7.7 Link-Based vs. Array-Based Sequences

We close this chapter by reflecting on the relative pros and cons of array-based and link-based data structures that have been introduced thus far. The dichotomy between these approaches presents a common design decision when choosing an appropriate implementation of a data structure. There is not a one-size-fits-all solution, as each offers distinct advantages and disadvantages.

Advantages of Array-Based Sequences

- *Arrays provide $O(1)$-time access to an element based on an integer index.* The ability to access the k^{th} element for any k in $O(1)$ time is a hallmark advantage of arrays (see Section 5.2). In contrast, locating the k^{th} element in a linked list requires $O(k)$ time to traverse the list from the beginning, or possibly $O(n-k)$ time, if traversing backward from the end of a doubly linked list.

- *Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array-based structure versus a linked structure.* As an example, consider the typical enqueue operation for a queue. Ignoring the issue of resizing an array, this operation for the ArrayQueue class (see Code Fragment 6.7) involves an arithmetic calculation of the new index, an increment of an integer, and storing a reference to the element in the array. In contrast, the process for a LinkedQueue (see Code Fragment 7.8) requires the instantiation of a node, appropriate linking of nodes, and an increment of an integer. While this operation completes in $O(1)$ time in either model, the actual number of CPU operations will be more in the linked version, especially given the instantiation of the new node.

- *Array-based representations typically use proportionally less memory than linked structures.* This advantage may seem counterintuitive, especially given that the length of a dynamic array may be longer than the number of elements that it stores. Both array-based lists and linked lists are referential structures, so the primary memory for storing the actual objects that are elements is the same for either structure. What differs is the auxiliary amounts of memory that are used by the two structures. For an array-based container of n elements, a typical worst case may be that a recently resized dynamic array has allocated memory for $2n$ object references. With linked lists, memory must be devoted not only to store a reference to each contained object, but also explicit references that link the nodes. So a singly linked list of length n already requires $2n$ references (an element reference and next reference for each node). With a doubly linked list, there are $3n$ references.

9.2 Implementing a Priority Queue

In this section, we show how to implement a priority queue by storing its entries in a positional list *L*. (See Section 7.4.) We provide two realizations, depending on whether or not we keep the entries in *L* sorted by key.

9.2.1 The Composition Design Pattern

One challenge in implementing a priority queue is that we must keep track of both an element and its key, even as items are relocated within our data structure. This is reminiscent of a case study from Section 7.6 in which we maintain access counts with each element. In that setting, we introduced the ***composition design pattern***, defining an _Item class that assured that each element remained paired with its associated count in our primary data structure.

For priority queues, we will use composition to store items internally as pairs consisting of a key *k* and a value *v*. To implement this concept for all priority queue implementations, we provide a PriorityQueueBase class (see Code Fragment 9.1) that includes a definition for a nested class named _Item. We define the syntax a < b, for item instances a and b, to be based upon the keys.

```
class PriorityQueueBase:
  """Abstract base class for a priority queue."""

  class _Item:
    """Lightweight composite to store priority queue items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __lt__(self, other):
      return self._key < other._key      # compare items based on their keys

  def is_empty(self):                    # concrete method assuming abstract len
    """Return True if the priority queue is empty."""
    return len(self) == 0
```

Code Fragment 9.1: A PriorityQueueBase class with a nested _Item class that composes a key and a value into a single object. For convenience, we provide a concrete implementation of is_empty that is based on a presumed __len__ impelementation.

11.1 Binary Search Trees

In Chapter 8 we introduced the tree data structure and demonstrated a variety of applications. One important use is as a ***search tree*** (as described on page 332). In this chapter, we use a search tree structure to efficiently implement a ***sorted map***. The three most fundamental methods of a map M (see Section 10.1.1) are:

M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError; implemented with __getitem__ method.

M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k; implemented with __setitem__ method.

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError; implemented with __delitem__ method.

The sorted map ADT includes additional functionality (see Section 10.3), guaranteeing that an iteration reports keys in sorted order, and supporting additional searches such as find_gt(k) and find_range(start, stop).

Binary trees are an excellent data structure for storing items of a map, assuming we have an order relation defined on the keys. In this context, a ***binary search tree*** is a binary tree T with each position p storing a key-value pair (k, v) such that:

- Keys stored in the left subtree of p are less than k.
- Keys stored in the right subtree of p are greater than k.

An example of such a binary search tree is given in Figure 11.1. As a matter of convenience, we will not diagram the values associated with keys in this chapter, since those values do not affect the placement of items within a search tree.

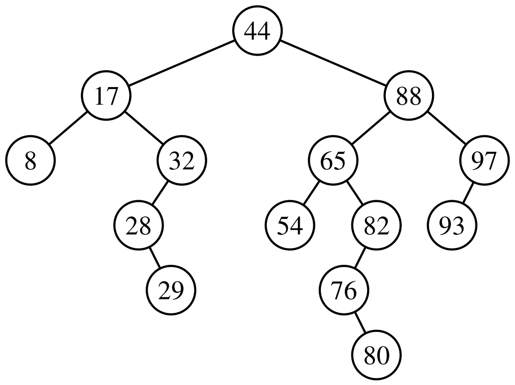

Figure 11.1: A binary search tree with integer keys. We omit the display of associated values in this chapter, since they are not relevant to the order of items within a search tree.

11.3 AVL Trees

The TreeMap class, which uses a standard binary search tree as its data structure, should be an efficient map data structure, but its worst-case performance for the various operations is linear time, because it is possible that a series of operations results in a tree with linear height. In this section, we describe a simple balancing strategy that guarantees worst-case logarithmic running time for all the fundamental map operations.

Definition of an AVL Tree

The simple correction is to add a rule to the binary search tree definition that will maintain a logarithmic height for the tree. Although we originally defined the height of a subtree rooted at position p of a tree to be the number of *edges* on the longest path from p to a leaf (see Section 8.1.3), it is easier for explanation in this section to consider the height to be the number of *nodes* on such a longest path. By this definition, a leaf position has height 1, while we trivially define the height of a "null" child to be 0.

In this section, we consider the following ***height-balance property***, which characterizes the structure of a binary search tree T in terms of the heights of its nodes.

***Height-Balance Property*:** For every position p of T, the heights of the children of p differ by at most 1.

Any binary search tree T that satisfies the height-balance property is said to be an ***AVL tree***, named after the initials of its inventors: Adel'son-Vel'skii and Landis. An example of an AVL tree is shown in Figure 11.11.

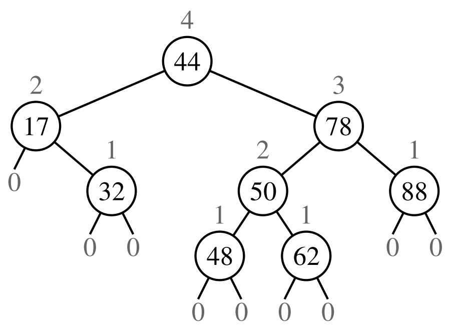

Figure 11.11: An example of an AVL tree. The keys of the items are shown inside the nodes, and the heights of the nodes are shown above the nodes (with empty subtrees having height 0).

11.4.4 Amortized Analysis of Splaying ⋆

After a zig-zig or zig-zag, the depth of position p decreases by two, and after a zig the depth of p decreases by one. Thus, if p has depth d, splaying p consists of a sequence of $\lfloor d/2 \rfloor$ zig-zigs and/or zig-zags, plus one final zig if d is odd. Since a single zig-zig, zig-zag, or zig affects a constant number of nodes, it can be done in $O(1)$ time. Thus, splaying a position p in a binary search tree T takes time $O(d)$, where d is the depth of p in T. In other words, the time for performing a splaying step for a position p is asymptotically the same as the time needed just to reach that position in a top-down search from the root of T.

Worst-Case Time

In the worst case, the overall running time of a search, insertion, or deletion in a splay tree of height h is $O(h)$, since the position we splay might be the deepest position in the tree. Moreover, it is possible for h to be as large as n, as shown in Figure 11.21. Thus, from a worst-case point of view, a splay tree is not an attractive data structure.

In spite of its poor worst-case performance, a splay tree performs well in an amortized sense. That is, in a sequence of intermixed searches, insertions, and deletions, each operation takes on average logarithmic time. We perform the amortized analysis of splay trees using the accounting method.

Amortized Performance of Splay Trees

For our analysis, we note that the time for performing a search, insertion, or deletion is proportional to the time for the associated splaying. So let us consider only splaying time.

Let T be a splay tree with n keys, and let w be a node of T. We define the ***size*** $n(w)$ of w as the number of nodes in the subtree rooted at w. Note that this definition implies that the size of a nonleaf node is one more than the sum of the sizes of its children. We define the ***rank*** $r(w)$ of a node w as the logarithm in base 2 of the size of w, that is, $r(w) = \log(n(w))$. Clearly, the root of T has the maximum size, n, and the maximum rank, $\log n$, while each leaf has size 1 and rank 0.

We use cyber-dollars to pay for the work we perform in splaying a position p in T, and we assume that one cyber-dollar pays for a zig, while two cyber-dollars pay for a zig-zig or a zig-zag. Hence, the cost of splaying a position at depth d is d cyber-dollars. We keep a virtual account storing cyber-dollars at each position of T. Note that this account exists only for the purpose of our amortized analysis, and does not need to be included in a data structure implementing the splay tree T.

11.5.2 (2,4)-Tree Operations

A multiway search tree that keeps the secondary data structures stored at each node small and also keeps the primary multiway tree balanced is the $(\mathbf{2},\mathbf{4})$ ***tree***, which is sometimes called a 2-4 tree or 2-3-4 tree. This data structure achieves these goals by maintaining two simple properties (see Figure 11.24):

***Size Property*:** Every internal node has at most four children.

***Depth Property*:** All the external nodes have the same depth.

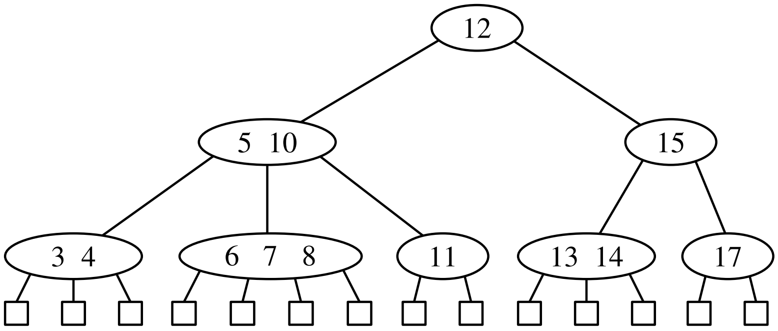

Figure 11.24: A $(2,4)$ tree.

Again, we assume that external nodes are empty and, for the sake of simplicity, we describe our search and update methods assuming that external nodes are real nodes, although this latter requirement is not strictly needed.

Enforcing the size property for $(2,4)$ trees keeps the nodes in the multiway search tree simple. It also gives rise to the alternative name "2-3-4 tree," since it implies that each internal node in the tree has 2, 3, or 4 children. Another implication of this rule is that we can represent the secondary map stored at each internal node using an unordered list or an ordered array, and still achieve $O(1)$-time performance for all operations (since $d_{\max} = 4$). The depth property, on the other hand, enforces an important bound on the height of a $(2,4)$ tree.

Proposition 11.8: *The height of a $(2,4)$ tree storing n items is $O(\log n)$.*

Justification: Let h be the height of a $(2,4)$ tree T storing n items. We justify the proposition by showing the claim

$$\frac{1}{2}\log(n+1) \le h \le \log(n+1). \tag{11.9}$$

To justify this claim note first that, by the size property, we can have at most 4 nodes at depth 1, at most 4^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at most 4^h. Likewise, by the depth property and the definition

Performance of the Edge List Structure

The performance of an edge list structure in fulfilling the graph ADT is summarized in Table 14.2. We begin by discussing the space usage, which is $O(n+m)$ for representing a graph with n vertices and m edges. Each individual vertex or edge instance uses $O(1)$ space, and the additional lists V and E use space proportional to their number of entries.

In terms of running time, the edge list structure does as well as one could hope in terms of reporting the number of vertices or edges, or in producing an iteration of those vertices or edges. By querying the respective list V or E, the vertex_count and edge_count methods run in $O(1)$ time, and by iterating through the appropriate list, the methods vertices and edges run respectively in $O(n)$ and $O(m)$ time.

The most significant limitations of an edge list structure, especially when compared to the other graph representations, are the $O(m)$ running times of methods get_edge(u,v), degree(v), and incident_edges(v). The problem is that with all edges of the graph in an unordered list E, the only way to answer those queries is through an exhaustive inspection of all edges. The other data structures introduced in this section will implement these methods more efficiently.

Finally, we consider the methods that update the graph. It is easy to add a new vertex or a new edge to the graph in $O(1)$ time. For example, a new edge can be added to the graph by creating an Edge instance storing the given element as data, adding that instance to the positional list E, and recording its resulting Position within E as an attribute of the edge. That stored position can later be used to locate and remove this edge from E in $O(1)$ time, and thus implement the method remove_edge(e)

It is worth discussing why the remove_vertex(v) method has a running time of $O(m)$. As stated in the graph ADT, when a vertex v is removed from the graph, all edges incident to v must also be removed (otherwise, we would have a contradiction of edges that refer to vertices that are not part of the graph). To locate the incident edges to the vertex, we must examine all edges of E.

Operation	**Running Time**
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u,v,x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

Table 14.2: Running times of the methods of a graph implemented with the edge list structure. The space used is $O(n+m)$, where n is the number of vertices and m is the number of edges.

Proposition 14.18 suggests a simple algorithm for computing the transitive closure of $\vec{G}$ that is based on the series of rounds to compute each $\vec{G}_k$. This algorithm is known as the ***Floyd-Warshall algorithm***, and its pseudo-code is given in Code Fragment 14.9. We illustrate an example run of the Floyd-Warshall algorithm in Figure 14.11.

```
Algorithm FloydWarshall($\vec{G}$):
   Input: A directed graph $\vec{G}$ with $n$ vertices
   Output: The transitive closure $\vec{G}^*$ of $\vec{G}$
    let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
    $\vec{G}_0 = \vec{G}$
    for $k = 1$ to $n$ do
       $\vec{G}_k = \vec{G}_{k-1}$
       for all $i, j$ in $\{1, \ldots, n\}$ with $i \neq j$ and $i, j \neq k$ do
          if both edges $(v_i, v_k)$ and $(v_k, v_j)$ are in $\vec{G}_{k-1}$ then
             add edge $(v_i, v_j)$ to $\vec{G}_k$ (if it is not already present)
    return $\vec{G}_n$
```

Code Fragment 14.9: Pseudo-code for the Floyd-Warshall algorithm. This algorithm computes the transitive closure $\vec{G}^*$ of G by incrementally computing a series of directed graphs $\vec{G}_0, \vec{G}_1, \ldots, \vec{G}_n$, for $k = 1, \ldots, n$.

From this pseudo-code, we can easily analyze the running time of the Floyd-Warshall algorithm assuming that the data structure representing G supports methods get_edge and insert_edge in $O(1)$ time. The main loop is executed n times and the inner loop considers each of $O(n^2)$ pairs of vertices, performing a constant-time computation for each one. Thus, the total running time of the Floyd-Warshall algorithm is $O(n^3)$. From the description and analysis above we may immediately derive the following proposition.

Proposition 14.19: *Let $\vec{G}$ be a directed graph with n vertices, and let $\vec{G}$ be represented by a data structure that supports lookup and update of adjacency information in $O(1)$ time. Then the Floyd-Warshall algorithm computes the transitive closure $\vec{G}^*$ of $\vec{G}$ in $O(n^3)$ time.*

Performance of the Floyd-Warshall Algorithm

Asymptotically, the $O(n^3)$ running time of the Floyd-Warshall algorithm is no better than that achieved by repeatedly running DFS, once from each vertex, to compute the reachability. However, the Floyd-Warshall algorithm matches the asymptotic bounds of the repeated DFS when a graph is dense, or when a graph is sparse but represented as an adjacency matrix. (See Exercise R-14.12.)

Let us first assume that we are representing the graph G using an adjacency list or adjacency map structure. This data structure allows us to step through the vertices adjacent to u during the relaxation step in time proportional to their number. Therefore, the time spent in the management of the nested **for** loop, and the number of iterations of that loop, is

$$\sum_{u \text{ in } V_G} \text{outdeg}(u),$$

which is $O(m)$ by Proposition 14.9. The outer **while** loop executes $O(n)$ times, since a new vertex is added to the cloud during each iteration. This still does not settle all the details for the algorithm analysis, however, for we must say more about how to implement the other principal data structure in the algorithm—the priority queue Q.

Referring back to Code Fragment 14.12 in search of priority queue operations, we find that n vertices are originally inserted into the priority queue; since these are the only insertions, the maximum size of the queue is n. In each of n iterations of the **while** loop, a call to remove_min is made to extract the vertex u with smallest D label from Q. Then, for each neighbor v of u, we perform an edge relaxation, and may potentially update the key of v in the queue. Thus, we actually need an implementation of an ***adaptable priority queue*** (Section 9.5), in which case the key of a vertex v is changed using the method update(ℓ,k), where ℓ is the locator for the priority queue entry associated with vertex v. In the worst case, there could be one such update for each edge of the graph. Overall, the running time of Dijkstra's algorithm is bounded by the sum of the following:

- n insertions into Q.
- n calls to the remove_min method on Q.
- m calls to the update method on Q.

If Q is an adaptable priority queue implemented as a heap, then each of the above operations run in $O(\log n)$, and so the overall running time for Dijkstra's algorithm is $O((n+m)\log n)$. Note that if we wish to express the running time as a function of n only, then it is $O(n^2 \log n)$ in the worst case.

Let us now consider an alternative implementation for the adaptable priority queue Q using an unsorted sequence. (See Exercise P-9.58.) This, of course, requires that we spend $O(n)$ time to extract the minimum element, but it affords very fast key updates, provided Q supports location-aware entries (Section 9.5.1). Specifically, we can implement each key update done in a relaxation step in $O(1)$ time—we simply change the key value once we locate the entry in Q to update. Hence, this implementation results in a running time that is $O(n^2+m)$, which can be simplified to $O(n^2)$ since G is simple.

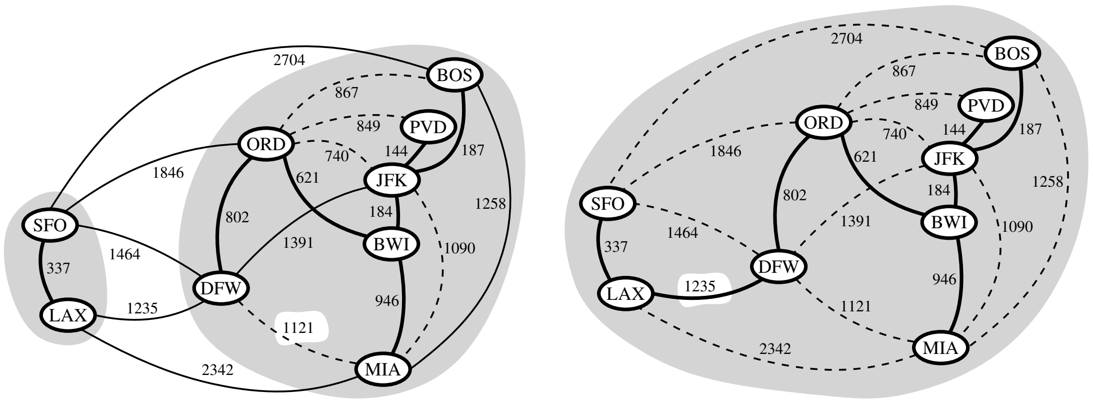

Figure 14.24: Example of an execution of Kruskal's MST algorithm (continued). The edge considered in (n) merges the last two clusters, which concludes this execution of Kruskal's algorithm. (Continued from Figure 14.23.)

The Running Time of Kruskal's Algorithm

There are two primary contributions to the running time of Kruskal's algorithm. The first is the need to consider the edges in nondecreasing order of their weights, and the second is the management of the cluster partition. Analyzing its running time requires that we give more details on its implementation.

The ordering of edges by weight can be implemented in $O(m \log m)$, either by use of a sorting algorithm or a priority queue Q. If that queue is implemented with a heap, we can initialize Q in $O(m \log m)$ time by repeated insertions, or in $O(m)$ time using bottom-up heap construction (see Section 9.3.6), and the subsequent calls to remove_min each run in $O(\log m)$ time, since the queue has size $O(m)$. We note that since m is $O(n^2)$ for a simple graph, $O(\log m)$ is the same as $O(\log n)$. Therefore, the running time due to the ordering of edges is $O(m \log n)$.

The remaining task is the management of clusters. To implement Kruskal's algorithm, we must be able to find the clusters for vertices u and v that are endpoints of an edge e, to test whether those two clusters are distinct, and if so, to merge those two clusters into one. None of the data structures we have studied thus far are well suited for this task. However, we conclude this chapter by formalizing the problem of managing ***disjoint partitions***, and introducing efficient ***union-find data structures***. In the context of Kruskal's algorithm, we perform at most $2m$ find operations and $n-1$ union operations. We will see that a simple union-find structure can perform that combination of operations in $O(m + n \log n)$ time (see Proposition 14.26), and a more advanced structure can support an even faster time.

For a connected graph, $m \geq n-1$, and therefore, the bound of $O(m \log n)$ time for ordering the edges dominates the time for managing the clusters. We conclude that the running time of Kruskal's algorithm is $O(m \log n)$.