12

Data Structures

Data structures are not so much algorithms as they are the fundamental constructs around which you build your application. Becoming fluent in what the standard data structures can do for you is essential to get full value from them.

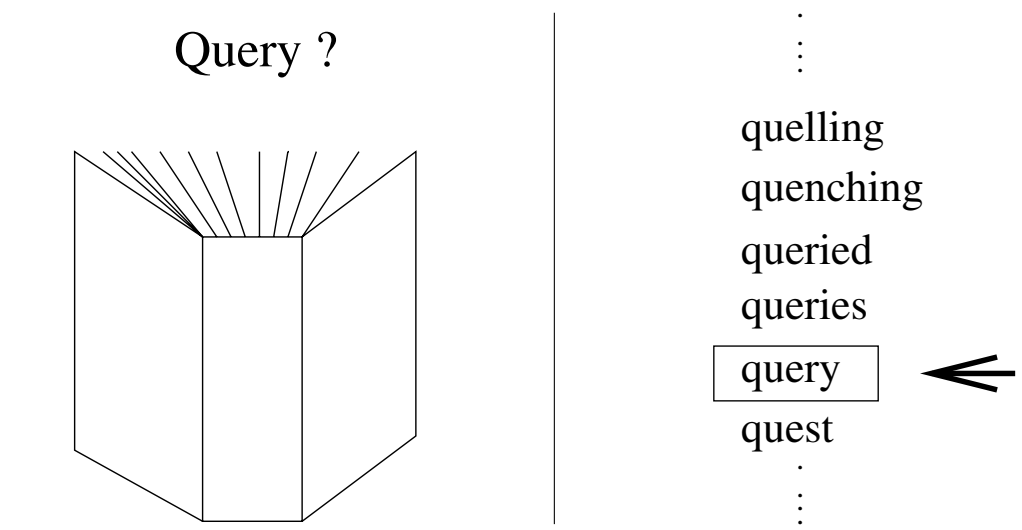
This puts data structures slightly out of sync with the rest of the catalog. Perhaps the most useful aspect of it will be the pointers to various implementations and data structure libraries. Many of these data structures are nontrivial to implement well, so the programs we point to will be useful as models even if they do not do exactly what you need. Certain fundamental data structures, like kd-trees and suffix trees, are not as well known as they should be. Hopefully, this catalog will serve to better publicize them.

There are a large number of books on elementary data structures available. My favorites include:

- *Sedgewick* [Sed98] – This comprehensive introduction to algorithms and data structures stands out for the clever and beautiful images of algorithms in action. It comes in C, C++, and Java editions.
- *Weiss* [Wei06] – A nice text, emphasizing data structures more than algorithms. Comes in Java, C, C++, and Ada editions.
- *Goodrich and Tamassia* [GT05] – The Java edition makes particularly good use of the author's Java Data Structures Library (JDSL).

The *Handbook of Data Structures and Applications* [MS05] provides a comprehensive and up-to-date survey of research in data structures. The student who took only an elementary course in data structures is likely to be impressed and surprised by the volume of recent work on the subject.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_12,



12.1 Dictionaries

Input description: A set of n records, each identified by one or more key fields.

Problem description: Build and maintain a data structure to efficiently locate, insert, and delete the record associated with any query key q.

Discussion: The abstract data type "dictionary" is one of the most important structures in computer science. Dozens of data structures have been proposed for implementing dictionaries, including hash tables, skip lists, and balanced/unbalanced binary search trees. This means that choosing the best one can be tricky. It can significantly impact performance. *However, in practice, it is more important to avoid using a bad data structure than to identify the single best option available.*

An essential piece of advice is to carefully isolate the implementation of the dictionary data structure from its interface. Use explicit calls to methods or subroutines that initialize, search, and modify the data structure, rather than embedding them within the code. This leads to a much cleaner program, but it also makes it easy to experiment with different implementations to see how they perform. Do not obsess about the procedure call overhead inherent in such an abstraction. If your application is so time-critical that such overhead can impact performance, then it is even more essential that you be able to identify the right dictionary implementation.

In choosing the right data structure for your dictionary, ask yourself the following questions:

- *How many items will you have in your data structure?* – Will you know this number in advance? Are you looking at a problem small enough that a simple data structure will suffice, or one so large that we must worry about running out of memory or virtual memory performance?

- *Do you know the relative frequencies of insert, delete, and search operations?* – Static data structures (like sorted arrays) suffice in applications when there are no modifications to the data structure after it is first constructed. *Semi-dynamic* data structures, which support insertion but not deletion, can have significantly simpler implementations than fully dynamic ones.

- *Can we assume that the access pattern for keys will be uniform and random?* – Search queries exhibit a skewed access distribution in many applications, meaning certain elements are much more popular than others. Further, queries often have a sense of temporal locality, meaning elements are likely to be repeatedly accessed in clusters instead of at fairly regular intervals. Certain data structures (such as splay trees) can take advantage of a skewed and clustered universe.

- *Is it critical that individual operations be fast, or only that the total amount of work done over the entire program be minimized?* – When response time is critical, such as in a program controlling a heart-lung machine, you can't wait too long between steps. When you have a program that is doing a lot of queries over the database, such as identifying all criminals who happen to be politicians, it is not as critical that you pick out any particular congressman quickly as it is that you get them all with the minimum total effort.

An object-oriented generation has emerged as no more likely to write a container class than fix the engine in their car. This is good; default containers should do just fine for most applications. Still, it is good sometimes to know what you have under the hood:

- *Unsorted linked lists or arrays* – For small data sets, an unsorted array is probably the easiest data structure to maintain. Linked structures can have terrible cache performance compared with sleek, compact arrays. However, once your dictionary becomes larger than (say) 50 to 100 items, the linear search time will kill you for either lists or arrays. Details of elementary dictionary implementations appear in Section 3.3 (page 72).

  A particularly interesting and useful variant is the *self-organizing list*. Whenever a key is accessed or inserted, we always move it to head of the list. Thus, if the key is accessed again sometime in the near future, it will be near the front and so require only a short search to find it. Most applications exhibit

both uneven access frequencies and locality of reference, so the average time for a successful search in a self-organizing list is typically much better than in a sorted or unsorted list. Of course, self-organizing data structures can be built from arrays as well as linked lists and trees.

- *Sorted linked lists or arrays* – Maintaining a sorted linked list is usually not worth the effort unless you are trying to eliminate duplicates, since we cannot perform binary searches in such a data structure. A sorted array will be appropriate if and only if there are not many insertions or deletions.

- *Hash tables* – For applications involving a moderate-to-large number of keys (say between 100 and 10,000,000), a hash table is probably the right way to go. We use a function that maps keys (be they strings, numbers, or whatever) to integers between 0 and $m-1$. We maintain an array of m buckets, each typically implemented using an unsorted linked list. The hash function immediately identifies which bucket contains a given key. If we use a hash function that spreads the keys out nicely, and a sufficiently large hash table, each bucket should contain very few items, thus making linear searches acceptable. Insertion and deletion from a hash table reduce to insertion and deletion from the bucket/list. Section 3.7 (page 89) provides a more detailed discussion of hashing and its applications.

  A well-tuned hash table will outperform a sorted array in most applications. However, several design decisions go into creating a well-tuned hash table:

  - *How do I deal with collisions?:* Open addressing can lead to more concise tables with better cache performance than bucketing, but performance will be more brittle as the load factor (ratio of occupancy to capacity) of the hash table starts to get high.
  - *How big should the table be?:* With bucketing, m should about the same as the maximum number of items you expect to put in the table. With open addressing, make it (say) 30% larger or more. Selecting m to be a prime number minimizes the dangers of a bad hash function.
  - *What hash function should I use?:* For strings, something like

    $$H(S,j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+j}) \bmod m$$

    should work, where α is the size of the alphabet and *char(x)* is the function that maps each character x to its ASCII character code. Use Horner's rule (or precompute values of α^x) to implement this hash function computation efficiently, as discussed in Section 13.9 (page 423). This hash function has the nifty property that

    $$H(S,j+1) = (H(S,j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

so hash codes of successive m-character windows of a string can be computed in constant time instead of $O(m)$.

Regardless of which hash function you decide to use, print statistics on the distribution of keys per bucket to see how uniform it *really* is. Odds are the first hash function you try will not prove to be the best. Botching up the hash function is an excellent way to slow down any application.

- *Binary search trees* – Binary search trees are elegant data structures that support fast insertions, deletions, and queries. They are reviewed in Section 3.4 (page 77). The big distinction between different types of trees is whether they are explicitly rebalanced after insertion or deletion, and how this rebalancing is done. In *random search trees*, we simply insert a node at the leaf position where we can find it and no rebalancing takes place. Although such trees perform well under random insertions, most applications are not really random. Indeed, unbalanced search trees constructed by inserting keys in sorted order are a disaster, performing like a linked list.

  Balanced search trees use local *rotation* operations to restructure search trees, moving more distant nodes closer to the root while maintaining the in-order search structure of the tree. Among balanced search trees, AVL and 2/3 trees are now passé, and *red-black trees* seem to be more popular. A particularly interesting self-organizing data structure is the *splay tree*, which uses rotations to move any accessed key to the root. Frequently used or recently accessed nodes thus sit near the top of the tree, allowing faster searches.

  Bottom line: Which tree is best for your application? Probably the one of which you have the best implementation. The flavor of balanced tree is probably not as important as the skill of the programmer who coded it.

- *B-trees* – For data sets so large that they will not fit in main memory (say more than 1,000,000 items) your best bet will be some flavor of a B-tree. Once a data structure has to be stored outside of main memory, the search time grows by several orders of magnitude. With modern cache architectures, similar effects can also happen on a smaller scale, since cache is much faster than RAM.

  The idea behind a B-tree is to collapse several levels of a binary search tree into a single large node, so that we can make the equivalent of several search steps before another disk access is needed. With B-tree we can access enormous numbers of keys using only a few disk accesses. To get the full benefit from using a B-tree, it is important to understand how the secondary storage device and virtual memory interact, through constants such as page size and virtual/real address space. *Cache-oblivious algorithms* (described below) can mitigate such concerns.

Even for modest-sized data sets, unexpectedly poor performance of a data structure may result from excessive swapping, so listen to your disk to help decide whether you should be using a B-tree.

- *Skip lists* – These are somewhat of a cult data structure. A hierarchy of sorted linked lists is maintained, where a coin is flipped for each element to decide whether it gets copied into the next highest list. This implies roughly $\lg n$ lists, each roughly half as large as the one above it. Search starts in the smallest list. The search key lies in an interval between two elements, which is then explored in the next larger list. Each searched interval contains an expected constant number of elements per list, for a total expected $O(\lg n)$ query time. The primary benefits of skip lists are ease of analysis and implementation relative to balanced trees.

Implementations: Modern programming languages provide libraries offering complete and efficient container implementations. The C++ *Standard Template Library* (STL) is now provided with most compliers, and available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01], and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

LEDA (see Section 19.1.1 (page 658)) provides an extremely complete collection of dictionary data structures in C++, including hashing, perfect hashing, B-trees, red-black trees, random search trees, and skip lists. Experiments reported in [MN99] identified hashing as the best dictionary choice, with skip lists and 2-4 trees (a special case of B-trees) as the most efficient tree-like structures.

Java Collections (JC), a small library of data structures, is included in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*). The *Data Structures Library in Java* (JDSL) is more comprehensive, and available for non-commercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL.

Notes: Knuth [Knu97a] provides the most detailed analysis and exposition on fundamental dictionary data structures, but misses certain modern data structures as red-black and splay trees. Spending some time with his books is a worthwhile rite of passage for all computer science students.

The Handbook of Data Structures and Applications [MS05] provides up-to-date surveys on all aspects of dictionary data structures. Other surveys include Mehlhorn and Tsakalidis [MT90b] and Gonnet and Baeza-Yates [GBY91]. Good textbook expositions on dictionary data structures include Sedgewick [Sed98], Weiss [Wei06], and Goodrich/Tamassia [GT05]. We defer to all these sources to avoid giving original references for each of the data structures described above.

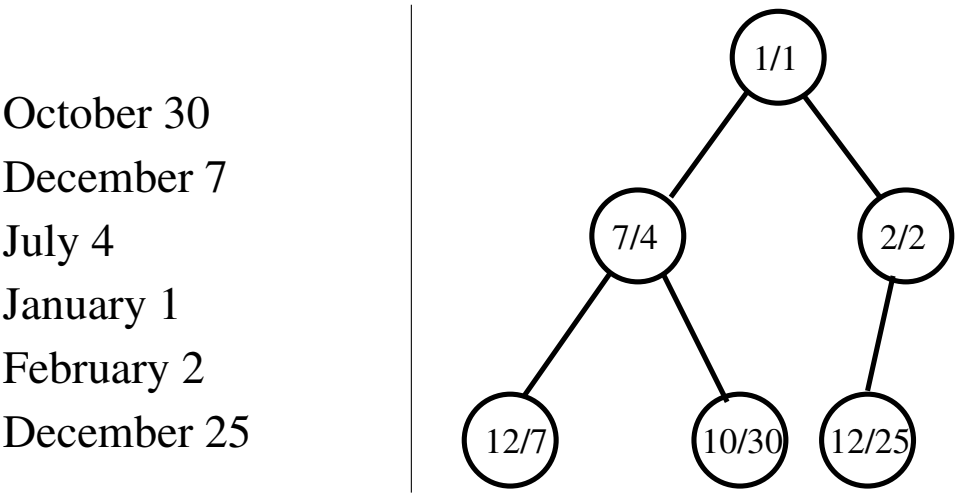
The 1996 DIMACS implementation challenge focused on elementary data structures, including dictionaries [GJM02]. Data sets, and codes are accessible from *http://dimacs.rutgers.edu/Challenges*.

The cost of transferring data back and forth between levels of the memory hierarchy (RAM-to-cache or disk-to-RAM) dominates the cost of actual computation for many

problems. Each data transfer moves one block of size b, so efficient algorithms seek to minimize the number of block transfers. The complexity of fundamental algorithm and data structure problems on such an external memory model has been extensively studied [Vit01]. *Cache-oblivious* data structures offer performance guarantees under such a model without explicit knowledge of the block-size parameter b. Hence, good performance can be obtained on any machine without architecture-specific tuning. See [ABF05] for an excellent survey on cache-oblivious data structures.

Several modern data structures, such as splay trees, have been studied using *amortized analysis*, where we bound the total amount of time used by any sequence of operations. In an amortized analysis, a single operation can be very expensive, but only because we have already benefited from enough cheap operations to pay off the higher cost. A data structure realizing an amortized complexity of $O(f(n))$ is less desirable than one whose worst-case complexity is $O(f(n))$ (since a very bad operation might still occur) but better than one with an average-case complexity $O(f(n))$, since the amortized bound will achieve this average on any input.

Related Problems: Sorting (see page 436), searching (see page 441).



INPUT OUTPUT

12.2 Priority Queues

Input description: A set of records with numerically or otherwise totally-ordered keys.

Problem description: Build and maintain a data structure for providing quick access to the *smallest* or *largest* key in the set.

Discussion: Priority queues are useful data structures in simulations, particularly for maintaining a set of future events ordered by time. They are called "priority" queues because they enable you to retrieve items not by the insertion time (as in a stack or queue), nor by a key match (as in a dictionary), but by which item has the highest priority of retrieval.

If your application will perform no insertions after the initial query, there is no need for an explicit priority queue. Simply sort the records by priority and proceed from top to bottom, maintaining a pointer to the last record retrieved. This situation occurs in Kruskal's minimum spanning tree algorithm, or when simulating a completely scripted set of events.

However, if you are mixing insertions, deletions, and queries, you will need a real priority queue. The following questions will help select the right one:

- *What other operations do you need?* – Will you be searching for arbitrary keys, or just searching for the smallest? Will you be deleting arbitrary elements from the data, or just repeatedly deleting the top or smallest element?

- *Do you know the maximum data structure size in advance?* – The issue here is whether you can preallocate space for the data structure.
- *Might you change the priority of elements already in the queue?* – Changing the priority of elements implies that we must be able to retrieve elements from the queue based on their key, in addition to being able to retrieve the largest element.

Your choices are between the following basic priority queue implementations:

- *Sorted array or list* – A sorted array is very efficient to both identify the smallest element and delete it by decrementing the top index. However, maintaining the total order makes inserting new elements slow. Sorted arrays are only suitable when there will be few insertions into the priority queue. Basic priority queue implementations are reviewed in Section 3.5 (page 83).
- *Binary heaps* – This simple, elegant data structure supports both insertion and extract-min in $O(\lg n)$ time each. Heaps maintain an implicit binary tree structure in an array, such that the key of the root of any subtree is less than that of all its descendents. Thus, the minimum key always sits at the top of the heap. New keys can be inserted by placing them at an open leaf and percolating the element upwards until it sits at its proper place in the partial order. An implementation of binary heap construction and retrieval in C appears in Section 4.3.1 (page 109)

  Binary heaps are the right answer when you know an upper bound on the number of items in your priority queue, since you must specify array size at creation time. Even this constraint can be mitigated by using dynamic arrays (see Section 3.1.1 (page 66)).
- *Bounded height priority queue* – This array-based data structure permits constant-time insertion and find-min operations whenever the range of possible key values is limited. Suppose we know that all key values will be integers between 1 and n. We can set up an array of n linked lists, such that the ith list serves as a bucket containing all items with key i. We will maintain a *top* pointer to the smallest nonempty list. To insert an item with key k into the priority queue, add it to the kth bucket and set $top = \min(top, k)$. To extract the minimum, report the first item from bucket *top*, delete it, and move *top* down if the bucket has become empty.

  Bounded height priority queues are very useful in maintaining the vertices of a graph sorted by degree, which is a fundamental operation in graph algorithms. Still, they are not as widely known as they should be. They are usually the right priority queue for any small, discrete range of keys.
- *Binary search trees* – Binary search trees make effective priority queues, since the smallest element is always the leftmost leaf, while the largest element is

always the rightmost leaf. The min (max) is found by simply tracing down left (right) pointers until the next pointer is nil. Binary tree heaps prove most appropriate when you need other dictionary operations, or if you have an unbounded key range and do not know the maximum priority queue size in advance.

- *Fibonacci and pairing heaps* – These complicated priority queues are designed to speed up *decrease-key* operations, where the priority of an item already in the priority queue is reduced. This arises, for example, in shortest path computations when we discover a shorter route to a vertex v than previously established.

  Properly implemented and used, they lead to better performance on very large computations.

Implementations: Modern programming languages provide libraries offering complete and efficient priority queue implementations. Member functions `push`, `top`, and `pop` of the C++ *Standard Template Library* (STL) `priority_queue` template mirror heap operations `insert`, `findmax`, and `deletemax`. STL is available with documentation at *http://www.sgi.com/tech/stl/*. See Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL.

LEDA (see Section 19.1.1 (page 658)) provides a complete collection of priority queues in C++, including Fibonacci heaps, pairing heaps, Emde-Boas trees, and bounded height priority queues. Experiments reported in [MN99] identified simple binary heaps as quite competitive in most applications, with pairing heaps beating Fibonacci heaps in head-to-head tests.

The *Java Collections* `PriorityQueue` class is included in the java.util package of Java standard edition (*http://java.sun.com/javase/*). The *Data Structures Library in Java* (JDSL) provides an alternate implementation, and is available for non-commercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL.

Sanders [San00] did extensive experiments demonstrating that his sequence heap, based on k-way merging, was roughly twice as fast as a well-implemented binary heap. See *http://www.mpi-inf.mpg.de/∼sanders/programs/spq/* for his implementations in C++.

Notes: *The Handbook of Data Structures and Applications* [MS05] provides several up-to-date surveys on all aspects of priority queues. Empirical comparisons between priority queue data structures include [CGS99, GBY91, Jon86, LL96, San00].

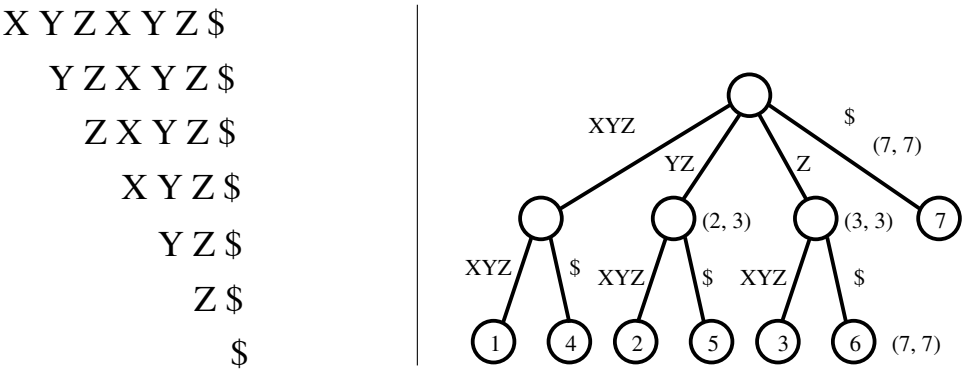
Double-ended priority queues extend the basic heap operations to simultaneously support both find-min and find-max. See [Sah05] for a survey of four different implementations of double-ended priority queues.

Bounded-height priority queues are useful data structures in practice, but do not promise good worst-case performance when arbitrary insertions and deletions are permitted. However, von Emde Boas priority queues [vEBKZ77] support $O(\lg \lg n)$ insertion, deletion, search, max, and min operations where each key is an element from 1 to n.

Fibonacci heaps [FT87] support insert and decrease-key operations in constant amortized time, with $O(\lg n)$ amortized time extract-min and delete operations. The constant-time decrease-key operation leads to faster implementations of classical algorithms for shortest-paths, weighted bipartite-matching, and minimum spanning tree. In practice, Fibonacci heaps are difficult to implement and have large constant factors associated with them. However, pairing heaps appear to realize the same bounds with less overhead. Experiments with pairing heaps are reported in [SV87].

Heaps define a partial order that can be built using a linear number of comparisons. The familiar linear-time merging algorithm for heap construction is due to Floyd [Flo64]. In the worst case, $1.625n$ comparisons suffice [GM86] and $1.5n - O(\lg n)$ comparisons are necessary [CC92].

Related Problems: Dictionaries (see page 367), sorting (see page 436), shortest path (see page 489).



INPUT OUTPUT

12.3 Suffix Trees and Arrays

Input description: A reference string S.

Problem description: Build a data structure to quickly find all places where an arbitrary query string q occurs in S.

Discussion: Suffix trees and arrays are phenomenally useful data structures for solving string problems elegantly and efficiently. Proper use of suffix trees often speeds up string processing algorithms from $O(n^2)$ to linear time—likely the answer. Indeed, suffix trees are the hero of the war story reported in Section 3.9 (page 94).

In its simplest instantiation, a suffix tree is simply a *trie* of the n suffixes of an n-character string S. A trie is a tree structure, where each edge represents one character, and the root represents the null string. Thus, each path from the root represents a string, described by the characters labeling the edges traversed. Any finite set of words defines a trie, and two words with common prefixes branch off from each other at the first distinguishing character. Each leaf denotes the end of a string. Figure 12.1 illustrates a simple trie.

Tries are useful for testing whether a given query string q is in the set. We traverse the trie from the root along branches defined by successive characters of q. If a branch does not exist in the trie, then q cannot be in the set of strings. Otherwise we find q in $|q|$ character comparisons *regardless* of how many strings are in the trie. Tries are very simple to build (repeatedly insert new strings) and very fast to search, although they can be expensive in terms of memory.

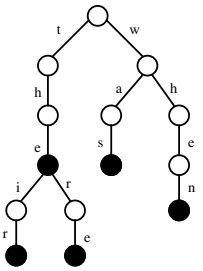


Figure 12.1: A trie on strings *the, their, there, was,* and *when*

A *suffix tree* is simply a trie of all the proper suffixes of S. The suffix tree enables you to test whether q is a substring of S, because any substring of S is the prefix of some suffix (got it?). The search time is again linear in the length of q.

The catch is that constructing a full suffix tree in this manner can require $O(n^2)$ time and, even worse, $O(n^2)$ space, since the average length of the n suffixes is $n/2$. However, linear space suffices to represent a full suffix tree, if we are clever. Observe that most of the nodes in a trie-based suffix tree occur on simple paths between branch nodes in the tree. Each of these simple paths corresponds to a substring of the original string. By storing the original string in an array and collapsing each such path into a single edge, we have all the information of the full suffix tree in only $O(n)$ space. The label for each edge is described by the starting and ending array indices representing the substring. The output figure for this section displays a collapsed suffix tree in all its glory.

Even better, there exist $O(n)$ algorithms to construct this collapsed tree, by making clever use of pointers to minimize construction time. These additional pointers can also be used to speed up many applications of suffix trees.

But what can you do with suffix trees? Consider the following applications. For more details see the books by Gusfield [Gus97] or Crochemore and Rytter [CR03]:

- *Find all occurrences of q as a substring of S* – Just as with a trie, we can walk from the root to the node n_q associated with q. The positions of all occurrences of q in S are represented by the descendents of n_q, which can be identified using a depth-first search from n_q. In collapsed suffix trees, it takes $O(|q| + k)$ time to find the k occurrences of q in S.

- *Longest substring common to a set of strings* – Build a single collapsed suffix tree containing all suffixes of all strings, with each leaf labeled with its original string. In the course of doing a depth-first search on this tree, we can label each node with both the length of its common prefix and the number of distinct strings that are children of it. From this information, the best node can be selected in linear time.

- *Find the longest palindrome in S* – A *palindrome* is a string that reads the same if the order of characters is reversed, such as *madam*. To find the longest palindrome in a string S, build a single suffix tree containing all suffixes of S and the reversal of S, with each leaf identified by its starting position. A palindrome is defined by any node in this tree that has forward and reversed children from the same position.

Since linear time suffix tree construction algorithms are nontrivial, I recommend using an existing implementation. Another good option is to use suffix arrays, discussed below.

Suffix arrays do most of what suffix trees do, while using roughly four times less memory. They are also easier to implement. A suffix array is in principle just an array that contains all the n suffixes of S in sorted order. Thus a binary search of this array for string q suffices to locate the prefix of a suffix that matches q, permitting an efficient substring search in $O(\lg n)$ string comparisons. With the addition of an index specifying the common prefix length of all bounding suffixes, only $\lg n + |q|$ *character* comparisons need be performed on any query, since we can identify the next character that must be tested in the binary search. For example, if the lower range of the search is *cowabunga* and the upper range is *cowslip*, all keys in between must share the same first three letters, so only the fourth character of any intermediate key must be tested against q. In practice, suffix arrays are typically as fast or faster to search than suffix trees.

Suffix arrays use less memory than suffix trees. Each suffix is represented completely by its unique starting position (from 1 to n) and read off as needed using a single reference copy of the input string.

Some care must be taken to construct suffix arrays efficiently, however, since there are $O(n^2)$ characters in the strings being sorted. One solution is to first build a suffix *tree*, then perform an in-order traversal of it to read the strings off in sorted order! However, recent breakthroughs have lead to space/time efficient algorithms for constructing suffix arrays directly.

Implementations: There now exist a wealth of suffix array implementations available. Indeed, all of the recent linear time construction algorithms have been implemented and benchmarked [PST07]. Schürmann and Stoye [SS07] provide an excellent C implementation at *http://bibiserv.techfak.uni-bielefeld.de/bpr/*.

No less than eight different C/C++ implementations of compressed text indexes appear at the *Pizza&Chili corpus* (*http://pizzachili.di.unipi.it/*). These data structures go to great lengths to minimize space usage, typically compressing the input string to near the empirical entropy while still achieving excellent query times!

Suffix tree implementations are also readily available. A `SuffixTree` class is provided in BioJava (*ttp://www.biojava.org/*)—an open source project providing a Java framework for processing biological data. `Libstree` is a C implementation of Ukkonen's algorithm, available at *http://www.icir.org/christian/libstree/*.

Nelson's C++ code [Nel96] is available from *http://marknelson.us/1996/08/01/suffix-trees/*.

Strmat is a collection of C programs implementing exact pattern matching algorithms in association with [Gus97], including an implementation of suffix trees. It is available at *http://www.cs.ucdavis.edu/~gusfield/strmat.html*.

Notes: Tries were first proposed by Fredkin [Fre62], the name coming from the central letters of the word "retrieval." A survey of basic trie data structures with extensive references appears in [GBY91].

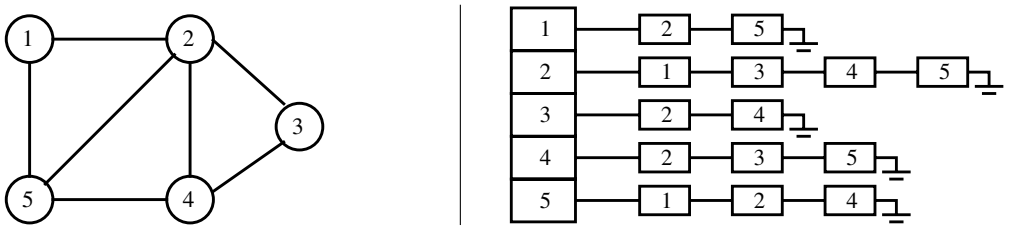
Efficient algorithms for suffix tree construction are due to Weiner [Wei73], McCreight [McC76], and Ukkonen [Ukk92]. Good expositions on these algorithms include Crochmore and Rytter [CR03] and Gusfield [Gus97].

Suffix arrays were invented by Manber and Myers [MM93], although an equivalent idea called *Pat trees* due to Gonnet and Baeza-Yates appears in [GBY91]. Three teams independently emerged with linear-time suffix array algorithms in 2003 [KSPP03, KA03, KSB05], and progress has continued rapidly. See [PST07] for a recent survey covering all these developments.

Recent work has resulted in the development of compressed full text indexes that offer essentially all the power of suffix trees/arrays in a data structure whose size is proportional to the *compressed* text string. Makinen and Navarro [MN07] survey these remarkable data structures.

The power of suffix trees can be further augmented by using a data structure for computing the *least common ancestor (LCA)* of any pair of nodes x, y in a tree in constant time, after linear-time preprocessing of the tree. The original data structure due to Harel and Tarjan [HT84], has been progressively simplified by Schieber and Vishkin [SV88] and later Bender and Farach [BF00]. Expositions include Gusfield [Gus97]. The least common ancestor of two nodes in a suffix tree or trie defines the node representing the longest common prefix of the two associated strings. That we can answer such queries in constant time is amazing, and proves useful as a building block for many other algorithms.

Related Problems: String matching (see page 628), text compression (see page 637), longest common substring (see page 650).



12.4 Graph Data Structures

Input description: A graph G.

Problem description: Represent the graph G using a flexible, efficient data structure.

Discussion: The two basic data structures for representing graphs are *adjacency matrices* and *adjacency lists*. Full descriptions of these data structures appear in Section 5.2 (page 151), along with an implementation of adjacency lists in C. In general, for most things, adjacency lists are the way to go.

The issues in deciding which data structure to use include:

- *How big will your graph be?* – How many vertices will it have, both typically and in the worse case? Ditto for the number of edges? Graphs with 1,000 vertices imply adjacency matrices with 1,000,000 entries. This seems to be the boundary of reality. Adjacency matrices make sense only for small or very dense graphs.

- *How dense will your graph be?* – If your graph is very dense, meaning that a large fraction of the vertex pairs define edges, there is probably no compelling reason to use adjacency lists. You will be doomed to using $\Theta(n^2)$ space anyway. Indeed, for complete graphs, matrices will be more concise due to the elimination of pointers.

- *Which algorithms will you be implementing?* – Certain algorithms are more natural on adjacency matrices (such as all-pairs shortest path) and others favor adjacency lists (such as most DFS-based algorithms). Adjacency matrices win for algorithms that repeatedly ask, "Is (i, j) in G?" However, most graph algorithms can be designed to eliminate such queries.

- *Will you be modifying the graph over the course of your application?* – Efficient *static graph* implementations can be used when no edge insertion/deletion operations will done following initial construction. Indeed, more

common than modifying the topology of the graph is modifying the *attributes* of a vertex or edge of the graph, such as size, weight, label, or color. Attributes are best handled as extra fields in the vertex or edge records of adjacency lists.

Building a good general purpose graph type is a substantial project. For this reason, we suggest that you check out existing implementations (particularly LEDA) before hacking up your own. Note that it costs only time linear in the size of the larger data structure to convert between adjacency matrices and adjacency lists. This conversion is unlikely to be the bottleneck in any application, so you may decide to use both data structures if you have the space to store them. This usually isn't necessary, but might prove simplest if you are confused about the alternatives.

Planar graphs are those that can be drawn in the plane so no two edges cross. Graphs arising in many applications are planar by definition, such as maps of countries. Others are planar by happenstance, like trees. Planar graphs are always sparse, since any n-vertex planar graph can have at most $3n - 6$ edges. Thus they should be represented using adjacency lists. If the planar drawing (or *embedding*) of the graph is fundamental to what is being computed, planar graphs are best represented geometrically. See Section 15.12 (page 520) for algorithms for constructing planar embeddings from graphs. Section 17.15 (page 614) discusses algorithms for maintaining the graphs implicit in the arrangements of geometric objects like lines and polygons.

Hypergraphs are generalized graphs where each edge may link subsets of more than two vertices. Suppose we want to represent who is on which Congressional committee. The vertices of our hypergraph would be the individual congressmen, while each hyperedge would represent one committee. Such arbitrary collections of subsets of a set are naturally thought of as hypergraphs.

Two basic data structures for hypergraphs are:

- *Incidence matrices*, which are analogous to adjacency matrices. They require $n \times m$ space, where m is the number of hyperedges. Each row corresponds to a vertex, and each column to an edge, with a nonzero entry in $M[i, j]$ iff vertex i is incident to edge j. On standard graphs there are two nonzero entries in each column. The degree of each vertex governs the number of nonzero entries in each row.

- *Bipartite incidence structures*, which are analogous to adjacency lists, and hence suited for sparse hypergraphs. There is a vertex of the incidence structure associated with each edge and vertex of the hypergraphs, and an edge (i, j) in the incidence structure if vertex i of the hypergraph appears in edge j of the hypergraph. Adjacency lists are typically used to represent this incidence structure. Drawing the associated bipartite graph provides a natural way to visualize the hypergraph.

Special efforts must be taken to represent very large graphs efficiently. However, interesting problems have been solved on graphs with millions of edges and

vertices. The first step is to make your data structure as lean as possible, by packing your adjacency matrix in a bit vector (see Section 12.5 (page 385)) or removing unnecessary pointers from your adjacency list representation. For example, in a static graph (which does not support edge insertions or deletions) each edge list can be replaced by a packed array of vertex identifiers, thus eliminating pointers and potentially saving half the space.

If your graph is extremely large, it may become necessary to switch to a hierarchical representation, where the vertices are clustered into subgraphs that are compressed into single vertices. Two approaches exist to construct such a hierarchical decomposition. The first breaks the graph into components in a natural or application-specific way. For example, a graph of roads and cities suggests a natural decomposition—partition the map into districts, towns, counties, and states. The other approach runs a graph partition algorithm discussed as in Section 16.6 (page 541). If you have one, a natural decomposition will likely do a better job than some naive heuristic for an NP-complete problem. If your graph is really unmanageably large, you cannot afford to do a very good job of algorithmically partitioning it. First verify that standard data structures fail on your problem before attempting such heroic measures.

Implementations: LEDA (see Section 19.1.1 (page 658)) provides the best graph data type currently implemented in C++. It is now a commercial product. You should at least study the methods it provides for graph manipulation, so as to see how the right level of abstract graph type makes implementing algorithms very clean and easy.

The C++ Boost Graph Library [SLL02] (*http://www.boost.org/libs/graph/doc*) is more readily available. Implementations of adjacency lists, matrices, and edge lists are included, along with a reasonable library of basic graph algorithms. Its interface and components are generic in the same sense as the C++ standard template library (STL).

JUNG (*http://jung.sourceforge.net/*) is a Java graph library particularly popular in the social networks community. The *Data Structures Library in Java* (JDSL) provides a comprehensive implementation with a decent algorithm library, and is available for noncommercial use at *http://www.jdsl.org/*. See [GTV05, GT05] for more detailed guides to JDSL. JGraphT (*http://jgrapht.sourceforge.net/*) is a more recent development with similar functionality.

The Stanford Graphbase (see Section 19.1.8 (page 660)) provides a simple but flexible graph data structure in CWEB, a literate version of the C language. It is instructive to see what Knuth does and does not place in his basic data structure, although we recommend other implementations as a better basis for further development.

My (biased) preference in C language graph types is the library from my book *Programming Challenges* [SR03]. See Section 19.1.10 (page 661) for details. Simple graph data structures in Mathematica are provided by *Combinatorica* [PS03], with a library of algorithms and display routines. See Section 19.1.9 (page 661).

Notes: The advantages of adjacency list data structures for graphs became apparent with the linear-time algorithms of Hopcroft and Tarjan [HT73b, Tar72]. The basic adjacency list and matrix data structures are presented in essentially all books on algorithms or data structures, including [CLRS01, AHU83, Tar83]. Hypergraphs are presented in Berge [Ber89]

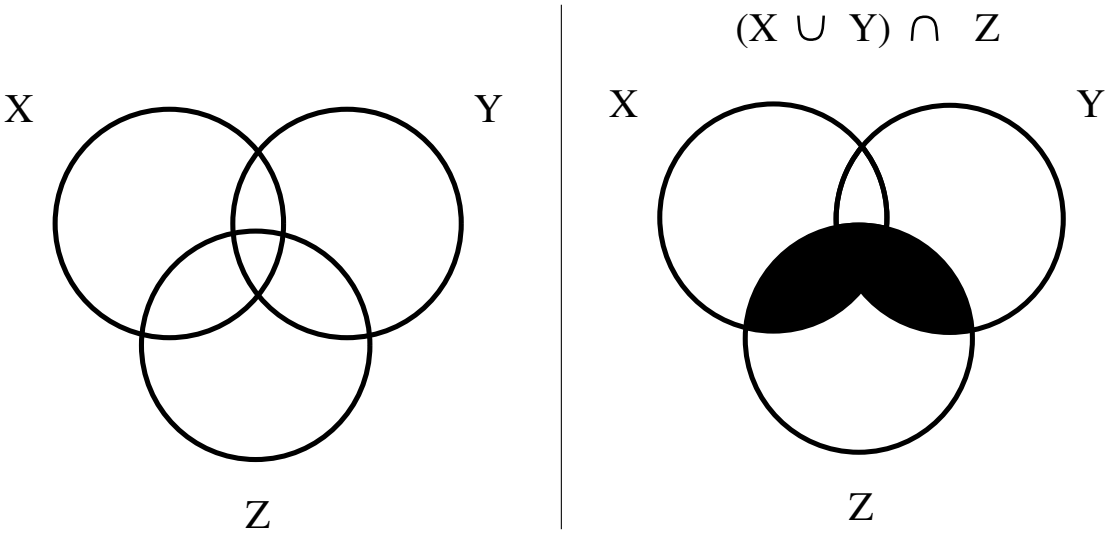
The improved efficiency of static graph types was revealed by Naher and Zlotowski [NZ02], who sped up certain LEDA graph algorithms by a factor of four by simply switching to a more compact graph structure.

An interesting question concerns minimizing the number of bits needed to represent arbitrary graphs on n vertices, particularly if certain operations must be supported efficiently. Such issues are surveyed in [vL90b].

Dynamic graph algorithms are data structures that maintain quick access to an invariant (such as minimum spanning tree or connectivity) under edge insertion and deletion. *Sparsification* [EGIN92] is a general approach to constructing dynamic graph algorithms. See [ACI92, Zar02] for experimental studies on the practicality of dynamic graph algorithms.

Hierarchically-defined graphs arise often in VLSI design problems, because designers make extensive use of cell libraries [Len90]. Algorithms specifically for hierarchically-defined graphs include planarity testing [Len89], connectivity [LW88], and minimum spanning trees [Len87a].

Related Problems: Set data structures (see page 385), graph partition (see page 541).



12.5 Set Data Structures

Input description: A universe of items $U = \{u_1, \ldots, u_n\}$ on which is defined a collection of subsets $S = \{S_1, \ldots, S_m\}$.

Problem description: Represent each subset so as to efficiently (1) test whether $u_i \in S_j$, (2) compute the union or intersection of S_i and S_j, and (3) insert or delete members of S.

Discussion: In mathematical terms, a set is an unordered collection of objects drawn from a fixed universal set. However, it is usually useful for implementation to represent each set in a single *canonical order*, typically sorted, to speed up or simplify various operations. Sorted order turns the problem of finding the union or intersection of two subsets into a linear-time operation—just sweep from left to right and see what you are missing. It makes possible element searching in sublinear time. Finally, printing the elements of a set in a canonical order paradoxically reminds us that order really doesn't matter.

We distinguish sets from two other kinds of objects: dictionaries and strings. A collection of objects *not* drawn from a fixed-size universal set is best thought of as a *dictionary*, discussed in Section 12.1 (page 367). Strings are structures where order matters—i.e., if $\{A, B, C\}$ is not the same as $\{B, C, A\}$. Sections 12.3 and 18 discuss data structures and algorithms for strings.

Multisets permit elements to have more than one occurrence. Data structures for sets can generally be extended to multisets by maintaining a count field or linked list of equivalent entries for each element.

If each subset contains exactly two elements, they can be thought of as edges in a graph whose vertices represent the universal set. A system of subsets with no restrictions on the cardinality of its members is called a *hypergraph*. It is worth considering whether your problem has a graph-theoretical analogy, like connected components or shortest path in a graph/hypergraph.

Your primary alternatives for representing arbitrary subsets are:

- *Bit vectors* – An n-bit vector or array can represent any subset S on a universal set U containing n items. Bit i will be 1 if $i \in S$, and 0 if not. Since only one bit is needed per element, bit vectors can be very space efficient for surprisingly large values of $|U|$. Element insertion and deletion simply flips the appropriate bit. Intersection and union are done by "and-ing" or "or-ing" the bits together. The only drawback of a bit vector is its performance on sparse subsets. For example, it takes $O(n)$ time to explicitly identify all members of sparse (even empty) subset S.

- *Containers or dictionaries* – A subset can also be represented using a linked list, array, or dictionary containing exactly the elements in the subset. No notion of a fixed universal set is needed for such a data structure. For sparse subsets, dictionaries can be more space and time efficient than bit vectors, and easier to work with and program. For efficient union and intersection operations, it pays to keep the elements in each subset sorted, so a linear-time traversal through both subsets identifies all duplicates.

- *Bloom filters* – We can emulate a bit vector in the absence of a fixed universal set by hashing each subset element to an integer from 0 to n and setting the corresponding bit. Thus, bit $H(e)$ will be 1 if $e \in S$. Collisons leave some possibility for error under this scheme, however, because a different key might have hashed to the same position.

  Bloom filters use several (say k) different hash functions $H_1, \ldots H_k$, and set all k bits $H_i(e)$ upon insertion of key e. Now e is in S only if all k bits are 1. The probability of false positives can be made arbitrarily low by increasing the number of hash functions k and table size n. With the proper constants, each subset element can be represented using a constant number of bits independent of the size of the universal set.

  This hashing-based data structure is much more space-efficient than dictionaries for static subset applications that can tolerate a small probability of error. Many can. For instance, a spelling checker that left a rare random string undetected would prove no great tragedy.

Many applications involve collections of subsets that are pairwise disjoint, meaning that each element is in exactly one subset. For example, consider maintaining

the connected components of a graph or the party affiliations of politicians. Each vertex/hack is in exactly one component/party. Such a system of subsets is called a *set partition*. Algorithms for generating partitions of a given set are provided in Section 14.6 (page 456).

The primary issue with set partition data structures is maintaining changes over time, perhaps as edges are added or party members defect. Typical queries include "which set is a particular item in?" and "are two items in the same set?" as we modify the set by (1) changing one item, (2) merging or unioning two sets, or (3) breaking a set apart. Your basic options are:

- *Collection of containers* – Representing each subset in its own container/dictionary permits fast access to all the elements in the subset, which facilitates union and intersection operations. The cost comes in membership testing, as we must search each subset data structure independently until we find our target.

- *Generalized bit vector* – Let the ith element of an array denote the number/name of the subset that contains it. Set identification queries and single element modifications can be performed in constant time. However, operations like performing the union of two subsets take time proportional to the size of the universe, since each element in the two subsets must be identified and (at least one subset's worth) must have its name changed.

- *Dictionary with a subset attribute* – Similarly, each item in a binary tree can be associated a field that records the name of the subset it is in. Set identification queries and single element modifications can be performed in the time it takes to search in the dictionary. However, union/intersection operations are again slow. The need to perform such union operations quickly provides the motivation for the ...

- *Union-find data structure* – We represent a subset using a rooted tree where each node points to its *parent* instead of its children. The name of each subset will be the name of the item at the root. Finding out which subset we are in is simple, for we keep traversing up the parent pointers until we hit the root. Unioning two subsets is also easy. Just assign the root of one of two trees to point to the other, so now *all* elements have the same root and hence the same subset name.

  Implementation details have a big impact on asymptotic performance here. Always selecting the larger (or taller) tree as the root in a merger guarantees logarithmic height trees, as presented with our implementation in Section 6.1.3 (page 198). Retraversing the path traced on each find and explicitly pointing all nodes on the path to the root (called *path compression*) reduces the tree to almost constant height. Union find is a fast, simple data structure that every programmer should know about. It does not support breaking up subsets created by unions, but usually this is not an issue.

Implementations: Modern programming languages provide libraries offering complete and efficient set implementations. The C++ *Standard Template Library* (STL) provides `set` and `multiset` containers. *Java Collections* (JC) contains `HashSet` and `TreeSet` containers and is included in the `java.util` package of Java standard edition.

LEDA (see Section 19.1.1 (page 658)) provides efficient dictionary data structures, sparse arrays, and union-find data structures to maintain set partitions, all in C++.

Implementation of union-find underlies any implementation of Kruskal's minimum spanning tree algorithm. For this reason, all the graph libraries of Section 12.4 (page 381) presumably contains an implementation. Minimum spanning tree codes are described in Section 15.3 (page 484).

The computer algebra system *REDUCE* (*http://www.reduce-algebra.com/*) contains `SETS`, a package supporting set-theoretic operations on both explicit and implicit (symbolic) sets. Other computer algebra systems may support similar functionality.

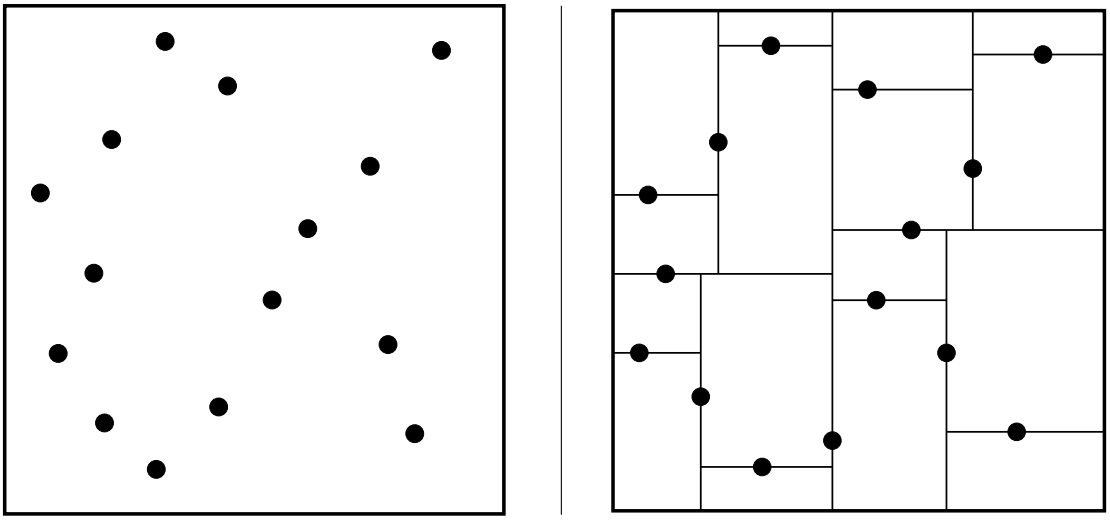
Notes: Optimal algorithms for such set operations as intersection and union were presented in [Rei72]. Raman [Ram05] provides an excellent survey on data structures for set operations on a variety of different operations. Bloom filters are ably surveyed in [BM05], with recent experimental results presented in [PSS07].

Certain balanced tree data structures support merge/meld/link/cut operations, which permit fast ways to union and intersect disjoint subsets. See Tarjan [Tar83] for a nice presentation of such structures. Jacobson [Jac89] augmented the bit-vector data structure to support select operations (where is the ith 1 bit?) efficiently in both time and space.

Galil and Italiano [GI91] survey data structures for disjoint set union. The upper bound of $O(m\alpha(m,n))$ on m union-find operations on an n-element set is due to Tarjan [Tar75], as is a matching lower bound on a restricted model of computation [Tar79]. The inverse Ackerman function $\alpha(m,n)$ grows notoriously slowly, so this performance is close to linear. An interesting connection between the worst-case of union-find and the length of Davenport-Schinzel sequences—a combinatorial structure that arises in computational geometry—is established in [SA95].

The *power set* of a set S is the collection of all $2^{|S|}$ subsets of S. Explicit manipulation of power sets quickly becomes intractable due to their size. Implicit representations of power sets in symbolic form becomes necessary for nontrivial computations. See [BCGR92] for algorithms on and computational experience with symbolic power set representations.

Related Problems: Generating subsets (see page 452), generating partitions (see page 456), set cover (see page 621), minimum spanning tree (see page 484).



INPUT OUTPUT

12.6 Kd-Trees

Input description: A set S of n points or more complicated geometric objects in k dimensions.

Problem description: Construct a tree that partitions space by half-planes such that each object is contained in its own box-shaped region.

Discussion: Kd-tree and related spatial data structures hierarchically decompose space into a small number of cells, each containing a few representatives from an input set of points. This provides a fast way to access any object by position. We traverse down the hierarchy until we find the smallest cell containing it, and then scan through the objects in this cell to identify the right one.

Typical algorithms construct kd-trees by partitioning point sets. Each node in the tree is defined by a plane cutting through one of the dimensions. Ideally, this plane equally partitions the subset of points into left/right (or up/down) subsets. These children are again partitioned into equal halves, using planes through a different dimension. Partitioning stops after $\lg n$ levels, with each point in its own leaf cell.

The cutting planes along any path from the root to another node defines a unique box-shaped region of space. Each subsequent plane cuts this box into two boxes. Each box-shaped region is defined by $2k$ planes, where k is the number of dimensions. Indeed, the "kd" in *kd-tree* is short for "k-dimensional." We maintain

the region of interest defined by the intersection of these half-spaces as we move down the tree.

Flavors of kd-trees differ in exactly how the splitting plane is selected. Options include:

- *Cycling through the dimensions* – partition first on d_1, then $d_2, \ldots, d_k$ before cycling back to d_1.
- *Cutting along the largest dimension* – select the partition dimension to make the resulting boxes as square or cube-like as possible. Selecting a plane to partition the points in half does not mean selecting a splitter in the middle of the box-shaped regions, since all the points may lie in the left side of the box.
- *Quadtrees or Octtrees* – Instead of partitioning with single planes, use all axis-parallel planes that pass through a given partition point. In two dimensions, this means creating four child cells; in 3D, this means eight child cells. Quadtrees seem particularly popular on image data, where leaf cells imply that all pixels in the regions have the same color.
- *BSP-trees – Binary space partitions* use general (i.e., not just axis-parallel) cutting planes to carve up space into cells so that each cell ends up containing only one object (say a polygon). Such partitions are not possible using only axis-parallel cuts for certain sets of objects. The downside is that such polyhedral cell boundaries are more complicated to work with than boxes.
- *R-trees* – This is another spatial data structure useful for geometric objects that cannot be partitioned into axis-oriented boxes without cutting them into pieces. At each level, the objects are partitioned into a small number of (possibly-overlapping) boxes to construct searchable hierarchies without partitioning objects.

Ideally, our partitions split both the space (ensuring fat, regular regions) and the set of points (ensuring a log height tree) evenly, but doing both simultaneously can be impossible on a given input. The advantages of fat cells become clear in many applications of kd-trees:

- *Point location* – To identify which cell a query point q lies in, we start at the root and test which side of the partition plane contains q. By repeating this process on the appropriate child node, we travel down the tree to find the leaf cell containing q in time proportional to its height. See Section 17.7 (page 587) for more on point location.
- *Nearest neighbor search* – To find the point in S closest to a query point q, we perform point location to find the cell c containing q. Since c is bordered by some point p, we can compute the distance $d(p, q)$ from p to q. Point p is likely

close to q, but it might not be the single closest neighbor. Why? Suppose q lies right at the boundary of a cell. Then q's nearest neighbor might lie just to the left of the boundary in another cell. Thus, we must traverse all cells that lie within a distance of $d(p, q)$ of cell c and verify that none of them contain closer points. In trees with nice, fat cells, very few cells should need to be tested. See Section 17.5 (page 580) for more on nearest neighbor search.

- *Range search* – Which points lie within a query box or region? Starting from the root, check whether the query region intersects (or contains) the cell defining the current node. If it does, check the children; if not, none of the leaf cells below this node can possibly be of interest. We quickly prune away irrelevant portions of the space. Section 17.6 (page 584) focuses on range search.

- *Partial key search* – Suppose we want to find a point p in S, but we do not have full information about p. Say we are looking for someone of age 35 and height 5'8" but of unknown weight in a 3D-tree with dimensions of age, weight, and height. Starting from the root, we can identify the correct descendant for all but the weight dimension. To be sure we find the right point, we must search *both children* of these nodes. The more fields we know the better, but such partial key search can be substantially faster than checking all points against the key.

Kd-trees are most useful for a small to moderate number of dimensions, say from 2 up to maybe 20 dimensions. They lose effectiveness as the dimensionality increases, primarily because the ratio of the volume of a unit sphere in k-dimensions shrinks exponentially compared to the unit cube. Thus exponentially many cells will have to be searched within a given radius of a query point, say for nearest-neighbor search. Also, the number of neighbors for any cell grows to $2k$ and eventually become unmanageable.

The bottom line is that you should try to avoid working in high-dimensional spaces, perhaps by discarding (or projecting away) the least important dimensions.

Implementations: *KDTREE 2* contains C++ and Fortran 95 implementations of *kd*-trees for efficient nearest nearest neighbor search in many dimensions. See *http://arxiv.org/abs/physics/0408067*.

Samet's spatial index demos (*http://donar.umiacs.umd.edu/quadtree/*) provide a series of Java applets illustrating many variants of *kd*-trees, in association with his book [Sam06].

Terralib (*http://www.terralib.org/*) is an open source geographic information system (GIS) software library written in C++. This includes an implementation of spatial data structures.

The 1999 DIMACS implementation challenge focused on data structures for nearest neighbor search [GJM02]. Data sets and codes are accessible from *http://dimacs.rutgers.edu/Challenges*.

Notes: Samet [Sam06] is the best reference on kd-trees and other spatial data structures. All major (and many minor) variants are developed in substantial detail. A shorter survey [Sam05] is also available. Bentley [Ben75] is generally credited with developing kd-trees, although they have the murky history associated with most folk data structures.

The performance of spatial data structures degrades with high dimensionality. Projecting high-dimensional spaces onto a random lower-dimensional hyperplane has recently emerged as a simple but powerful method for dimensionality reduction. Both theoretical [IM04] and empirical [BM01] results indicate that this method preserves distances quite nicely.

Algorithms that quickly produce a point provably close to the query point are a recent development in higher-dimensional nearest neighbor search. A sparse weighted-graph structure is built from the data set, and the nearest neighbor is found by starting at a random point and walking greedily in the graph towards the query point. The closest point found during several random trials is declared the winner. Similar data structures hold promise for other problems in high-dimensional spaces. See [AM93, AMN+98].

Related Problems: Nearest-neighbor search (see page 580), point location (see page 587), range search (see page 584).