9.2.3 Implementation with a Sorted List

An alternative implementation of a priority queue uses a positional list, yet main-
taining entries sorted by nondecreasing keys. This ensures that the ·rst element ofthe list is an entry with the sm
OurSortedPriorityQueue class is given in Code Fragment 9.3. The implemen-
tation of minandremove
minare rather straightforward given knowledge that the
·rst element of a list has a minimum key. We rely on the ·rst method of the posi-
tional list to ·nd the position of the ·rst item, and the delete method to remove the
entry from the list. Assuming that the list is implemented with a doubly linked list,operations minandremove
mintake O(1)time.

This bene·t comes at a cost, however, for method addnow requires that we scan
the list to ·nd the appropriate position to insert the new item. Our implementation
starts at the end of the list, walking backward until the new key is smaller than
an existing item; in the worst case, it progresses until reaching the front of thelist. Therefore, the addmethod take
of entries in the priority queue at the time the method is executed. In summary,
when using a sorted list to implement a priority queue, insertion runs in linear time,
whereas ·nding and removing the minimum can be done in constant time.

Comparing the Two List-Based Implementations

Table 9.2 compares the running times of the methods of a priority queue realizedby means of a sorted and unso
off when we use a list to implement the priority queue ADT. An unsorted list
supports fast insertions but slow queries and deletions, whereas a sorted list allowsfast queries and deletions, bu

Operation	Unsorted List	Sorted List
len	O(1)	O(1)
is empty	O(1)	O(1)
add	O(1)	O(n)
min	O(n)	O(1)
remove min	O(n)	O(1)

Table 9.2: Worst-case running times of the methods of a priority queue of size n,
realized by means of an unsorted or sorted list, respectively. We assume that the
list is implemented by a doubly linked list. The space requirement is O(n).

R-10.24 Give a pseudo-code description of the
delitem
map operation when
using a skip list.
R-10.25 Give a concrete implementation of the pop method, in the context of a
MutableSet abstract base class, that relies only on the ·ve core set behav-
iors described in Section 10.5.2.
R-10.26 Give a concrete implementation of the isdisjoint method in the context
of the MutableSet abstract base class, relying only on the ·ve primary
abstract methods of that class. Your algorithm should run in O(min (n,m))
where nand mdenote the respective cardinalities of the two sets.
R-10.27 What abstraction would you use to manage a database of friends· birth-
days in order to support ef·cient queries such as ··nd all friends whosebirthday is today· and ··nd the friend who
Creativity
C-10.28 On page 406 of Section 10.1.3, we give an implementation of the methodsetdefault as it might appear i
While that method accomplishes the goal in a general fashion, its ef·-ciency is less than ideal. In particular, wher
getitem
, and then a subse-
quent insertion via
setitem
. For a concrete implementation, such as
theUnsortedTableMap , this is twice the work because a complete scan
of the table will take place during the failed
getitem
, and then an-
other complete scan of the table takes place due to the implementation of
setitem
. A better solution is for the UnsortedTableMap class to over-
ridesetdefault to provide a direct solution that performs a single search.
Give such an implementation of UnsortedTableMap.setdefault .
C-10.29 Repeat Exercise C-10.28 for the ProbeHashMap class.
C-10.30 Repeat Exercise C-10.28 for the ChainHashMap class.
C-10.31 For an ideal compression function, the capacity of the bucket array for ahash table should be a prime nu
·nding such a prime by using the sieve algorithm . In this algorithm, we
allocate a 2 Mcell Boolean array A, such that cell iis associated with the
integer i. We then initialize the array cells to all be ·true· and we ·mark
off· all the cells that are multiples of 2, 3, 5, 7, and so on. This processcan stop after it reaches a number larger t
2M. (Hint: Consider a
bootstrapping method for ·nding the primes up to·
2M.)

10.3.2 Two Applications of Sorted Maps
In this section, we explore applications in which there is particular advantage to using a sorted map rather than a traditional (unsorted) map. To apply a sorted map, keys must come from a domain that is totally ordered. Furthermore, to take advantage of the inexact or range searches afforded by a sorted map, there should be some reason why nearby keys have relevance to a search.
Flight Databases
There are several Web sites on the Internet that allow users to perform queries on·ight databases to ·nd ·ights b
buy a ticket. To make a query, a user speci·es origin and destination cities, a depar-
ture date, and a departure time. To support such queries, we can model the ·ightdatabase as a map, where keys
to these four parameters. That is, a key is a tuple
k=(origin,destination ,date,time).
Additional information about a ·ight, such as the ·ight number, the number of seatsstill available in ·rst (F) and co
be stored in the value object.
Finding a requested ·ight is not simply a matter of ·nding an exact match for a requested query. Although a user typically wants to exactly match the ori-
gin and destination cities, he or she may have ·exibility for the departure date,and certainly will have some ·exibi
We can handle such a query by ordering our keys lexicographically. Then, an ef-
·cient implementation for a sorted map would be a good way to satisfy users·queries. For instance, given a user
ge(k) to return
the ·rst ·ight between the desired cities, having a departure date and time match-ing the desired query or later. B
use·nd
range(k1, k2) to ·nd all ·ights within a given range of times. For exam-
ple, if k1=(ORD, PVD, 05May, 09:30),a n d k2=(ORD, PVD, 05May, 20:00),
a respective call to ·nd
range(k1, k2) might result in the following sequence of
key-value pairs:
(ORD, PVD, 05May, 09:53):(AA 1840, F5, Y15, 02:05,
251),
(ORD, PVD, 05May, 13:29):(AA 600, F2, Y0, 02:16,
713),
(ORD, PVD, 05May, 17:39):(AA 416, F3, Y9, 02:09,
365),
(ORD, PVD, 05May, 19:50):(AA 1828, F9, Y25, 02:13,
186)

12.7 Selection

As important as it is, sorting is not the only interesting problem dealing with a total order relation on a set of elements. There are a number of applications in whichwe are interested in identifying a
maximum elements, but we may also be interested in, say, identifying the median element, that is, the element such that half of the other elements are smaller and the remaining half are larger. In general, queries that ask for an element with a givenrank are called order statistics .

De·ning the Selection Problem

In this section, we discuss the general order-statistic problem of selecting the kth smallest element from an unsorted collection of ncomparable elements. This is known as the selection problem. Of course, we can solve this problem by sorting the collection and then indexing into the sorted sequence at index k·1. Using the best comparison-based sorting algorithms, this approach would take O(nlogn) time, which is obviously an overkill for the cases where k=1o r k=n(or even k=2,k=3,k=n·1, or k=n·5), because we can easily solve the selection problem for these values of kinO(n)time. Thus, a natural question to ask is whether we can achieve an O(n)running time for all values of k(including the interesting case of ·nding the median, where k=·n/2·).

12.7.1 Prune-and-Search

We can indeed solve the selection problem in O(n)time for any value of k.M o r e - over, the technique we use to achieve this result involves an interesting algorithmic design pattern. This design pattern is known as prune-and-search ordecrease- and-conquer . In applying this design pattern, we solve a given problem that is de·ned on a collection of nobjects by pruning away a fraction of the nobjects and recursively solving the smaller problem. When we have ·nally reduced theproblem to one de·ned on a const
the problem using some brute-force method. Returning back from all the recursivecalls completes the constructio
in Section 4.1.3 is an example of the prune-and-search design pattern.

13.5 Tries
The pattern-matching algorithms presented in Section 13.2 speed up the search in
a text by preprocessing the pattern (to compute the failure function in the Knuth-Morris-Pratt algorithm or the last
a series of queries is performed on a ·xed text, so that the initial cost of preprocess-
ing the text is compensated by a speedup in each subsequent query (for example, aWeb site that offers pattern m
that offers Web pages on the Hamlet topic).
Atrie(pronounced ·try·) is a tree-based data structure for storing strings in
order to support fast pattern matching. The main application for tries is in infor-mation retrieval. Indeed, the name
information retrieval application, such as a search for a certain DNA sequence in agenomic database, we are giv
alphabet. The primary query operations that tries support are pattern matching andpre·x matching . The latter op
for all the strings in Sthat contain Xas a pre·x.
13.5.1 Standard Tries
Let Sbe a set of sstrings from alphabet ·such that no string in Sis a pre·x
of another string. A standard trie for Sis an ordered tree Twith the following
properties (see Figure 13.10):
·Each node of T, except the root, is labeled with a character of ·.
·The children of an internal node of Thave distinct labels.
·Thas sleaves, each associated with a string of S, such that the concatenation
of the labels of the nodes on the path from the root to a leaf vofTyields the
string of Sassociated with v.
Thus, a trie Trepresents the strings of Swith paths from the root to the leaves
ofT. Note the importance of assuming that no string in Sis a pre·x of another
string. This ensures that each string of Sis uniquely associated with a leaf of T.
(This is similar to the restriction for pre·x codes with Huffman coding, as describedin Section 13.4.) We can alwa
acter that is not in the original alphabet ·at the end of each string.
An internal node in a standard trie Tcan have anywhere between 1 and |·|
children. There is an edge going from the root rto one of its children for each
character that is ·rst in some string in the collection S. In addition, a path from the
root of Tto an internal node vat depth kcorresponds to a k-character pre·x X[0:k]

e
zeze
mizei
nimize ze nimizemi nimize
(a)
0:2 6:8
6:8 2:8 2:82:8 1:2
6:87:8
4:8
e01234567
minimiz
(b)
Figure 13.14: (a) Suf·x trie Tfor the string X="minimize" . (b) Compact representation of T, where pair j:kdenotes slice X[j:k]in the reference string.
Using a Su·x Trie
The suf·x trie Tfor a string Xcan be used to ef·ciently perform pattern-matching queries on text X. Namely, we can determine whether a pattern Pis a substring ofXby trying to trace a path associated with PinT.Pis a substring of Xif and only if such a path can be traced. The search down the trie Tassumes that nodes in Tstore some additional information, with respect to the compact representation of the suf·x trie:
If node vhas label (j,k)and Yis the string of length yassociated with the path from the root to v(included), then X[k·y:k]= Y.
This property ensures that we can easily compute the start index of the pattern in the text when a match occurs.

Performance of the Edge List Structure
The performance of an edge list structure in ful·lling 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 nvertices and medges. Each individual vertex or edge instance uses O(1)space, and the additional lists Vand Euse 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 VorE,t h evertex
count
andedge
count methods run in O(1)time, and by iterating through the appropriate list, the methods vertices andedges run respectively in O(n)and O(m)time.
The most signi·cant 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) ,a n d 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 ef·ciently.
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 Eas an attribute of the edge. That stored position can later be used to locate and remove this edge from EinO(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 vis removed from the graph, all edges incident to vmust 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)

14.4 Transitive Closure

We have seen that graph traversals can be used to answer basic questions of reach-
ability in a directed graph. In particular, if we are interested in knowing whether
there is a path from vertex uto vertex vin a graph, we can perform a DFS or BFS
traversal starting at uand observe whether vis discovered. If representing a graph
with an adjacency list or adjacency map, we can answer the question of reachability
foruand vinO(n+m)time (see Propositions 14.15 and 14.17).

In certain applications, we may wish to answer many reachability queries more
ef·ciently, in which case it may be worthwhile to precompute a more convenientrepresentation of a graph. For ex
driving directions from an origin to a destination might be to assess whether the
destination is reachable. Similarly, in an electricity network, we may wish to beable to quickly determine whether
transitive closure of a directed graph /vectorGis itself a directed graph /vectorG
·such that the
vertices of /vectorG·are the same as the vertices of /vectorG,a n d/vectorG·has an edge (u,v), when-
ever/vectorGhas a directed path from utov(including the case where (u,v)is an edge of
the original /vectorG).

If a graph is represented as an adjacency list or adjacency map, we can compute
its transitive closure in O(n(n+m))time by making use of ngraph traversals, one
from each starting vertex. For example, a DFS starting at vertex ucan be used to
determine all vertices reachable from u, and thus a collection of edges originating
with uin the transitive closure.

In the remainder of this section, we explore an alternative technique for comput-
ing the transitive closure of a directed graph that is particularly well suited for when
a directed graph is represented by a data structure that supports O(1)-time lookup
for the get
edge(u,v) method (for example, the adjacency-matrix structure). Let /vectorG
be a directed graph with nvertices and medges. We compute the transitive closure
of/vectorGin a series of rounds. We initialize /vectorG0=/vectorG. We also arbitrarily number the
vertices of /vectorGasv1,v2,..., vn. We then begin the computation of the rounds, begin-
ning with round 1. In a generic round k, we construct directed graph /vectorGkstarting
with/vectorGk=/vectorGk·1and adding to /vectorGkthe directed edge (vi,vj)if directed graph /vectorGk·1
contains both the edges (vi,vk)and (vk,vj). In this way, we will enforce a simple
rule embodied in the proposition that follows.

Proposition 14.18: For i=1,..., n, directed graph /vectorGkhas an edge (vi,vj)if and
only if directed graph /vectorGhas a directed path from vitovj, whose intermediate
vertices (if any) are in the set {v1,..., vk}. In particular, /vectorGnis equal to /vectorG·,t h e
transitive closure of /vectorG.

15.3 External Searching and B-Trees
Consider the problem of maintaining a large collection of items that does not ·t in
main memory, such as a typical database. In this context, we refer to the secondary-memory blocks as disk bloc
secondary memory and primary memory as a disk transfer . Recalling the great
time difference that exists between main memory accesses and disk accesses, themain goal of maintaining such
count as the I/O complexity of the algorithm involved.
Some Ine·cient External-Memory Representations
A typical operation we would like to support is the search for a key in a map. If wewere to store nitems unordered
key within the list requires ntransfers in the worst case, since each link hop we
perform on the linked list might access a different block of memory.
We can reduce the number of block transfers by using an array-based sequence.
A sequential search of an array can be performed using only O(n/B)block transfers
because of spatial locality of reference, where Bdenotes the number of elements
that ·t into a block. This is because the block transfer when accessing the ·rst
element of the array actually retrieves the ·rst Belements, and so on with each
successive block. It is worth noting that the bound of O(n/B)transfers is only
achieved when using a compact array representation (see Section 5.2.2). The
standard Python listclass is a referential container, and so even though the sequence
of references are stored in an array, the actual elements that must be examinedduring a search are not generally
transfers in the worst case.
We could alternately store a sequence using a sorted array. In this case, a search
performs O(log
2n)transfers, via binary search, which is a nice improvement. But
we do not get signi·cant bene·t from block transfers because each query duringa binary search is likely in a differ
operations are expensive for a sorted array.
Since these simple implementations are I/O inef·cient, we should consider the
logarithmic-time internal-memory strategies that use balanced binary trees (for ex-
ample, AVL trees or red-black trees) or other search structures with logarithmicaverage-case query and update t
cally, each node accessed for a query or update in one of these structures will be in
a different block. Thus, these methods all require O(log
2n)transfers in the worst
case to perform a query or update operation. But we can do better! We can performmap queries and updates us
Bn)= O(logn/logB)transfers.

13.1 Abundance of Digitized Text
Despite the wealth of multimedia information, text processing remains one of the
dominant functions of computers. Computer are used to edit, store, and displaydocuments, and to transport docu
tems are used to archive a wide range of textual information, and new data is being
generated at a rapidly increasing pace. A large corpus can readily surpass a petabyteof data (which is equivalen
·Snapshots of the World Wide Web, as Internet document formats HTML andXML are primarily text formats, with
·All documents stored locally on a user·s computer
·Email archives
·Customer reviews
·Compilations of status updates on social networking sites such as Facebook
·Feeds from microblogging sites such as Twitter and Tumblr
These collections include written text from hundreds of international languages.
Furthermore, there are large data sets (such as DNA) that can be viewed computa-
tionally as ·strings· even though they are not language.
In this chapter we explore some of the fundamental algorithms that can be used
to ef·ciently analyze and process large textual data sets. In addition to having
interesting applications, text-processing algorithms also highlight some important
algorithmic design patterns.
We begin by examining the problem of searching for a pattern as a substring
of a larger piece of text, for example, when searching for a word in a document.
The pattern-matching problem gives rise to the brute-force method , which is often
inef·cient but has wide applicability.
Next, we introduce an algorithmic technique known as dynamic programming ,
which can be applied in certain settings to solve a problem in polynomial time that
appears at ·rst to require exponential time to solve. We demonstrate the applicationon this technique to the probl
be similar but not perfectly aligned. This problem arises when making suggestions
for a misspelled word, or when trying to match related genetic samples.
Because of the massive size of textual data sets, the issue of compression is
important, both in minimizing the number of bits that need to be communicated
through a network and to reduce the long-term storage requirements for archives.
For text compression, we can apply the greedy method , which often allows us to
approximate solutions to hard problems, and for some problems (such as in textcompression) actually gives rise
Finally, we examine several special-purpose data structures that can be used to
better organize textual data in order to support more ef·cient run-time queries.