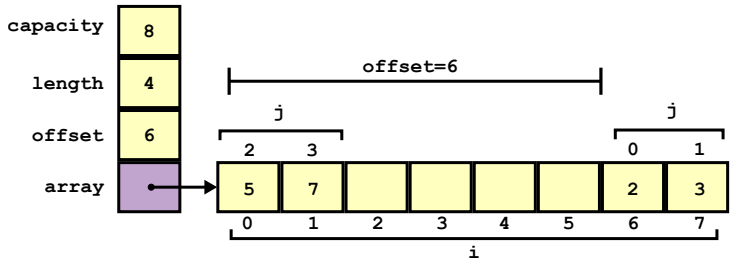


Figure 3.11: A visualization of a circular array where values wrap around the end.

In this example, we still have the additional index `j` to correspond to the shifted index. However, these indices circle around the array indices `i`. This relationship can be described by the remainder operator:

```
i = ( offset + j ) % capacity
```

These indices are often called described as i being the "physical" index, and j being the "logical" index. Meaning, in terms of the interface you would access by j, such that the first element in memory has index $j = 0$, making it the logical index. However, the actual physical location of the element is stored at $i = 6$, making it the physical index.

3.5.2 Remainder operator

Generally speaking, the remainder operator `%` returns the remainder when dividing by a number. For example,

```
11 % 10 == 1
```

or

```
9 % 7 == 2
```

Going back to the previous example, we had `offset=6` and the index in our circular array is `j=3`, so we should have:

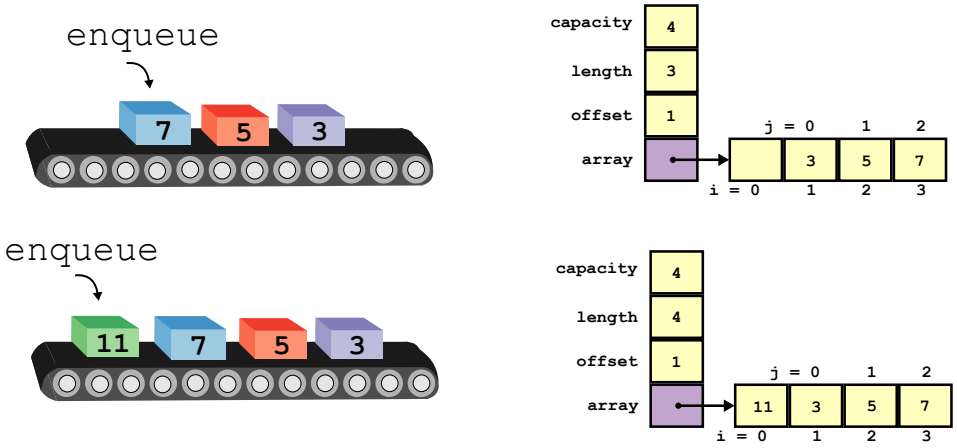
```
i = ( 6 + 3 ) % 8
i = 9 % 8
i = 1
```

We can see with this that using the remainder operator, the index that we need to access the desired element is $i = 1$, which corresponds to the Figure 3.11.

3.5.3 Implementing `enqueue()` and `dequeue()` Operations with a Circular Array

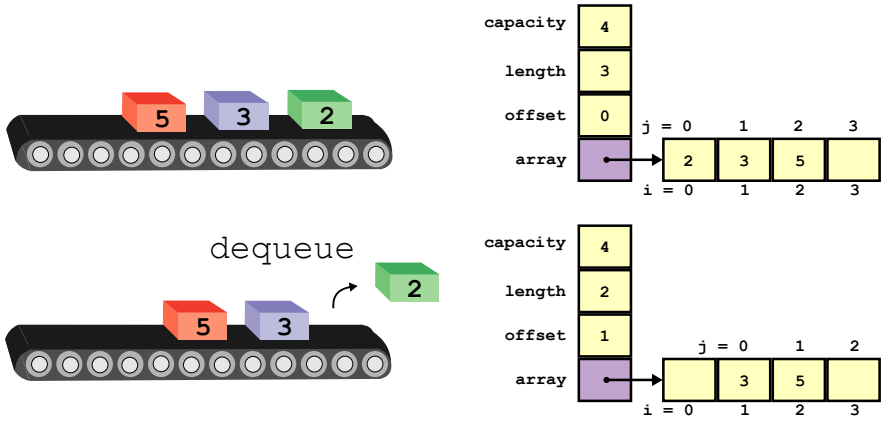
Our `enqueue()` operation might look like something like this:

```
def enqueue(self, value: Any) -> None:
        """Add an element to the back of the queue."""
        # This is logically identical to 'insert_last'
        if self._length == self._capacity:
            self._resize(2 * self._capacity)

        # Calculate the index i using the offset and current length
        i = (self._offset + self._length) % self._capacity
        self._data[i] = value
        self._length += 1
```



Our `dequeue()` operation might look like something like this:

```
def dequeue(self) -> Any:
        """Remove and return the element at the front (the offset)."""
        if self._length == 0:
            raise IndexError("Dequeue from an empty queue")

        # 1. Capture the value at the current offset
        value = self._data[self._offset]

        # 2. Optional: Clear the memory (C-style safety)
        self._data[self._offset] = self._dtype()

        # 3. Move the offset forward (wrap around if it hits capacity)
        self._offset = (self._offset + 1) % self._capacity

        # 4. Decrease the logical length
        self._length -= 1

        return value
```



3.5.4 Resizing and Reindexing Circular Arrays

Upon resizing our array (increasing the capacity) we could also shift all the values back to an offset of 0, since we will need to copy the values anyway. This reindexing procedure would produce a new index for each value.

Under this framework, whatever the original value of `j` was would be the new index `i` for the resized array.

But how would we write `j` in terms of `i` in a general way?

In this simplest case, with no wrap-around occurring, we would just have

```
j = i - offset
```

However, when this value is negative, we would need to shift it back to within our array indices $0, 1, ...,$ `capacity`. Therefore, we would use:

```
j = i - offset + capacity
```

Putting this into one expression would give:

```
if(i - offset > 0):
    j = i - offset
else:
    j = i - offset + capacity
```

However, given this clumsy equation for `j` in terms of `i`, it is easier to just use loops over `j`, the roaming index of the circular buffer. An example of how the resize function might work for this would be:

```
def _resize(self, new_capacity: int) -> None:
    # Create a new buffer
    NewArrayType = self._dtype * new_capacity
    new_data = NewArrayType()

    # 'Unroll' the circular data into the new buffer starting at index 0
    for j in range(self._length):
        i = (self._offset + j) % self._capacity
        new_data[j] = self._data[i]

    self._data = new_data
    self._capacity = new_capacity
    self._offset = 0 # Reset offset after straightening the data
```

In this example, we are looping over the values of the circular array, which we have defined as j, and storing the values into the new array using this index, rather than the original "physical" index i of the actual array, which compacts the value at the beginning of the physical array, and makes the `offset` equal to 0.

3.6 Deques: Double-ended Queues

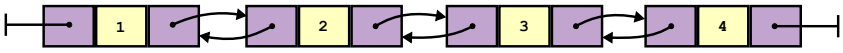
The word "deque" is an abbreviation of double-ended queue, and is pronounced "deck". The pronunciation makes me wonder if it is like a deck of cards where you can draw from either the top of the deck or the bottom of the deck.

Therefore, our deque implementation should consider the following operations:

1. `enqueue from front`
2. `enqueue from back`
3. `dequeue from front`
4. `dequeue from back`

3.6.1 Implementing a Deque with a Doubly Linked List

Let's consider a doubly linked list implementation of deques. This is a well-suited data structure because the pair of pointers allow us to interface nodes from the two sides of the deuque: front and back.



3.6.2 Front and Back Sentinels

A useful construct for working with this kind of doubly linked list is that of a sentinel.

A **sentinel** is a special node in our linked list that demarcate the boundaries of our linked list, and act as a "buffer" on the edges. The sentinel nodes might contain special values, or be specially defined nodes with their own `struct` declaration, such that the value is of a different data type.

One of the advantages of a sentinel is it provides a fixed front and back of the linked list. Without them, we would always have to update the `front` and `back` pointers every time we add to the front or add to the back respectively, and similar for remove of nodes.

As a comparison, here is the doubly linked list implementation of a deque without the sentinels:

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
        self.prev = None

class DoublyLinkedListDeque:
    def __init__(self):
        self.front = None
        self.back = None

    def is_empty(self):
        return self.front is None
```

We have set the first and last node to `None` to initialize the data structure. We would also need to define a set of functions for the `enqueue()` and `dequeue()` operations. The insert methods would be implemented by the following functions:

```
    def insert_front(self, value):
        node = Node(value)
        if self.is_empty():
            self.front = self.back = node
        else:
            node.next = self.front
            self.front.prev = node
            self.front = node

    def insert_back(self, value):
        node = Node(value)
        if self.is_empty():
            self.front = self.back = node
        else:
            node.prev = self.back
            self.back.next = node
            self.back = node
```

We can see there there is a similar structure in both cases. The `next` and `prev` pointers are updated for the new node, and the `front` and `back` sentinels are used to place the new node respectively. The common them here suggests the possibility of a helper function that could repeat these two tasks in the different contexts. Another notable piece is the boundary conditions that check if the doubly linked list is empty. Can those be simplified? The delete operations can be implemented as follows:

```
    def delete_front(self):
        if self.is_empty():
            raise IndexError("Empty")
        value = self.front.value
        self.front = self.front.next
        if self.front is None:
            self.back = None
        else:
            self.front.prev = None
        return value

    def delete_back(self):
        if self.is_empty():
            raise IndexError("Empty")
        value = self.back.value
        self.back = self.back.prev
        if self.back is None:
            self.front = None
        else:
            self.back.next = None
        return value
```

Now if we add the sentinels, we would have following initialization and basic functions, using the same `Node` class definition from before to initialize the sentinels instead of just `None` pointers:

```
class SentinelDeque:
    def __init__(self):
        # Create dummy nodes
        self.front = Node()
        self.back = Node()

        # Link them to each other
        self.front.next = self.back
        self.back.prev = self.front

    def is_empty(self):
        return self.front.next == self.back
```

The main difference is the `front` and `back` nodes are initialized to nodes that function as the sentinels. Furthermore, we initialize our doubly linked list with the next and prev pointers of our front and back sentinels pointing to each other, like so:

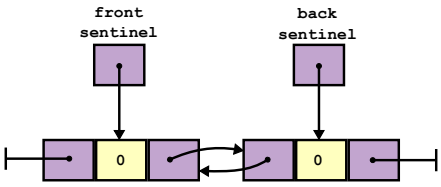


Figure 3.12: The starting point of a doubly linked list with front and back sentinel nodes.

This is the starting point above, but one we add nodes, have the following update:

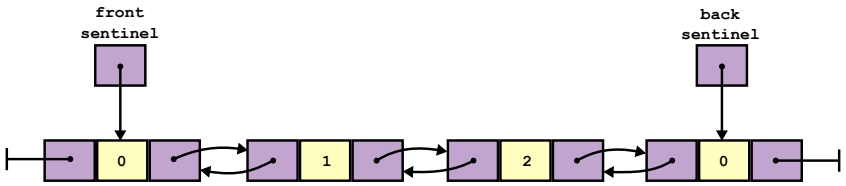


Figure 3.13: The doubly linked list with sentinel nodes and other nodes added.

The `insert` operations would be updated to the following set of functions that have room for improvement. Keep an eye on places where code can be consolidated into common functions:

```
def insert_front(self, value):
    node = Node(value)
    current_first = self.front.next

    node.next = current_first
    node.prev = self.front
    self.front.next = node
    current_first.prev = node

def insert_back(self, value):
    node = Node(value)
    current_last = self.back.prev

    node.next = self.back
    node.prev = current_last
    self.back.prev = node
    current_last.next = node
```

Note that the sentinels prevent the need for checking the edge cases of an empty linked list because with the sentinels the linked list is never totally empty. Also, note the parallel blocks of code that look like they could be simplified by helper functions. Essentially the same four pointers are updating, but with the contextual nodes being different. The delete operations are as follows:

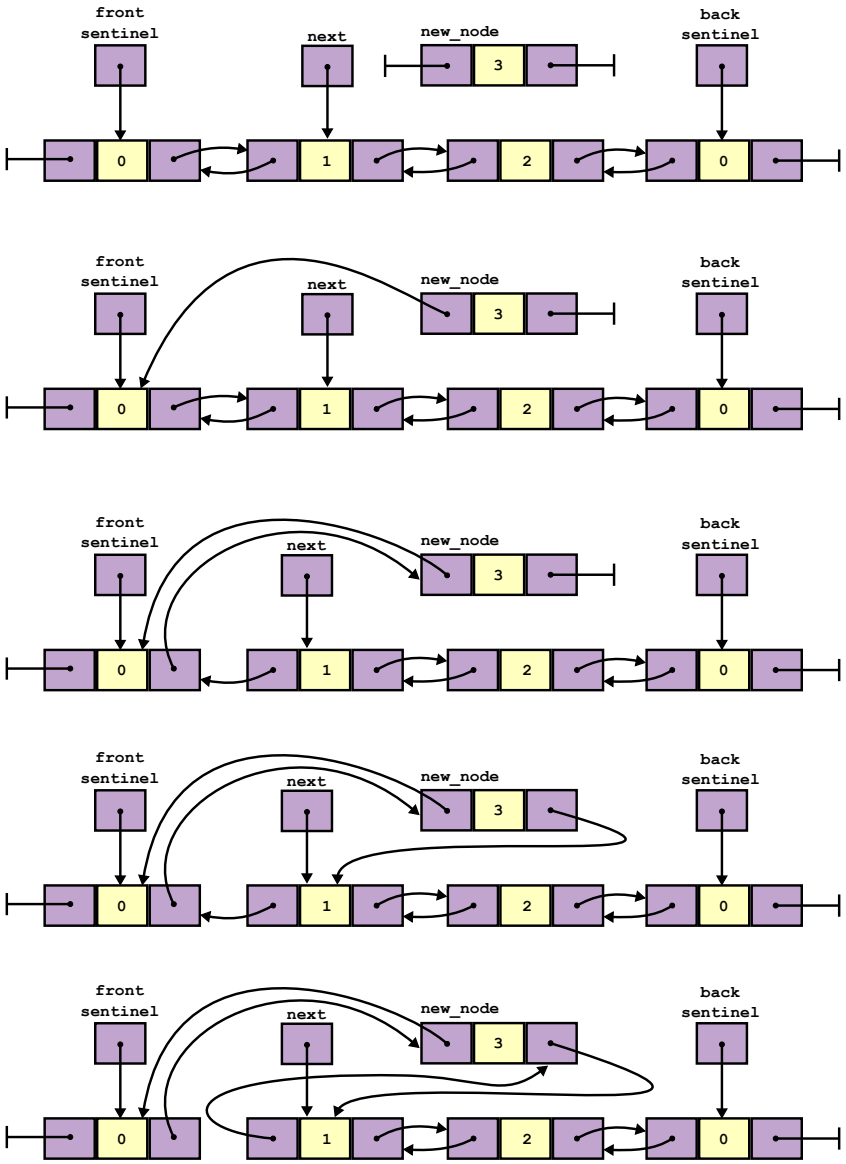
```
def delete_front(self):
    if self.is_empty():
        raise IndexError("Delete from empty deque")

    target = self.front.next
    self.front.next = target.next
    target.next.prev = self.front

    return target.value

def delete_back(self):
    if self.is_empty():
        raise IndexError("Delete from empty deque")

    target = self.back.prev
    self.back.prev = target.prev
    target.prev.next = self.back

    return target.value
```

There are still several similar code elements in the previous function declaration. Note the parallels between these new `delete` operations. A core improvement can be achieved by defining a helper function that can be reused.

First, let's define a helper function `_add_before()`

```
def _add_before(self, value, target_node):
        # Helper method: Inserts a new node directly before 'target_node'.
        node = Node(value)
        predecessor = target_node.prev

        # Set up the new node's links
        node.next = target_node
        node.prev = predecessor

        # Reroute existing neighbors to the new node
        predecessor.next = node
        target_node.prev = node
```

Let's look at this operation step-by-step:



Next, we can implement the operation `insert_front()` using our helper function `_add_before()`

```
def insert_front(self, value):
        # The node at the actual front is the one after the front sentinel
        self._add_before(value, self.front.next)
```

Similarly, we can implement the operation `insert_back()` with the same helper function:

```
def insert_back(self, value):
        # To insert at the back, we place it directly before the back sentinel
        self._add_before(value, self.back)
```

We can also define a helper function to remove a node from the list and return its value:

```
 def _remove(self, node):
        """Unlinks a node from the list and returns its value."""
        node.prev.next = node.next
        node.next.prev = node.prev
        return node.value
```

Next, we can implement the operation `delete_front()` using our helper function `_remove()`

```
def delete_front(self):
        if self.is_empty(): raise IndexError("Delete from empty deque")
        return self._remove(self.front.next)
```

Similarly, we can implement the operation `dequeue_back()`:

```
def delete_back(self):
        if self.is_empty(): raise IndexError("Delete from empty deque")
        return self._remove(self.back.prev)
```

The addition of the front and back sentinel nodes allows for greater reuse of code. We also simplify the code because we do not have to check the edge cases of the linked list being empty. By defining one `_add_before()` function, we can keep track of node removal in one location, such as with a counter to keep track of the size of the data structure, in one location of the class definition.

Chapter 4

Dictionaries

A **dictionary** (also called an **associative array**) is an abstract data type that is defined by a collection of keys and values. For our dictionary D we expect the following operations:

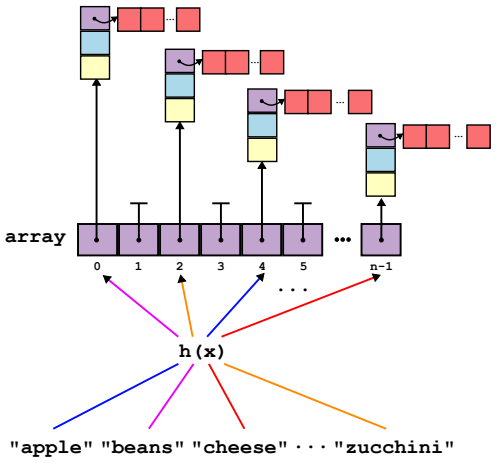
1. set a key-value pair: `D[key] = new_value`
2. remove a key-value pair: delete(D[key]) or `remove(D,key)`
3. lookup an existing key-value pair: `print(D[key])`

Dictionaries have several uses. First is the obvious case of storing data as key-value pairs, such as in a phone book, or glossary. Another important use is in memoization, which is is the act of storing the results of function calls as key-value pairs to a dictionary to save compute time later. Dictionaries can also be useful for counting. For instance, one can count the number of occurrences of a word using the word as the key and the count as the value.

How does one design a data structure to implement a dictionary that maintains data for these set, remove, and lookup operations? The two most common solutions to this are hash tables and search trees. For now, let's focus on hash tables because search trees have so many uses they deserve their own chapter.

4.1 Implementing Dictionaries with Hash Tables

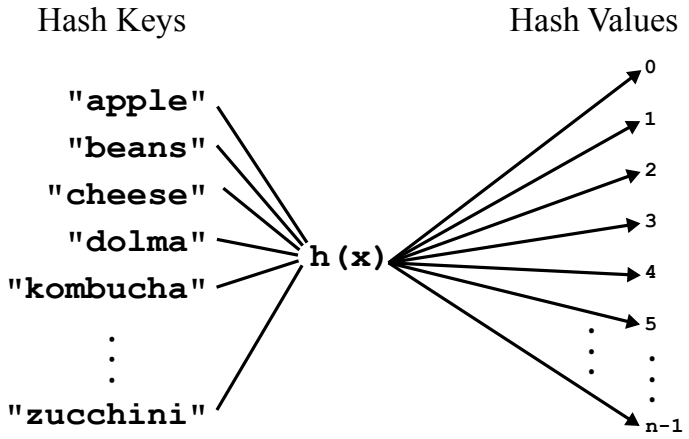
A `hash table` is a data structure that combines an array with a hash function. The hash function $h(x)$ takes a text string x as input, and maps it to an index of an array. For a hash table, depending on the hash function, the order of terms in the array need not correspond to the order of the inputs.



4.1.1 Hash Functions

A `hash function` is a function that takes in a string as input (for example) and returns a number of a fixed size.

For example, we might have the input be any string, and the value returned be an integer from 0 to $n-1$ for some n.



4.1.2 The virtues of a good hash function

The hash function should have several desired properties. The computations required to compute the hash function output should be efficient and computed in $O(1)$ time. It is expected that hash functions are have a deterministic, defined output, such that function reliably and consistently returns the same defined value for each distinct key. The output value returned by $h(x)$ should have a defined range of values, preferably in a range $0, 1, \ldots, n-1$ when using an array of size n. the output of the hash function should distribute evenly amongst these valid indices of the array. More strongly, we expect that the hash function behaves as well at distributing values as a randomized algorithm would. This last point can be called an assumption of **uniformity**, and it can allow us to model the indices output from our hash function are uniformly distributed random variables.

4.1.3 Hash function collisions

A **collision** for a hash function is a instance where two keys have the same value or array index. A uniform hash function that maps to n different values should have a probability of $\frac{1}{n}$ of maping to any particular value. A **perfect** hash function has no collisions. A **minimally perfect** hash function has not collisions for a set of keys equal to the number of available indices in our array.

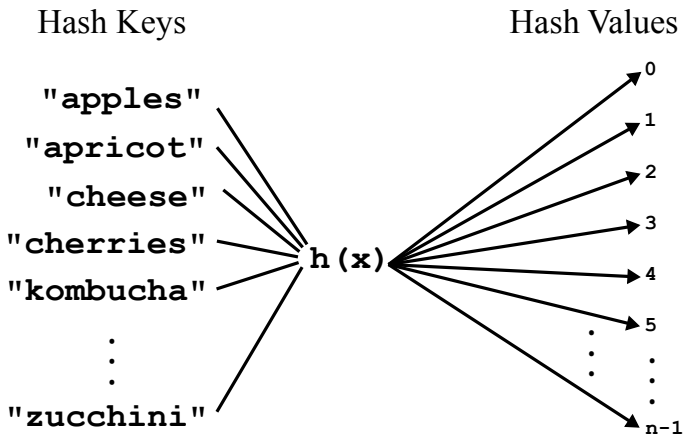
The output ranging from $0, \ldots, n-1$ reminds me of the output of the remainder operator %. So what about something like this:

```
h(x) = f(x) % n
```

As an example, consider the input as a string `x`, and we could just use the build-in C operator `(int)` on the first character to convert to the corresponding ASCII code for that character.

```
x: str = "apples"

def h(x: str) -> int:
    """
    Returns the integer ASCII value of the first character
    of string 'x' modulo 'n'.
    """
    return ord(x[0]) % n
```

This function wouldn't work in a case where two key words have a common first character.



As a second example, consider the input as a string `x`, and the value as the sum of the ascii codes:

```
x: str = "apples"

def h(x: str) -> int:
    sum: int = 0
    # Iterating over a str yields single-character strings
    for char in x:
        sum += ord(char)

    return sum % n
```

In this example, collisions would result for words that are anagrams of each other, such as "lemon" and "melon".

As a third example, consider the input as a string `x`, and we assign a different weight for each character by a power of 2, so that all strings map to a different value:

```
def h(x: str) -> int:
    sum: int = 0
    weight: int = 1

    for char in x:
        # sum over characters weighted by position
        sum += weight * ord(char)
        weight *= 2

    return sum % n
```

In this example, the weight for each position of the string is different, so it will resolve the anagram examples. However, there inevitably will be some collisions for this function, especially depending on how big n is. More advanced hash functions can operate at the level of bit-wise operations to produce an even more randomized output.

4.2 Hash Tables: Space and time.

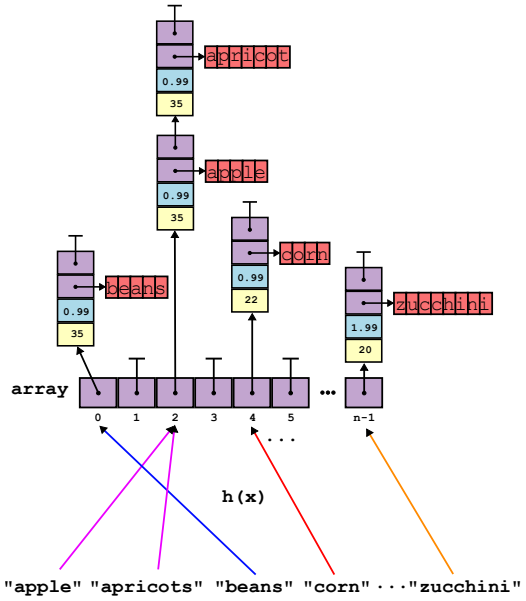
It turns out the key consideration for space and time complexity of hash tables is directly related to how collisions are resolved. The two most common methods for resolving collisions are **chaining** and **open addressing**. In many ways these two solutions have parallels to the two solutions to implementing a list that we saw in Chapter 2.7.

4.2.1 Resolving hash table collisions with "chaining"

The **chaining** method assigns multiple values for a given index, such as in the form of a linked list, to account for collisions.

The collections of possible values for a given index, such as the linked list, is called **buckets**.

When collisions occur, a new value is added to the linked list. During lookup, you would just walk across valid terms until you find the right term.



4.2.2 The load factor for a hash table using chaining

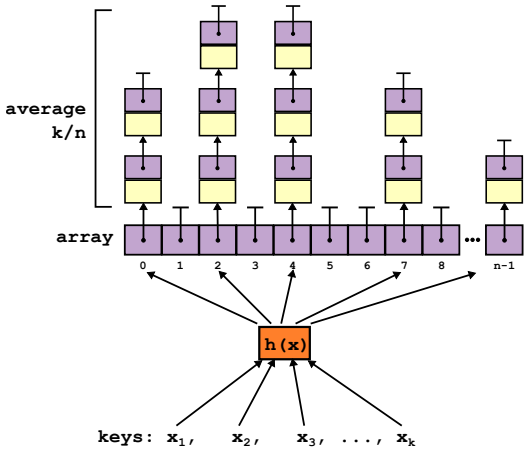
Consider storing k elements into hash table that uses an array of length n. We sometimes call the different terms of our array "buckets", so we have n buckets.

Then on average, the k elements would uniformly distribute amongst the n chains (buckets). Therefore, on average, there would be $\frac{k}{n}$ elements loaded into each of the n terms of the array.

The load factor λ describes the average number of elements in each term of the array.

$$\lambda = \frac{k}{n}$$

Simplifying our illustration here, we can imagine an average length of $\frac{k}{n}$ for our chains.



4.2.3 Hash table with chaining: worst-case time complexity

The time complexity for the worst case would be the situation with 100% collisions. We would essentially have a linked list. In the worst-case scenario we would have to traverse that linked list, which would be $O(k)$, the number of added key-value pairs.

4.2.4 Hash Table with chaining: average time complexity

We could also consider an average case time complexity, where we might expect to find the `key` we are searching for after traversing half of the list. The result will be a function of the load factor of the hash table. For example, consider the `D[key] = value` operation, that would assign the `value` with the provided `key`. In general, we would have to do a list traversal across the linked list. The call of the hash function itself would be constant time $O(1)$. Assuming we have to go half-way across each linked list on average, the average-case time complexity would involve $T(n) = \frac{\lambda}{2}$ steps, resulting in $O(\lambda)$.

4.2.5 Adjusting the load factor with Dynamic Hash Tables

Much like our dynamic array, we could also resize our hash table's array to decrease the load factor.

For example, doubling the length of the array n would reduce the load factor to half the previous value.

For a set of 100 elements, with an array of length 32, the worst-case time complexity is $O(\frac{100}{32}) \approx O(3.2)$ but if you double the array length to 64 the new time complexity would become $O(\frac{100}{64}) \approx O(1.6)$.

If the average-case time complexity for a linked-list-based chained hash table is $O(\frac{\lambda}{2})$, then what would you want the length n of your array to be to achieve $O(1)$ time?

The average-case time complexity for a linked-list-based chained hash table is $O(\frac{\lambda}{2}) = O(\frac{k}{2n})$, then if you set the array length $n = \frac{k}{2}$ or more generally, make n on the order of $O(k)$ then the average time would be $O(1)$.

4.2.6 Resolving hash table collisions with "open addressing"

As an alternative to chaining, some hash tables use open addressing. Under this framework, data is stored directly to terms in the array, but the hash function would map to the initial index of the array. If there are collisions, one would access the neighboring element of the array.

Programmatically, it might look something like this:

1. if the term of the array indexed by `i = h(key)` is empty, store value there.
2. while the term of the array indexed by `i=h(key)` is not empty, update `i++` until you find an empty location.

More generally, the process of searching for an empty position is called probing, and is defined by the probing function `f(x)`

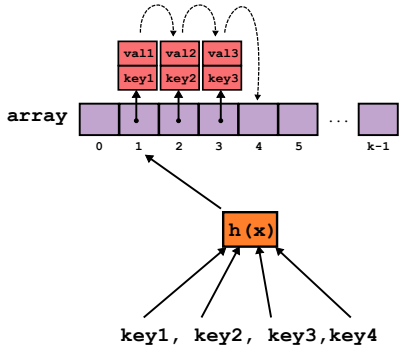
1. if the term of the array indexed by `i = h(key)` is empty, store value there.
2. while the term of the array indexed by `i=h(key)` is not empty, update `i = f(i)` until you find an empty location.

Different implementations of open addressing might use the following schemes. Here are some examples, but other schemes or variations on this could be used.

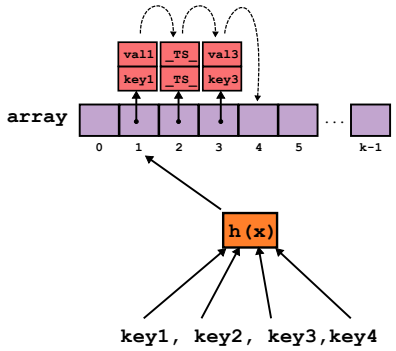
In each scheme, we start with a hash value $i = h(key)$ and we could create a series of probed indices, each parameterized by a counter j

- Linear Probing: $f(i, j) = i + c_1 j$ for $j = 1, 2, 3, ...$
- Quadratic Probing: $f(i, j) = i + c_2 j^2$ for $j = 1, 2, 3, ...$
- Double Hashing: $f(i, j, key) = i + j \times h_2(key)$ for $j = 1, 2, 3, ...$

We could compute a series of j values and test if one of these positions are empty until we find an empty bucket.



If we remove an item, this may complicate the open addressing system. When an item is deleted, a special element could be used to designate a deleted term until it is filled in again. This special term is called a "tombstone".



Reducing clustering with probing functions

Under this open address framework, the probability of adjacent items can complicate proper indexing. A series of adjacent elements can be called a cluster. With linear probing, as the array fills up, the probability of a collision increase, and the probability of forming a cluster also increases. Quadratic and double hashing methods reduce clustering by spreading out the terms.

4.2.7 Time-complexity of Open Addressing

The time complexity of working with a hash table using open addressing is probabilistic in nature, because the average-case time complexity involves probabilistic estimates of collisions. Let's consider a hash table using an array of size n and a uniform hash function. Then we can expect there is a probability $q = \frac{k}{n}$ of a collision. Similarly, there is a probability of $p = 1 - q = \frac{n-k}{n}$ of not having a collision.

4.2.8 Modeling Collisions as a Bernoulli Process

A **Bernoulli process** is a sequence $X_1, X_2, X_3, \ldots$ of binary random events that takes on two values 0 and 1.

In the example of hash table collisions, we have two outcomes: collision or no collision, let's consider the "no collision" case to be a success and have the value of 1, while "collision" will have the value of 0. In the Bernoulli process, we consider the two outcomes to be such that $P(X_i = 1) = p$ and $P(X_i = 0) = 1 - p$. We repeat the application of the probing function until we find an open slot, and this process can be modeled as a random process. With a sufficiently uniform probing function, this model becomes more accurate.

4.2.9 Modeling Collisions with a Geometric Distribution

To explain the time complexity of open addressing, we need to consider the geometric distribution. The geometric distribution describes the number of trials it would take to get a success (in our case, no collision). The probability of observing a success on the m-th trial would be:

$$P(X_1 = 0, X_1 = 0, \ldots, X_m = 1) = (1-p)^{m-1}p$$

To get the expected wait-time, we need to do an infinite series. We are summing over all values of m, the trial where the first success is observed, weighted by the probability $P(X_1 = 0, X_1 = 0, \ldots, X_m = 1)$. The result is the following:

$$E[T] = \sum_{m=1}^{\infty} m(1-p)^{m-1}p$$

Factoring out the single factor of p that down't depend on m out of the sum, gives us:

$$E[T] = p\sum_{m=1}^{\infty} m(1-p)^{m-1} = p\sum_{m=1}^{\infty} m(1-p)^{m-1}$$

Recognizing that this looks like a derivative, we can note that $\frac{d}{dp}(1-p)^m = m(1-p)^{m-1}(-1)$. Therefore, our expected wait-time can be rewritten as:

$$E[T] = p\sum_{m=1}^{\infty} m(1-p)^{m-1} = p\frac{d}{dp}\left[\sum_{m=1}^{\infty} -(1-p)^m\right]$$

Next, we note that we have a geometric series. The expression describing this sum is something that comes up again and again, and the most useful form here is (see Appendix 1.4.1):

$$\sum_{m=0}^{N} r^m = \frac{1+r^{N+1}}{1-r}$$

In the event that $|r| < 1$, we have $\sum_{m=0}^{\infty} r^m = \frac{1}{1-r}$. Therefore, in our example, we are almost there. We have this expression within our expression for $E[T]$:

$$\sum_{m=1}^{\infty}(1-p)^m = \frac{1}{1-(1-p)} - 1 = \frac{1}{p} - 1$$

Substituting this back into our previous expression, we have the following:

$$E[T] = p\frac{d}{dp}\left[1 - \frac{1}{p}\right] = p(\frac{1}{p^2}) = \frac{1}{p}$$

Recall that our expression for p was $p = 1 - \frac{k}{n} = 1 - \lambda$ the probability of not having a collision (success). Putting this together, we can compute the time complexity of building a hash table of n terms:

$$T(n) = \frac{1}{p} = \frac{1}{1-\frac{k}{n}} = \frac{1}{1-\lambda}$$

In summary, the average time complexity to add a new term to a hash table using open addressing and a uniform hash function would be $O(\frac{1}{1-\lambda})$. Compare this result to what we found for chaining, which was $O(\lambda)$. Arguably in both cases you would want the load factor to be λ. How do these two approaches compare as the load factor changes?

4.3 Implementing a Dictionary with Binary Search Trees

In many ways, the data structures have been presented as a series of comparisons between two schools of thought: arrays and pointers. The arrays adapted into dynamic arrays, and later as circular dynamic arrays. Meanwhile, the linked list added first and last pointers, and then evolved into a doubly linked list along with front and back sentinels.

The advantage of binary search is clear when you compare to brute-force search, which would take $O(n)$. Binary search would be able to find the element in $O(\lg n)$, and the improved search time comes from the splitting in two at each iteration of the recursive approach. At each step, the size of the remaining elements to check is divided by two, resulting in the logarithmic search time. However, this can only really be implemented with an array, because of the middle portion, where we access the middle element of the array, arr[mid]. This is only possible with an array, otherwise with a linked list we would need to traverse the remaining elements to get to the middle, adding extra steps. In other situations, one could also argue that the linked list has an advantage when inserting a new element, because the array requires the shifting of each element forward by one to make room, as we have seen many times before. This trade off between the search time and insertion time might place the array and linked list approach on par for some tasks, but there are other data structures that would solve both of these limitations.

The splitting-by-two operation for binary search is very similar to the branching of a tree. This insight turns out to be a breakthrough idea in designing data structures that can maintain a sorted list. With this in mind, let's consider the properties of trees.

Binary Trees

A **tree** can be both an abstract data type and a data structure, so the term is often used interchangeably, but in some contexts it can be valuable to make a distinction.

A tree is a useful concept in many fields, which can provide motivation for thinking about trees as an abstract data type. A tree is composed of nodes and edges, but also has a hierarchical structure. Trees have a root, and should have a single root. The edges of a tree are directed, and this direction defines the nested structure of the tree. Some examples of trees used in computing-related fields are described in Figure 4.1

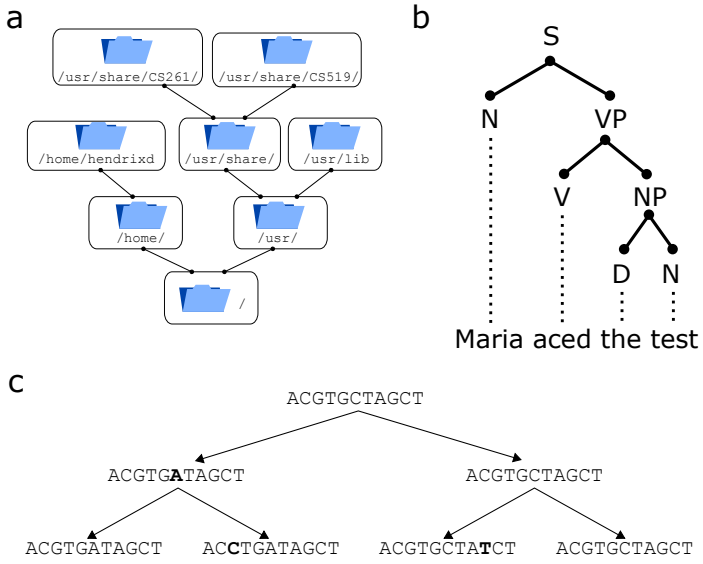


Figure 4.1: The following are examples of trees in computer science. **a.** directories in a GNU/Linux form a tree structure.**b.** A parse tree is a way of studying the grammar of a sentence. **c.** DNA mutations can be viewed as a phylogenetic tree.

All but one of these examples is a **binary tree**, which is defined such that each node has at most

two outward edges. These edges can be labeled `left` and `right` and are said to be child nodes, or more specifically the left child and right child. Similarly, we call the nodes directed toward a node (there should only be one) to be the parent node.

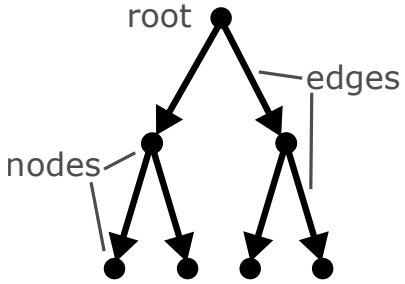


Figure 4.2: binary tree example 1

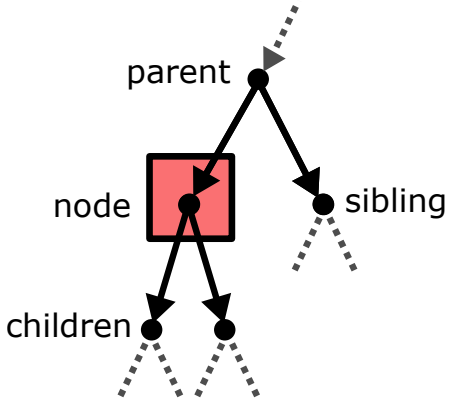
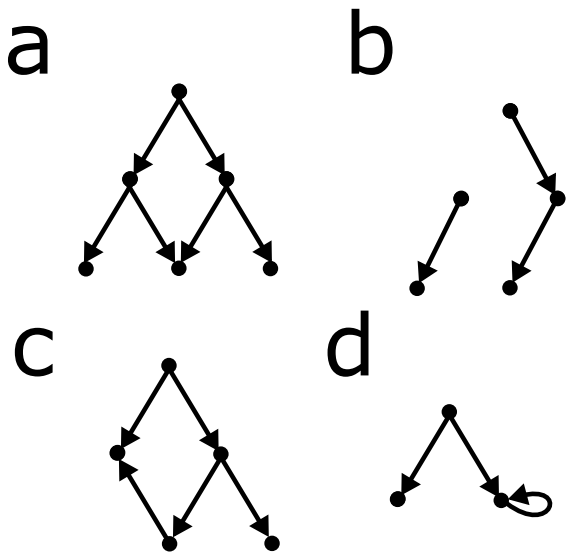


Figure 4.3: binary tree example 2

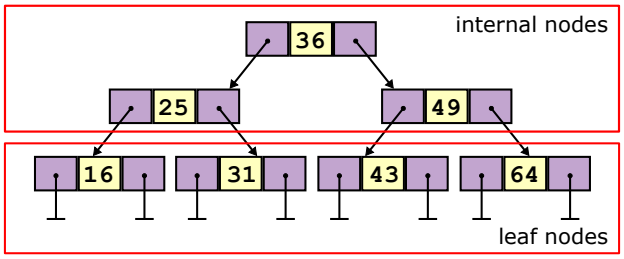
One way to learn the defining characteristics of trees is to see examples that are not trees. What properties of the following structures violate the properties of a tree?



Leaf nodes vs Internal Nodes

The leaf nodes are defined as nodes that no other nodes below them, so the left and right pointers should be NULL.

The internal nodes are the other nodes in the tree that are not leaf nodes. Note, that for a tree with only one node, the root is a leaf node. For a tree with more than one node, the root is considered an internal node.



4.3.1 Storing and Retrieving Keys in a Binary Search Trees

How would we create a data structure that we can add key/value pairs to nodes to such that the data associated with each node remains in order so that we can access the elements with binary search? We would like to:

- search
- insert
- delete

And do so fast. Let's say at most $O(\lg n)$ time for each task.

What about a dynamic array? This would take $O(\lg n)$ for search using binary search, but insertion would take $O(n)$ in worst case because the values would need to be shifted.

What about a linked list? Although the linked list solves the insertion problem and does so in $O(1)$, it does not solve the search. Search for a location in a linked list would take $O(n)$ in worst case, because you cannot implement binary search on a linked list.

The arrangement values of a binary search tree are motivated by binary search. The values less than a particular value at a particular node are left descendants of that node. Similarly, the values greater than a particular value at a particular node are right descendants of that node

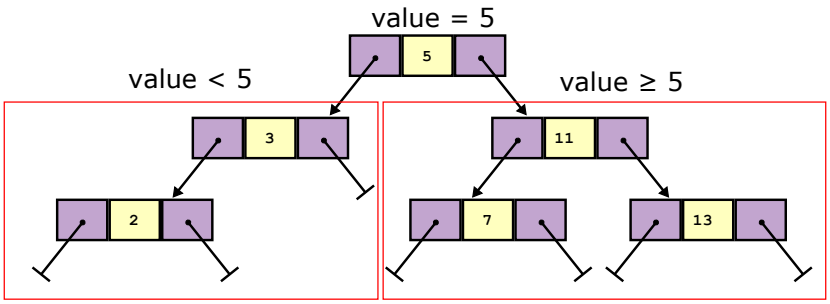


Figure 4.4: An example of a binary search tree. The left descendants of any node have a key less than, and all the right descendants have a key greater than the key of the root node.

Building of the motivation of the dictionary abstract data type, we can also think of the value presented in each of these nodes as a **key**. We can also think of each node containing extra data, called **value**. The keys and values can be `string`, `int`, `float` data type or more complex data with the requirement that the key can be compared for organization into the tree. Let's take a look at the implementation in python for the BST:

```
from typing import Optional, Any

class Node:
    def __init__(self, key: int, value: Any) -> None:
        self.key: int = key
        self.value: Any = value
        self.left: Optional[Node] = None
        self.right: Optional[Node] = None

def get(root: Optional[Node], query_key: int) -> Any:
    """
    Search for a key and return its associated value.
    Returns None if the key is not found.
    """
    n: Optional[Node] = root

    while n is not None:
        if query_key == n.key:
            return n.value
        elif query_key < n.key:
            n = n.left
        else:
            n = n.right

    return None
```

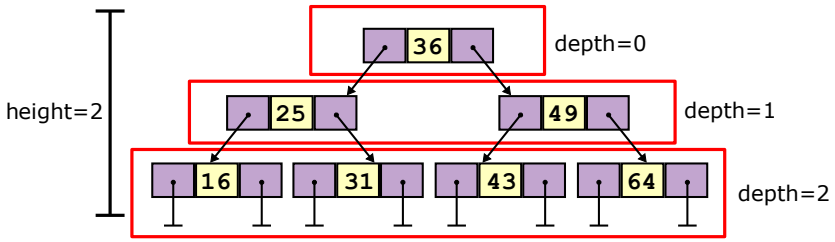
In this example, we can see the `get()` operation, which will take a key is input, and search through the tree. We can see how the recursion in the while loop will work its way down the tree. If we were to set a key/value pair, we would follow a similar procedure until we find a leaf node to which we can attach the created node:

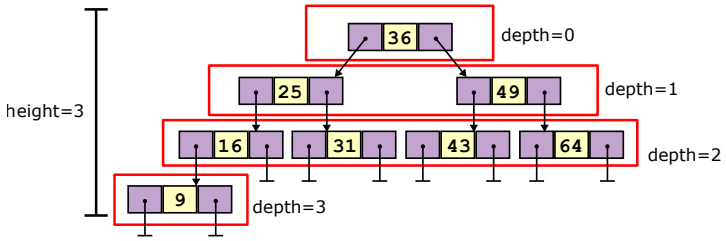
```
def set(root: Optional[Node], key: int, value: Any) -> Node:
    """
    Inserts a new key-value pair or updates an existing key's value.
    Returns the root of the tree.
    """
    if root is None:
        return Node(key, value)

    n: Optional[Node] = root
    p: Node = root

    while n is not None:
        p = n
        if key == n.key:
            n.value = value  # Update existing key, just like a dict
            return root
        elif key < n.key:
            n = n.left
        else:
            n = n.right

    # Standard BST insertion onto the parent 'p'
    if key < p.key:
        p.left = Node(key, value)
    else:
        p.right = Node(key, value)

    return root
```

4.4 Binary Search Tree: Space and Time

Tree Height and Depth

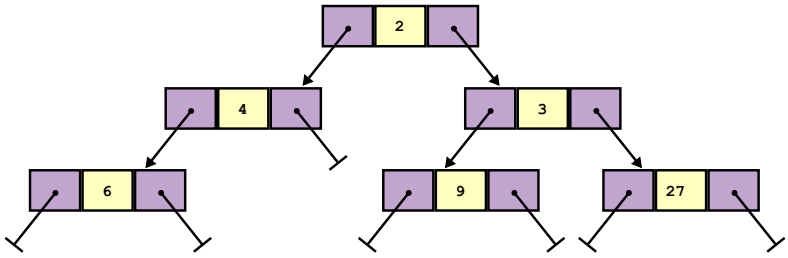
In order to talk about the time complexity of searching a BST, we need to consider the height and depth of the tree, and whether it is balanced. The **height** of the tree is the maximum number of edges needed to be traversed to get from the root to the furthest leaf node. The **depth** of a node is also defined as the number of ancestors of that node, or equivalently, the number of edges in the path to the root. The root node has a depth of 0. You can also define the depth recursively, and the depth of a node is one more than the depth of its parent node. The height of a binary tree is therefore the maximum depth of any node in the tree. Note that all the examples we have seen have been balanced trees, but this need not be the case.



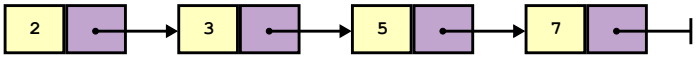


We can see that nodes on the same level with a common parent are called siblings. We can also call the nodes connected to by a directed path and above a given node the ancestors, and the nodes connected to by a directed path and below a given node as the descendants.

We could implement a tree using nodes, such that pointers defined as fields for these nodes would be branches in our tree. Such a construction might look like this:



With this example in mind, what about a singly linked list?



A linked list is a special and important case of a tree. It satisfies all the requirements of a tree, but it would be an example of a very unbalanced tree. This concept will be important later on when we consider worst-case time complexity of tree-related data structures.

4.4.1 Full and perfect binary trees

A **full** binary tree is such that every internal node has exactly two child nodes.

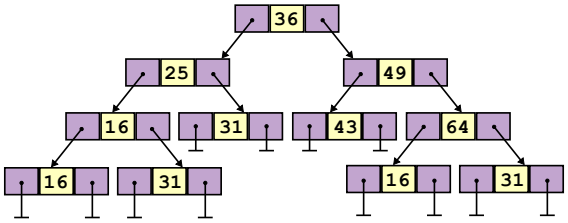


Figure 4.5: An example of a full binary tree. Every internal node has both a left and right child.

A **perfect** binary tree is a full tree such that every leaf node has the same depth.

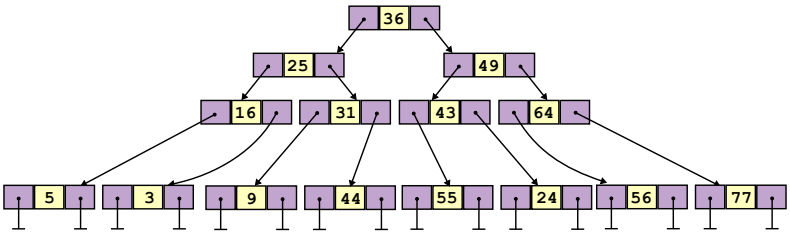


Figure 4.6: An example of a perfect binary tree. Every leaf node has the same depth.

A perfect binary tree of height h must have 2^h leaves. This can be seen by induction. With only two leaf nodes, there must be a root node, so the depth would be 1. The number of leaf nodes is $2^1 = 2$. For four leaf nodes, there must be three internal nodes, which includes a root and the two nodes that are parents of the leaf nodes. There would be a height of 2 and $2^2 = 4$.

A perfect binary tree of height h must have $2^{h+1} - 1$ total nodes. This can be understood by a sum. To count the total nodes, we have 1 for the root, 2 nodes at depth $d = 1$, and 4 nodes at depth $d = 2$, and so on. The total nodes n would be:

$$n = \sum_{d=0}^{h} 2^d = \frac{1 - 2^{h+1}}{1 - 2} = 2^{h+1} - 1$$

Therefore, $n = 2^{h+1} - 1$. Similarly, if we know that a perfect tree has n total nodes, then we can see that

$$h = \lg(n + 1) - 1$$

To a reasonable approximation, we can use $h = \lg(n)$. This rule is not generally tree for any binary tree, and in worst-case we would have h to be $O(n)$.

How many leaf nodes in a perfect binary tree?

How many leaf nodes in a perfect binary tree of n nodes? Recall that the root node is depth 0 and the height h of a tree is the maximum depth of any node. A perfect binary tree with n nodes must have a height h is such that $h = \lg(n + 1) - 1$. And we also saw that because of the doubling effect, the number of leaf nodes L should be such that $L = 2^h$. Therefore:

$$L = 2^h = 2^{\lg(n+1)-1} = 2^{\lg(n+1)}2^{-1} = \frac{n+1}{2}$$

4.4.2 The Time Complexity of Searching a BST

Search in a binary search tree would take $O(h)$ for a tree of height h. For a balanced tree, this would be such that $h \propto \lg(n)$, so the result is $O(\lg n)$. In Python, we would implement something like this approach for search, but clearly the structure is similar to our `get()` function above.

```
from typing import Optional, Any

def search(bst: Optional[Node], query: Any) -> Optional[Node]:
    n: Optional[Node] = bst

    while n is not None:
        if n.key == query:
            return n
        elif query < n.key:
            n = n.left
        else:
```

```
            n = n.right

    return None  # Explicitly return None if not found
```

4.4.3 The Time Complexity of `insert` with a BST

In the this example, the variable `p` is used to track the location of the new node's parent, but otherwise the search proceeds as it does in the search() method.

```
def insert(bst: Optional[Node], k: int, v: int) -> Node:
    new_node = Node(k, v)

    # Handle the empty tree case
    if bst is None:
        return new_node

    n: Optional[Node] = bst
    p: Node = bst # Parent tracker

    # 1. Traverse to find the insertion point
    while n is not None:
        p = n
        if k < n.key:
            n = n.left
        else:
            n = n.right

    # 2. Attach the new node to the parent
    if k < p.key:
        p.left = new_node
    else:
        p.right = new_node

    return bst
```

Like binary trees, a BST that is perfect and has n nodes has a height of $h \propto O(\lg(n))$. This rule is not generally tree for any binary search tree. The structure for a binary tree in general depends on the order in which elements were added. In worst-case we would have $h \propto O(n)$.

On average, for uniformly random input data we might see balance as a long-term statistical trend, but individual trees could be quite unbalanced. We will see later on that we can impose further restrictions on how we build and modify a tree to ensure that it is always balanced.

4.4.4 The Time Complexity of Deleting from a BST

To efficiently manage space of a BST we should consider how to delete a node.

There are three possible scenarios. Either the node to be deleted has no child nodes, one child node, or two child nodes.

If there are no child nodes, we can just free the memory.

If there is only one child node, that node will take the place of the deleted node, and will be called its **successor**.

In the case when there are two children, we would need to define what node would replace a deleted node. If a node has more than one child node, we have to evaluate the distribution of numbers below it.

The **in-order successor** of a given node in a BST is defined as the node with the key next in ascending order.

The in-order successor of a given node in a BST can be found as the leftmost node in its right subtree.

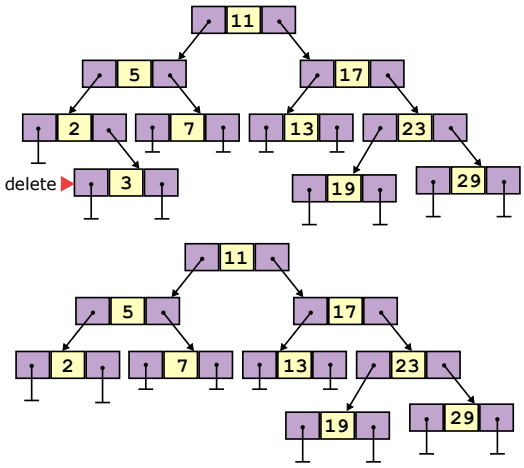


Figure 4.7: Deleting from a Binary Search Tree when there are no child nodes

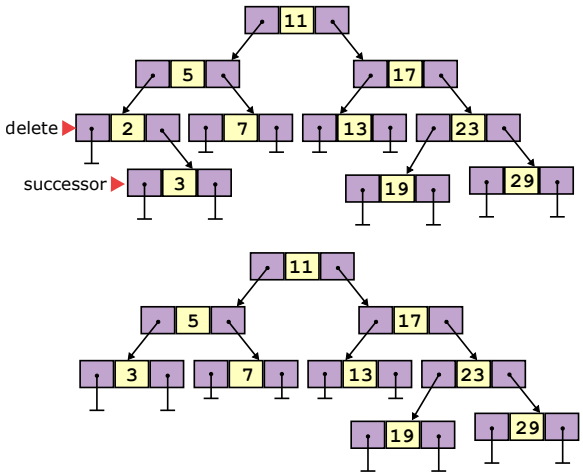


Figure 4.8: Deleting from a Binary Search Tree when there is one child node (successor)

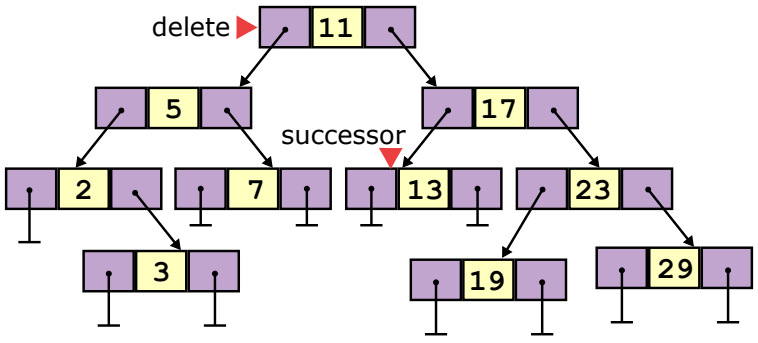


Figure 4.9: In general, the successor node is the left most node in the right subtree of the deleted node.

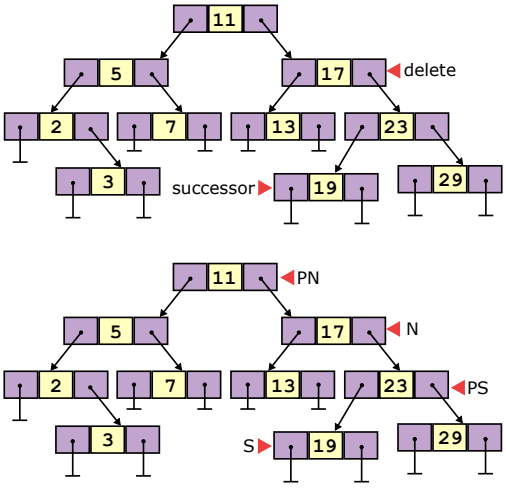
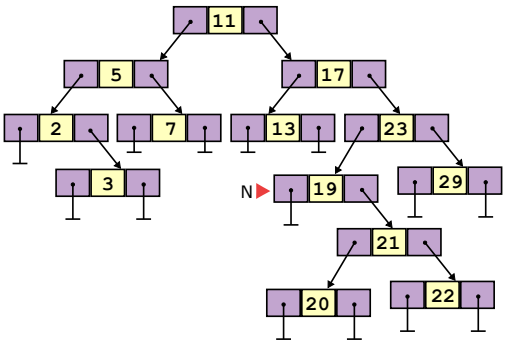


Figure 4.10: Deleting from a Binary Search Tree in general requires defining the node, the parent node, the successor, and the parent of the successor.



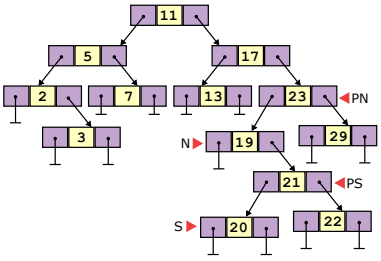
Let's define the following

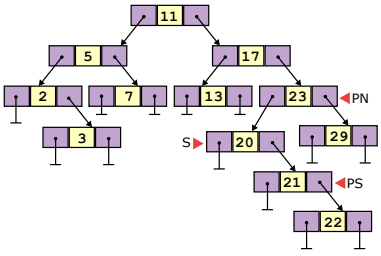
- `N` The node to be deleted
- `PN` The parent of the node to be deleted
- `S` The successor of the node to be deleted
- `PS` The parent of the successor

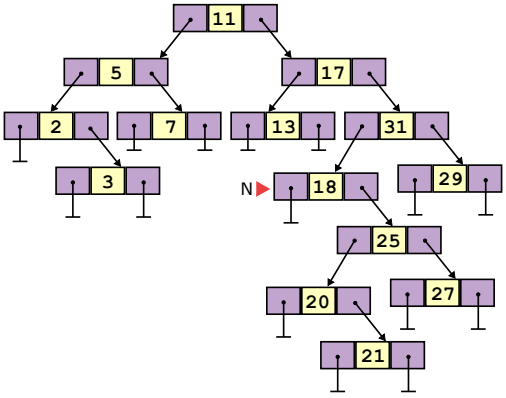
In general, our deletion procedure should proceed as follows:

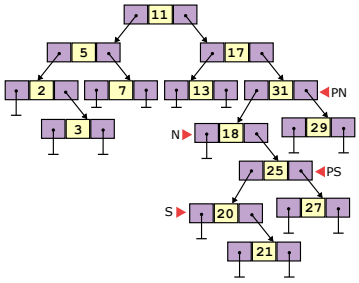
1. `S.left = n.left`
2. `PS.left = S.right`
3. `S.right = N.right`
4. `PN.left = S` or `PN.right = S`, depending on where `N` was.
5. `free(N)`

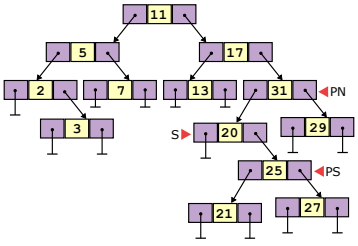
That previous example used the other successor rule, but since the node only has one child, that could be the successor. This result might be more straightforward.

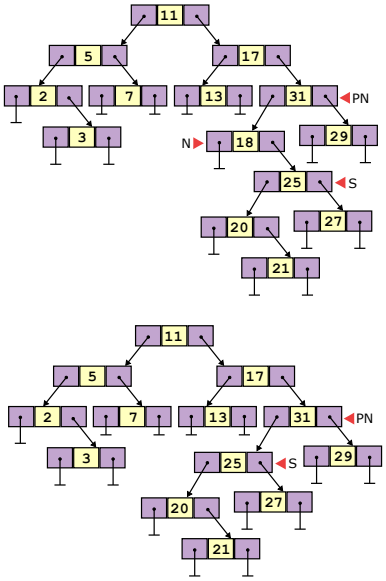

11
5
17
2
7
13
23
PN
3
N
19
29
21
PS
S
20
22


11
5
17
2
7
13
23
PN
3
S
20
29
21
PS
22


11
5
17
2
7
13
31
3
N
18
29
25
20
27
21


11
5
17
2
7
13
31
PN
3
N
18
29
25
PS
S
20
27
21


11
5
17
2
7
13
31
PN
3
S
20
29
25
PS
21
27




```
def get_min_key(node):
    current = node
    while current.left is not None:
        current = current.left
    return current.key

def remove(key, n):
    if n is None:
        return None

    if key < n.key:
        n.left = remove(key, n.left)
    elif key > n.key:
        n.right = remove(key, n.right)
    else:
        # Found it!
        # Case: Two children
        if n.left is not None and n.right is not None:
            n.key = get_min_key(n.right)
            n.right = remove(n.key, n.right)
        # Case: Left child only
        elif n.left is not None:
            return n.left
        # Case: Right child only
        elif n.right is not None:
            return n.right
        # Case: No children (leaf)
        else:
            return None

    return n
```

Summary of the Time Complexity of BSTs

For each of the operations we have considered, we should have roughly the same time complexity:

- insert - $O(h)$
- delete - $O(h)$
- search - $O(h)$

However, the height h can vary quite a bit as a function of the length n. Therefore, in worst case, these operations could be considered $O(n)$

Algorithm	Average-case	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\lg(n))$	$O(n)$
Insert	$O(\lg(n))$	$O(n)$
Delete	$O(\lg(n))$	$O(n)$

4.5 Tree Traversal

To put in perspective both space and time complexity in the context of binary trees, let's consider the operation of visiting every node in a tree.

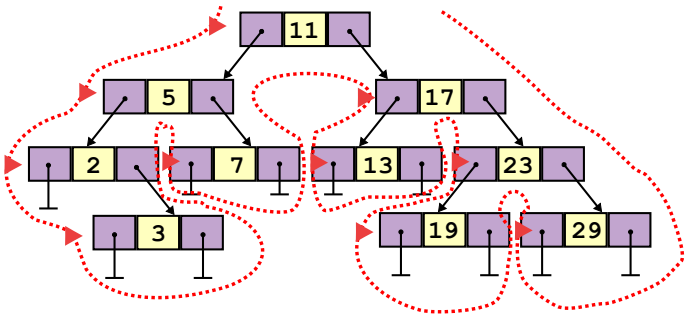
These types of traversals must be $O(n)$ but it is best to define them systematically so that think about the sequence of events involved in traversing them. This will build intuition about the time complexity of tree operations later on.

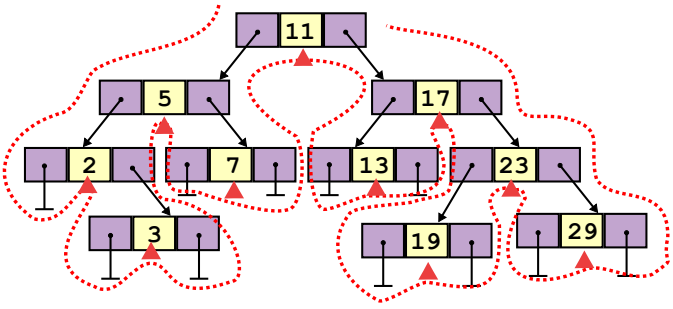
- Depth-First Search - Traverse the tree from root to leaf node, visiting all of a nodes descendents for each path on the tree
- Breadth-First Search - Traverse the tree such that you broadly search each level, visiting all nodes of the same depth together

4.5.1 Depth-first traversal of a binary tree

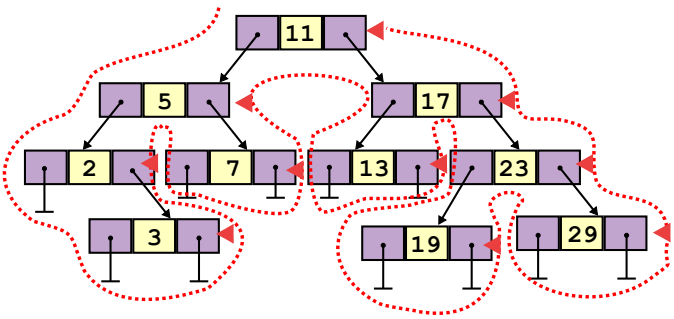
- Pre-Order Traversal - traverse a tree starting at the root, and proceed in order increasing depth. Parent nodes processed before any of its child nodes
- Post-Order Traversal - traverse the tree starting at the leaf nodes, and proceed in order of decreasing depth. Child nodes processed before their parent node.

We could implement the pre-order traversal recursively, such that we process the current node before traversing the left and right subtrees. Similarly, we could recursively define post-order traversal by traversing the left and right subtrees before processing the current node.





In-order Traversal: 2,3,5,7,11,13,17,19,23,29



Post-order Traversal: 3,2,7,5,13,19,29,23,17,11

If you label the three operations as N, L, and R corresponding to processing the current node, traversing the left subtree, and traversing the right subtree respectively, we can simplify the explanation of pre-order traversal with NLR, and label post-order traversal as LRN.

- In-Order Traversal - traverse a sorted binary search tree such that the keys are visited in ascending order
- Reverse-Order Traversal - traverse a sorted binary search tree such that the keys are visited in descending order

We can implement in-order traversal using recursion such that we first traverse the left subtree, then process current node, then traverse the right subtree. Similarly, we can implement reverse-order traversal using recursion that traverses the right subtree, then processes the current node, then traverses the left subtree.

Using the N, L, and R system described above, we can understand in-order traversal as LNR, and reverse-order traversal as RNL. We can summarize all four methods of depth-first traversal with the following:

- Pre-Order Traversal - NLR
- Post-Order Traversal - LRN
- In-Order Traversal - LNR
- Reverse-Order Traversal - RNL

The other two possibilities not listed are NRL and RLN, which correspond to performing pre- and post-order traversal in the opposite direction (clockwise) because it would visit right nodes before left nodes.

4.6 Implementing a Dictionary with AVL Trees

As we saw with BSTs, the critical issue is the fact that the basic operations were found to be $O(h)$ and this can be $O(n)$ in the worst-case scenario. We can achieve $O(lg(n))$, which was or initial goal, if we maintain balance on our binary search trees. There is no built-in mechanism to BSTs with our treatment that would ensure balance in the resulting tree depending on how the nodes are added. Based on this issue with the BST, what is the worst possible order that you could add values to a BST?

4.6.1 Balance and binary search trees

We will need a quantitative definition of balance. Intuitively, we might consider something where the number of left descendants for each node would be equal to the number of right descendants. We can also describe this comparison as comparing the sizes of the left and right subtrees. Calculating these values for a given tree would be somewhat costly, but we could update these values as we go.

Recall that `height` was introduced as related to maximum depth of a tree. We could also define it for a subtree, and expect that the left and right subtrees to have the same height.

Height balance is defined as a situation where the height of the left subtree differs from the height of the right subtree by at most one.

It has been shown that maintaining height balance for a BST will result in a tree with $h \propto O(\lg n)$

For our self-balancing trees, we can add the height to the node, and keep track of it as we go.

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 0  # Initialize height field
```

4.6.2 Balance Factor

Balance factor more generally describes the degree of balance of a node by comparing the height of its subtrees.

$$balance\_factor(node) = height(right(node)) - height(left(node))$$

in Python we would have something like this. Let's first assume that we do not have the `height` field that we added above:

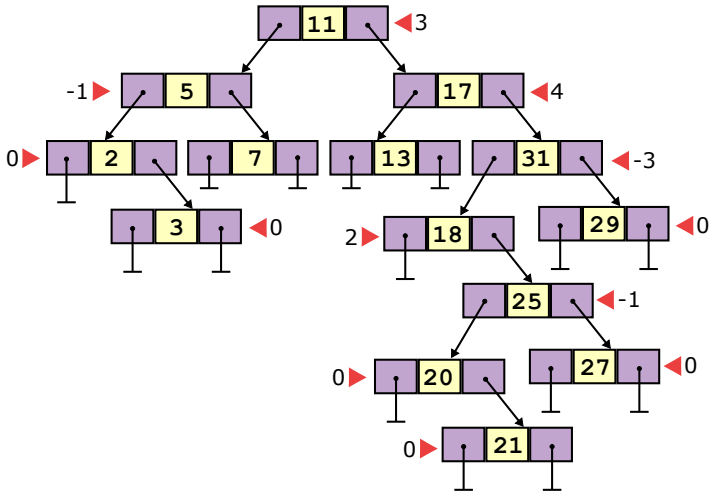
```
def height(n):
    # Base case: an empty node has a height of -1
    if n is None:
        return -1

    # Recursively find the height of left and right subtrees
    left_height = height(n.left)
    right_height = height(n.right)

    # Current node height is 1 + the height of its tallest child
    return 1 + max(left_height, right_height)

def balance_factor(n):
    if n is None:
        return 0
    return height(n.right) - height(n.left)
```

The above code includes a function to calculate the height if needed, but we have also added a `height` field for each node to keep track of it, which leads to $O(1)$ time to get the height.

```
def get_height(n):
    """Helper to handle None nodes safely."""
    if n is None:
        return -1
    return n.height

def update_height(n):
    # Height = 1 + max height of children
    n.height = 1 + max(get_height(n.left), get_height(n.right))

def balance_factor(n):
    """Calculates balance factor in O(1) time using the stored height field."""
    if n is None:
        return 0
    # Right height - Left height
    return get_height(n.right) - get_height(n.left)
```

We also have a `update_height` function there to update the height of a node if needed.

The height balance would be achieved with a balance factor between -1 and +1. Because of the direction of the difference, a negative value would correspond to more nodes on the left, and a positive value would correspond to more nodes on the right. Large positive values are called **right heavy** and large negative values are called **left heavy**



4.6.3 Maintaining height balance with AVL Trees

To address the limitations of binary search trees Adelson-Velsky and Landis created self-balancing trees that are named after their initials: AVL Trees. The AVL tree maintains a balance factor between -1 and 1 for each node, and hence maintains height balance for each node.

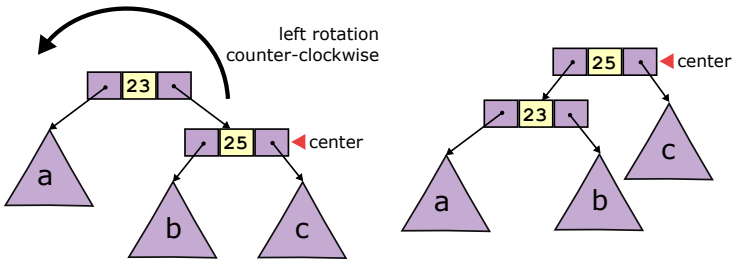
4.6.4 Rotations with AVL Trees

AVL trees maintain balance through an operation called "rotation".

A **rotation** is a procedure whereby the balance score is checked after each node insertion or removal. Each rotation can be described as a rudimentary exchange of pointers of the nodes such that one node moves up the tree, and another moves down the tree to bring the balance score closer to 0.

To motivate the name, each rotation operation has a center node, and a direction.

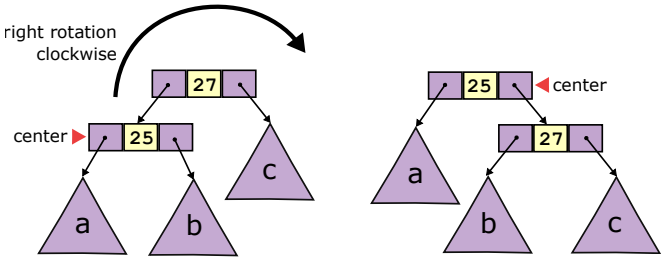
The left rotation results in a counter-clockwise exchange, where the parent node of a center node becomes the left child of the center.



If we implement the left rotation in Python, we have the following. Note that we update the heights of the root and center as these are the only ones that are changing. The height of each node in the subtrees doesn't change because it is calculated relative to their leaf nodes.

```
def left_rotate(root):
    center = root.right
    subtree_b = center.left

    # Perform left rotation
    center.left = root
    root.right = subtree_b

    # Update heights: only root and center changed
    # Assuming a helper function max() and height() exist
    root.height = max(height(root.left), height(root.right)) + 1
    center.height = max(height(center.left), height(center.right)) + 1

    # center is the new root
    return center
```

The right rotation results in a clockwise exchange, where the parent node of a center node becomes the right child of the center.



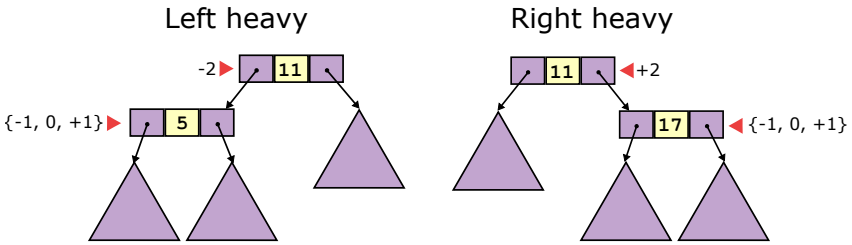
The code for the right rotation proceeds similarly to the left rotation. The subtree to consider in this case is now the right subtree below the center node.

```
def right_rotate(root):
    center = root.left
    subtree_b = center.right

    # Perform right rotation
    center.right = root
```

```
    root.left = subtree_b

    # Update heights: only root and center changed
    root.height = max(height(root.left), height(root.right)) + 1
    center.height = max(height(center.left), height(center.right)) + 1

    # center is the new root
    return center
```

4.6.5 Situations when the AVL Tree is rebalanced

The AVL tree would be rebalanced in situations when a node is left heavy or right heavy.

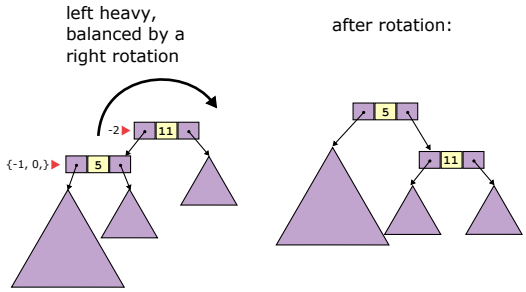


But the decision on whether to perform a single or a double rotation is based on the left or right child of the unbalanced node, depending on whether the unbalanced node is left or right heavy.

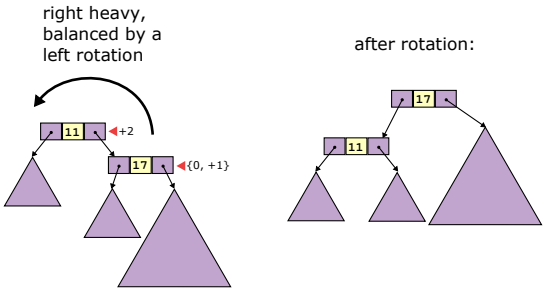
Single Rotations with AVL Trees

Single rotations will balance a node when the heavy side has 0 or the same sign as the unbalanced node above.

A left heavy node (balanceFactor=-2) with a left child that is either -1 or 0 is solved by a single right rotation.



A right heavy node (balanceFactor=+2) with a right child that is either +1 or 0 is solved by a single left rotation.



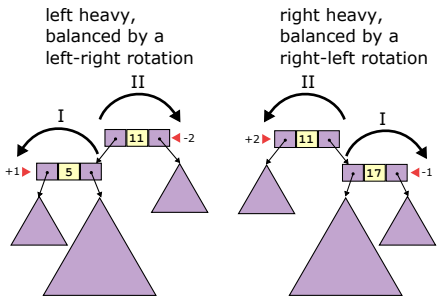
Double Rotation with AVL Trees

It is also possible that two rotations are needed. This would be a double rotation, and usual would be sufficient to create a balanced tree.

Double rotations will balance a node when the heavy side has the opposite sign as the unbalanced node above.

A left heavy node -2 with a left child that is +1 is solved by a double rotation: a left-right rotation.

A right heavy node +2 with a right child that is -1 is solved by a double rotation: a right-left rotation.



4.7 AVL Trees: Space and Time

4.7.1 AVL Trees: Space Complexity

The space complexity of an AVL trees would stay $O(n)$. We might store details about the height or balance score for each node, but this would just be an additive factor.

4.7.2 AVL Trees: Time Complexity

Our goal for the BST was the achieve $O(\lg(n))$ for insert, delete, and search. We found that we did get this on average, assuming uniformly distributed data. Instead we got $O(h)$ for a height h.

The time complexity of an AVL trees is $O(\lg(n))$ for these basic operations because height balance is maintained.

It has been shown that maintaining height balance for a BST (i.e. AVL Trees) will result in a tree with $h \propto O(\lg(n))$, and more precisely,

$$\lg(n+1) - 1 \leq h \leq \frac{\lg(n+2)}{\lg(\varphi)} + \frac{\lg(5)}{2\lg(\varphi)} - 3$$

where $\varphi = \frac{1+\sqrt{5}}{2}$, the golden ratio. This result comes from Knuth, D. E. *The Art of Computing: Sorting and searching* , 2000. Putting it all together, we have the following details on AVL Trees:

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\lg(n))$	$O(\lg(n))$
Insert	$O(\lg(n))$	$O(\lg(n))$
Delete	$O(\lg(n))$	$O(\lg(n))$

Chapter 5

Priority Queues

A **priority queue** is an abstract data type such that each element has a priority. We can consider a priority queue P such that each element x has a quantity that defines its priority. Let's consider this priority the key $key(x)$ for each element x, and it defines a ranking within the ADT.

We expect to define the operation $max(P)$ to return the element with the largest priority.

Our priority queue should have the following defined operations:

- $insert(P, x)$ - insert the element x into the priority queue P.
- $update(P, x, k)$ - update the priority of element x such that $key(x) = k$
- $max\_dequeue(P)$ - remove the element with the maximum value and return it
- $max(P)$ - return the element with the maximum value, but don't alter P.
- $enqueue(P, x)$ - add the element x at the back of the priority queue, the position with lowest priority.

How could we do this in an efficient way? If we would do this with a BST, we could incur a cost of $O(n)$ in worst-case to retrieve the maximum value. That said, we could work around this by having a pointer to the lower right node, but this would need to be updated. Moreover, it isn't clear how a BST or AVL tree would be the most efficient data structure when we need to frequently dequeue the maximum element.

5.1 Implementing Priority Queues with Heaps

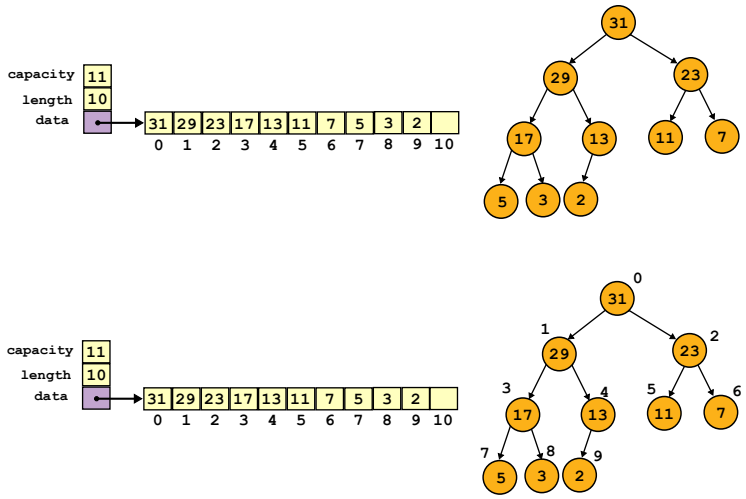
A **heap** is a data structure that can be used to implement a priority queue. In many ways the heap bridges the divide between arrays and trees, and we can consider it an array imagined as a tree.

Let's define a heap as a binary tree that satisfies the **heap property**. For a **max heap**, the heap property requires that for any node n, its parent node $parent(n)$ is such that $key(parent(n)) \geq key(n)$. Similarly, for a **min heap**, the heap property requires that for any node n, its parent node p is such that $key(parent(n)) \leq key(n)$.

Without loss of generality, from here on out we will only consider max heaps, and will use the terms heap and max heap interchangeably. There can be non-binary heaps, but let's consider the binary case here, which is the most common.

The heap property is attained only when the largest term is always stored as the root of the heap. Let's consider the following example where we compare the array representation with the tree representation.

Now, let's label the nodes of the heap with the corresponding position of the heap.



As we can see, when we compare the array and the tree representations, we can see that the root is $H[0]$ the first term of the array.

We can see that the left and right nodes of the node n are such that $left(i) = 2i+1$ and $right(i) = 2i+2$.

Similarly, we can also see that the parent of the node corresponding to position i of the array can be computed as the node at position $\lfloor \frac{i-1}{2} \rfloor$, where we are using a type of floor function $\lfloor x \rfloor$ to round down to an integer. In other words,

$$parent(i) = \lfloor \frac{i-1}{2} \rfloor$$

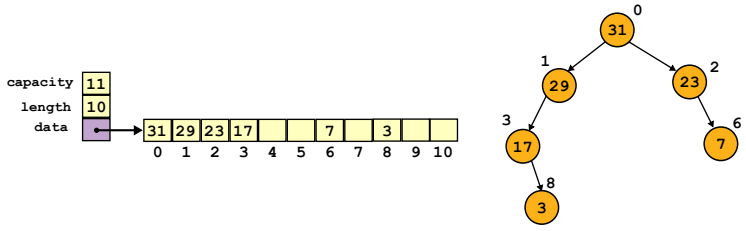
5.1.1 Heaps correspond to complete trees

Heaps are defined as "complete" because it is represented as a compact array. A **complete** binary tree of height h is such that:

1. At each level, or depth $d < h$, there are 2^d nodes.
2. The nodes at depth $d = h$ are as far left as possible.

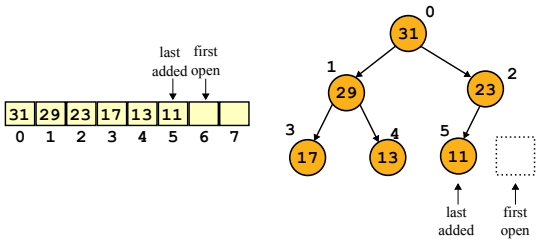
Many of the time complexity rules we've seen before for a perfect tree of height h hold for a complete binary tree of height h.

If the tree was not near-complete, we might have this type of scenario:



In this example, using an array becomes more costly, because time is spent searching for elements.

However, if the tree is kept near complete, it is clear where available positions are with lowest priority, and where the last element was added, assuming we are adding to our queue with an type of enqueue operation.



5.1.2 insert(P,x)

Let's consider the insert operation for the priority queue P and the element x. We want to ensure that the key $key(x)$ is such that it is not larger than it's parent. The sequence of steps is as follows:

1. Put the element at the end of the array, i.
2. Identify the parent node, which should be at $\lfloor \frac{i-1}{2} \rfloor$
3. Compare $key(H[i])$ with the parent's key $key(H[parent(i)])$.
4. If $key(H[i]) > key(H[parent(i)])$, then swap the elements i and $\lfloor \frac{i-1}{2} \rfloor$ of the array
5. repeat from step 2

This process is perhaps best illustrated by an example. An example of inserting a new element to a heap is illustrated in Figure 5.1.

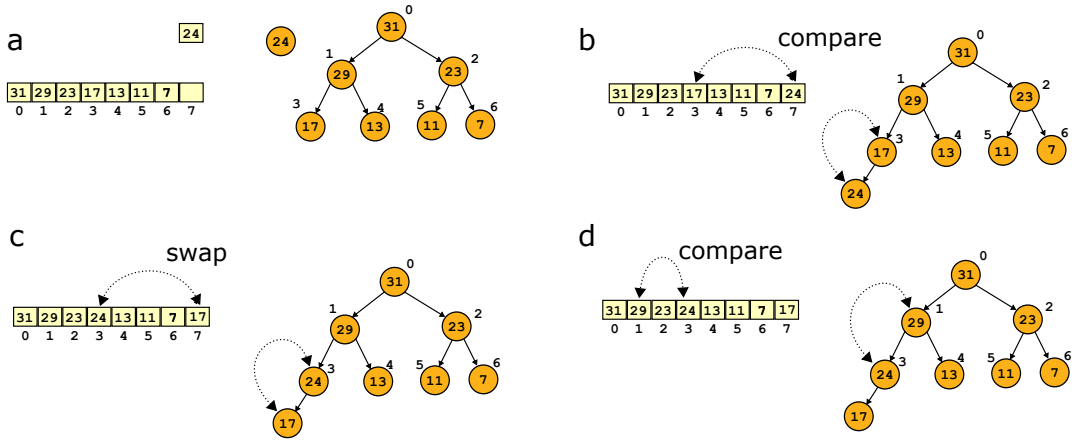


Figure 5.1: An example of adding an element to a heap. **a.** Initially, the new node is created and the heap is in in a particular state. The heap should have a first available position, in this example, the available position is indexed by 7. **b.** The new node is added to the first available slot, and compared to its parent's key. **c.** In this example, since $24 > 17$, the node is swapped with its parent to preserve the heap property. **d.** Lastly, the node is compared to its parent node, but in this example $24 < 29$, so we are done.

5.2 Heaps: Space and Time

5.2.1 Time Complexity of Insert to a Heap

Because we add to the first available position, and we compare via a path from this leaf node up to the root, we are performing an operation for each level of the tree. Because the tree is near-complete, this should be $O(\lg(n))$ comparisons.

5.2.2 $max(P)$ and $max\_dequeue(P)$

Deleting a node from a heap properly would require that we maintain the heap property.

Recall that for a max heap, the maximum-priority node should be the root node. Therefore, to retrieve the max node, or to dequeue the maximum priority node would involve the root node. Removing the root node would require some sort of percolation down the tree to maintain the heap property. At the same time, removing the root node splits the tree into two disconnected subtrees, making it difficult to merge and maintain the heap property, so what should we do?

Perhaps counter-intuitively, we can solve this replacing the root node with the least likely candidate for root, node at the last position of the heap. This can sometimes be the lowest priority node, but doesn't have to be. This prevents our tree from being disconnected, and allows us to percolate down the value until it reaches the correct level of the tree. Let's see an example.

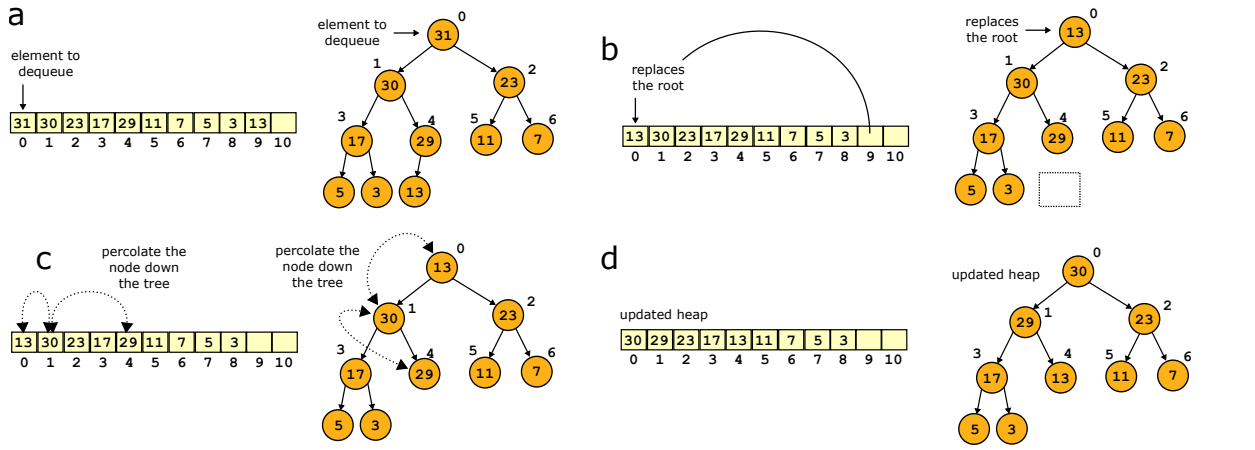


Figure 5.2: An example of deleting an element from a heap. **a.** First the heap is in its initial state, and the element to be removed, with the maximum priority, will be the root. **b.** Next, the root is replaced with the last added element. **c.** After the root is replaced, the element is percolated down the heap, through a series of comparisons between the added node and its children. At each step, the element that does not satisfy the heap property is swapped with its largest priority child. **d.** Lastly, after percolation of the elements, the heap is in the updated state.

The operation of $max\_dequeue(P)$ when implemented in a heap must therefore take $O(\lg(n))$ comparisons, which would go from root to leaf node at worst-case, which for a near-complete tree should be logarithmic.

5.2.3 The Time Complexity of Fixing a Node

If you have an array H that has an element at position i that causes a violation of the heap property, how would you fix it to create a heap? Let's assume that the left and right subtrees of this node satisfy the heap property, but the node itself does not. Note: a binary tree with a single node always satisfies the heap property, and similarly, leaf nodes correspond to subtrees that satisfy the heap property.

Let's define an operation, called `fix_node(H,i)` that for a heap H and position i.

The operation `fix_node(H,i)` should do the following:

1. Compare $H[i]$ with it's left child $H[2i+1]$ and right child $H[2i+2]$, swap it with the max of those two.
2. recursively call `fix_node()` on the swapped position if the swap resulted in a violation.

Because we are performing this operation from an arbitrary position down to the leaf nodes, this operation should also result in logarithmic time $O(\lg(n))$.

5.2.4 Building a max-heap from an unordered array.

Let's say you have a random array H of length n. How would adjust the elements so that it results in a max heap?

First, note that the leaf nodes are already max heaps. Which positions correspond to the leaf nodes?

We saw before that a perfect binary tree with n nodes there are $\frac{n+1}{2}$ leaf nodes. Therefore, for a perfect tree, there must be $n - \frac{n+1}{2} = \frac{n-1}{2}$ internal nodes.

But we are dealing with a near-complete tree. The most number of leaf nodes it can have is $\lceil \frac{n}{2} \rceil$, so our last internal node would be $\frac{n}{2} - 1$

Since all leaf nodes correspond to subtrees that max-heaps, we only need to fix the internal nodes, which will be indexed from 0 to $\frac{n}{2} - 1$

The operation `build_heap(H)` should do the following:

```
import math

def build_heap(H):
    n = len(H)
    # Start from the last non-leaf node and move up to the root (index 0)
    # i goes from ceil(n/2) down to 0
    for i in range(math.ceil(n / 2), -1, -1):
        fix_node(H, i)
```

In python, there is an optimized module called `heapq` that implements this and turn any list into a heap in place:

```
import heapq
heapq.heapify(H)
```

5.2.5 The Time Complexity of `build_heap()`

You might think it is $O(n \lg(n))$, because we have to work on $O(\frac{n}{2}))$ nodes, with $O(\lg(n))$ operations for each node. However, let's double check this.

Let's consider that each node x has a height $h(x)$, defined as the height of the subtree that the node is a root of, and a depth $d(x)$ defined by the distance to the root.

Note that for a near-complete tree of total height h, we have the rule $h \geq d(x) + h(x)$ for each node x.

Let's consider a sum over k, which will represent the height of the node $h(x)$. We know there should be 2^d nodes at depth d, which corresponds to 2^{h-k} nodes with height k.

$$T(h) = \sum_{k=0}^{h} 2^{h-k} k = 2^h \sum_{k=0}^{h} \frac{1}{2}^k k$$

We can solve that sum exactly by considering the following. As we have seen,

$$\sum_{k=0}^{n} r^k = \frac{1 - r^{n+1}}{1 - r}$$

In the case where $|r| < 1$, this series converges for $n \to \infty$ because $r^{n+1} \to 0$

$$\sum_{k=0}^{\infty} r^k = \frac{1}{1 - r}$$

but we can use the fact that $kr^k = rkr^{k-1} = r\frac{d}{dr}r^k$ to show that:

$$r\frac{d}{dr}\sum_{k=0}^{\infty} r^k = r\frac{d}{dr}\frac{1}{1 - r}$$

and thus:

$$\sum_{k=0}^{\infty} kr^k = \frac{r}{(1-r)^2}$$

We can now use this expression to define an upper-bound to our time-complexity.

$$T(h) \leq 2^h \sum_{k=0}^{\infty} \frac{1}{2}^k k = 2^h \left(\frac{1/2}{(1-1/2)^2} \right) = 2^{h+1}$$

So the time complexity of building a heap of height h is such that $T(h) \leq 2^{h+1}$ and therefore is $O(n)$

5.2.6 Sorting a list with heap-sort

We can use this method to sort a list, a method called `heap sort`.

1. Run `max_dequeue(P)`, but instead of removing the root, just swap it with the last node `H(k)`. Do not percolate down beyond the node indexed by k.
2. Repeat until $k = 0$

In this procedure, we can keep track of the current position k we are on (current last node). As this position decreases, the values of the array greater than k will be sorted.