3

Data Structures

Changing a data structure in a slow program can work the same way an organ transplant does in a sick patient. Important classes of *abstract data types* such as containers, dictionaries, and priority queues, have many different but functionally equivalent *data structures* that implement them. Changing the data structure does not change the correctness of the program, since we presumably replace a correct implementation with a different correct implementation. However, the new implementation of the data type realizes different tradeoffs in the time to execute various operations, so the total performance can improve dramatically. Like a patient in need of a transplant, only one part might need to be replaced in order to fix the problem.

But it is better to be born with a good heart than have to wait for a replacement. The maximum benefit from good data structures results from designing your program around them in the first place. We assume that the reader has had some previous exposure to elementary data structures and pointer manipulation. Still, data structure (CS II) courses these days focus more on data abstraction and object orientation than the nitty-gritty of how structures should be represented in memory. We will review this material to make sure you have it down.

In data structures, as with most subjects, it is more important to really understand the basic material than have exposure to more advanced concepts. We will focus on each of the three fundamental abstract data types (containers, dictionaries, and priority queues) and see how they can be implemented with arrays and lists. Detailed discussion of the tradeoffs between more sophisticated implementations is deferred to the relevant catalog entry for each of these data types.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_3,

3.1 Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either *contiguous* or *linked*, depending upon whether they are based on arrays or pointers:

- *Contiguously-allocated structures* are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.

- *Linked data structures* are composed of distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

In this section, we review the relative advantages of contiguous and linked data structures. These tradeoffs are more subtle than they appear at first glance, so I encourage readers to stick with me here even if you may be familiar with both types of structures.

3.1.1 Arrays

The *array* is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.

A good analogy likens an array to a street full of houses, where each array element is equivalent to a house, and the index is equivalent to the house number. Assuming all the houses are equal size and numbered sequentially from 1 to n, we can compute the exact position of each house immediately from its address.[1]

Advantages of contiguously-allocated arrays include:

- *Constant-time access given the index* – Because the index of each element maps directly to a particular memory address, we can access arbitrary data items instantly provided we know the index.

- *Space efficiency* – Arrays consist purely of data, so no space is wasted with links or other formatting information. Further, end-of-record information is not needed because arrays are built from fixed-size records.

- *Memory locality* – A common programming idiom involves iterating through all the elements of a data structure. Arrays are good for this because they exhibit excellent memory locality. Physical continuity between successive data accesses helps exploit the high-speed *cache memory* on modern computer architectures.

The downside of arrays is that we cannot adjust their size in the middle of a program's execution. Our program will fail soon as we try to add the $(n +$

[1] Houses in Japanese cities are traditionally numbered in the order they were built, not by their physical location. This makes it extremely difficult to locate a Japanese address without a detailed map.

1)st customer, if we only allocate room for n records. We can compensate by allocating extremely large arrays, but this can waste space, again restricting what our programs can do.

Actually, we *can* efficiently enlarge arrays as we need them, through the miracle of *dynamic arrays*. Suppose we start with an array of size 1, and double its size from m to $2m$ each time we run out of space. This doubling process involves allocating a new contiguous array of size $2m$, copying the contents of the old array to the lower half of the new one, and returning the space used by the old array to the storage allocation system.

The apparent waste in this procedure involves the recopying of the old contents on each expansion. How many times might an element have to be recopied after a total of n insertions? Well, the first inserted element will have been recopied when the array expands after the first, second, fourth, eighth, ... insertions. It will take $\log_2 n$ doublings until the array gets to have n positions. However, most elements do not suffer much upheaval. Indeed, the $(n/2 + 1)$st through nth elements will move at most once and might never have to move at all.

If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is given by

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

Thus, each of the n elements move only two times on average, and the total work of managing the dynamic array is the same $O(n)$ as it would have been if a single array of sufficient size had been allocated in advance!

The primary thing lost using dynamic arrays is the guarantee that each array access takes constant time *in the worst case*. Now all the queries will be fast, except for those relatively few queries triggering array doubling. What we get instead is a promise that the nth array access will be completed quickly enough that the *total* effort expended so far will still be $O(n)$. Such *amortized* guarantees arise frequently in the analysis of data structures.

3.1.2 Pointers and Linked Structures

Pointers are the connections that hold the pieces of linked structures together. Pointers represent the address of a location in memory. A variable storing a pointer to a given data item can provide more freedom than storing a copy of the item itself. A cell-phone number can be thought of as a pointer to its owner as they move about the planet.

Pointer syntax and power differ significantly across programming languages, so we begin with a quick review of pointers in C language. A pointer `p` is assumed to

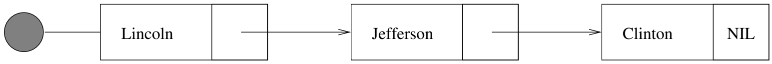

Figure 3.1: Linked list example showing data and pointer fields

give the address in memory where a particular chunk of data is located.[2] Pointers in C have types declared at compile time, denoting the data type of the items they can point to. We use `*p` to denote the item that is pointed to by pointer `p`, and `&x` to denote the address of (i.e., pointer to) of a particular variable `x`. A special `NULL` pointer value is used to denote structure-terminating or unassigned pointers.

All linked data structures share certain properties, as revealed by the following linked list type declaration:

```
typedef struct list {
        item_type item;                 /* data item */
        struct list *next;              /* point to successor */
} list;
```

In particular:

- Each node in our data structure (here `list`) contains one or more data fields (here `item`) that retain the data that we need to store.
- Each node contains a pointer field to at least one other node (here `next`). This means that much of the space used in linked data structures has to be devoted to pointers, not data.
- Finally, we need a pointer to the head of the structure, so we know where to access it.

The list is the simplest linked structure. The three basic operations supported by lists are searching, insertion, and deletion. In *doubly-linked lists*, each node points both to its predecessor and its successor element. This simplifies certain operations at a cost of an extra pointer field per node.

Searching a List

Searching for item x in a linked list can be done iteratively or recursively. We opt for recursively in the implementation below. If x is in the list, it is either the first element or located in the smaller rest of the list. Eventually, we reduce the problem to searching in an empty list, which clearly cannot contain x.

[2] C permits direct manipulation of memory addresses in ways which may horrify Java programmers, but we will avoid doing any such tricks.

```
list *search_list(list *l, item_type x)
{
        if (l == NULL) return(NULL);

        if (l->item == x)
                return(l);
        else
                return( search_list(l->next, x) );
}
```

Insertion into a List

Insertion into a singly-linked list is a nice exercise in pointer manipulation, as shown below. Since we have no need to maintain the list in any particular order, we might as well insert each new item in the simplest place. Insertion at the beginning of the list avoids any need to traverse the list, but does require us to update the pointer (denoted `l`) to the head of the data structure.

```
void insert_list(list **l, item_type x)
{
        list *p;                        /* temporary pointer */

        p = malloc( sizeof(list) );
        p->item = x;
        p->next = *l;
        *l = p;
}
```

Two C-isms to note. First, the `malloc` function allocates a chunk of memory of sufficient size for a new node to contain `x`. Second, the funny double star (`**l`) denotes that `l` is a *pointer to a pointer* to a list node. Thus the last line, `*l=p;` copies `p` to the place pointed to by `l`, which is the external variable maintaining access to the head of the list.

Deletion From a List

Deletion from a linked list is somewhat more complicated. First, we must find a pointer to the *predecessor* of the item to be deleted. We do this recursively:

```
list *predecessor_list(list *l, item_type x)
{
        if ((l == NULL) || (l->next == NULL)) {
                //predecessor sought on null list
                return(NULL);
        }

        if ((l->next)->item == x)
                return(l);
        else
                return( predecessor_list(l->next, x) );
}
```

The predecessor is needed because it points to the doomed node, so its `next` pointer must be changed. The actual deletion operation is simple, once ruling out the case that the to-be-deleted element does not exist. Special care must be taken to reset the pointer to the head of the list (`l`) when the first element is deleted:

```
delete_list(list **l, item_type x)
{
        list *p;                        /* item pointer */
        list *pred;                     /* predecessor pointer */
        list *search_list(), *predecessor_list();

        p = search_list(*l,x);
        if (p != NULL) {
                pred = predecessor_list(*l,x);
                if (pred == NULL)       /* splice out of list */
                        *l = p->next;
                else
                        pred->next = p->next;

                free(p);                /* free memory used by node */
        }
}
```

C language requires explicit deallocation of memory, so we must `free` the deleted node after we are finished with it to return the memory to the system.

3.1.3 Comparison

The relative advantages of linked lists over static arrays include:

- Overflow on linked structures can never occur unless the memory is actually full.

- Insertions and deletions are *simpler* than for contiguous (array) lists.

- With large records, moving pointers is easier and faster than moving the items themselves.

while the relative advantages of arrays include:

- Linked structures require extra space for storing pointer fields.

- Linked lists do not allow efficient random access to items.

- Arrays allow better memory locality and cache performance than random pointer jumping.

> *Take-Home Lesson:* Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.

One final thought about these fundamental structures is that they can be thought of as recursive objects:

- *Lists* – Chopping the first element off a linked list leaves a smaller linked list. This same argument works for strings, since removing characters from string leaves a string. Lists are recursive objects.

- *Arrays* – Splitting the first k elements off of an n element array gives two smaller arrays, of size k and $n-k$, respectively. Arrays are recursive objects.

This insight leads to simpler list processing, and efficient divide-and-conquer algorithms such as quicksort and binary search.

3.2 Stacks and Queues

We use the term *container* to denote a data structure that permits storage and retrieval of data items *independent of content*. By contrast, dictionaries are abstract data types that retrieve based on key values or content, and will be discussed in Section 3.3 (page 72).

Containers are distinguished by the particular retrieval order they support. In the two most important types of containers, this retrieval order depends on the insertion order:

- *Stacks* – Support retrieval by last-in, first-out (LIFO) order. Stacks are simple to implement and very efficient. For this reason, stacks are probably the right container to use when retrieval order doesn't matter at all, such as when processing batch jobs. The *put* and *get* operations for stacks are usually called *push* and *pop*:

- *Push(x,s)*: Insert item x at the top of stack s.
- *Pop(s)*: Return (and remove) the top item of stack s.

LIFO order arises in many real-world contexts. People crammed into a subway car exit in LIFO order. Food inserted into my refrigerator usually exits the same way, despite the incentive of expiration dates. Algorithmically, LIFO tends to happen in the course of executing recursive algorithms.

- *Queues* – Support retrieval in first in, first out (FIFO) order. This is surely the fairest way to control waiting times for services. You want the container holding jobs to be processed in FIFO order to minimize the *maximum* time spent waiting. Note that the *average* waiting time will be the same regardless of whether FIFO or LIFO is used. Many computing applications involve data items with infinite patience, which renders the question of maximum waiting time moot.

  Queues are somewhat trickier to implement than stacks and thus are most appropriate for applications (like certain simulations) where the order is important. The *put* and *get* operations for queues are usually called *enqueue* and *dequeue*.

  - *Enqueue(x,q)*: Insert item x at the back of queue q.
  - *Dequeue(q)*: Return (and remove) the front item from queue q.

  We will see queues later as the fundamental data structure controlling breadth-first searches in graphs.

Stacks and queues can be effectively implemented using either arrays or linked lists. The key issue is whether an upper bound on the size of the container is known in advance, thus permitting the use of a statically-allocated array.

3.3 Dictionaries

The *dictionary* data type permits access to data items by content. You stick an item into a dictionary so you can find it when you need it.

The primary operations of dictionary support are:

- *Search(D,k)* – Given a search key k, return a pointer to the element in dictionary D whose key value is k, if one exists.
- *Insert(D,x)* – Given a data item x, add it to the set in the dictionary D.
- *Delete(D,x)* – Given a pointer to a given data item x in the dictionary D, remove it from D.

Certain dictionary data structures also efficiently support other useful operations:

- *Max(D)* or *Min(D)* – Retrieve the item with the largest (or smallest) key from D. This enables the dictionary to serve as a priority queue, to be discussed in Section 3.5 (page 83).

- *Predecessor(D,x)* or *Successor(D,x)* – Retrieve the item from D whose key is immediately before (or after) x in sorted order. These enable us to iterate through the elements of the data structure.

Many common data processing tasks can be handled using these dictionary operations. For example, suppose we want to remove all duplicate names from a mailing list, and print the results in sorted order. Initialize an empty dictionary D, whose search key will be the record name. Now read through the mailing list, and for each record *search* to see if the name is already in D. If not, *insert* it into D. Once finished, we must extract the remaining names out of the dictionary. By starting from the first item *Min(D)* and repeatedly calling *Successor* until we obtain *Max(D)*, we traverse all elements in sorted order.

By defining such problems in terms of abstract dictionary operations, we avoid the details of the data structure's representation and focus on the task at hand.

In the rest of this section, we will carefully investigate simple dictionary implementations based on arrays and linked lists. More powerful dictionary implementations such as binary search trees (see Section 3.4 (page 77)) and hash tables (see Section 3.7 (page 89)) are also attractive options in practice. A complete discussion of different dictionary data structures is presented in the catalog in Section 12.1 (page 367). We encourage the reader to browse through the data structures section of the catalog to better learn what your options are.

Stop and Think: Comparing Dictionary Implementations (I)

Problem: What are the asymptotic worst-case running times for each of the seven fundamental dictionary operations (search, insert, delete, successor, predecessor, minimum, and maximum) when the data structure is implemented as:

- An unsorted array.

- A sorted array.

Solution: This problem (and the one following it) reveals some of the inherent tradeoffs of data structure design. A given data representation may permit efficient implementation of certain operations at the cost that other operations are expensive.

In addition to the array in question, we will assume access to a few extra variables such as n—the number of elements currently in the array. Note that we must *maintain* the value of these variables in the operations where they change (e.g., insert and delete), and charge these operations the cost of this maintenance.

The basic dictionary operations can be implemented with the following costs on unsorted and sorted arrays, respectively:

Dictionary operation	Unsorted array	Sorted array
Search(L, k)	$O(n)$	$O(\log n)$
Insert(L, x)	$O(1)$	$O(n)$
Delete(L, x)	$O(1)^*$	$O(n)$
Successor(L, x)	$O(n)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(1)$
Minimum(L)	$O(n)$	$O(1)$
Maximum(L)	$O(n)$	$O(1)$

We must understand the implementation of each operation to see why. First, we discuss the operations when maintaining an *unsorted* array A.

- *Search* is implemented by testing the search key k against (potentially) each element of an unsorted array. Thus, search takes linear time in the worst case, which is when key k is not found in A.

- *Insertion* is implemented by incrementing n and then copying item x to the nth cell in the array, $A[n]$. The bulk of the array is untouched, so this operation takes constant time.

- *Deletion* is somewhat trickier, hence the superscript(*) in the table. The definition states that we are given a pointer x to the element to delete, so we need not spend any time searching for the element. But removing the xth element from the array A leaves a hole that must be filled. We could fill the hole by moving each of the elements $A[x+1]$ to $A[n]$ up one position, but this requires $\Theta(n)$ time when the first element is deleted. The following idea is better: just write over $A[x]$ with $A[n]$, and decrement n. This only takes constant time.

- The definition of the traversal operations, *Predecessor* and *Successor*, refer to the item appearing before/after x *in sorted order*. Thus, the answer is not simply $A[x-1]$ (or $A[x+1]$), because in an unsorted array an element's physical predecessor (successor) is not necessarily its logical predecessor (successor). Instead, the predecessor of $A[x]$ is the biggest element smaller than $A[x]$. Similarly, the successor of $A[x]$ is the smallest element larger than $A[x]$. Both require a sweep through all n elements of A to determine the winner.

- *Minimum* and *Maximum* are similarly defined with respect to sorted order, and so require linear sweeps to identify in an unsorted array.

Implementing a dictionary using a *sorted* array completely reverses our notions of what is easy and what is hard. Searches can now be done in $O(\log n)$ time, using binary search, because we know the median element sits in $A[n/2]$. Since the upper and lower portions of the array are also sorted, the search can continue recursively on the appropriate portion. The number of halvings of n until we get to a single element is $\lceil \lg n \rceil$.

The sorted order also benefits us with respect to the other dictionary retrieval operations. The minimum and maximum elements sit in $A[1]$ and $A[n]$, while the predecessor and successor to $A[x]$ are $A[x-1]$ and $A[x+1]$, respectively.

Insertion and deletion become more expensive, however, because making room for a new item or filling a hole may require moving many items arbitrarily. Thus both become linear-time operations. ■

> *Take-Home Lesson:* Data structure design must balance all the different operations it supports. The fastest data structure to support both operations A and B may well not be the fastest structure to support either operation A or B.

Stop and Think: Comparing Dictionary Implementations (II)

Problem: What is the asymptotic worst-case running times for each of the seven fundamental dictionary operations when the data structure is implemented as

- A singly-linked unsorted list.
- A doubly-linked unsorted list.
- A singly-linked sorted list.
- A doubly-linked sorted list.

Solution: Two different issues must be considered in evaluating these implementations: singly- vs. doubly-linked lists and sorted vs. unsorted order. Subtle operations are denoted with a superscript:

Dictionary operation	Singly unsorted	Double unsorted	Singly sorted	Doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)^*$	$O(1)$	$O(n)^*$	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)^*$	$O(1)$
Minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Maximum(L)	$O(n)$	$O(n)$	$O(1)^*$	$O(1)$

As with unsorted arrays, search operations are destined to be slow while maintenance operations are fast.

- *Insertion/Deletion* – The complication here is deletion from a singly-linked list. The definition of the *Delete* operation states we are given a pointer x to the item to be deleted. But what we *really* need is a pointer to the element pointing to x in the list, because that is the node that needs to be changed. We can do nothing without this list predecessor, and so must spend linear time searching for it on a singly-linked list. Doubly-linked lists avoid this problem, since we can immediately retrieve the list predecessor of x.

  Deletion is faster for sorted doubly-linked lists than sorted arrays, because splicing out the deleted element from the list is more efficient than filling the hole by moving array elements. The predecessor pointer problem again complicates deletion from singly-linked sorted lists.

- *Search* – Sorting provides less benefit for linked lists than it did for arrays. Binary search is no longer possible, because we can't access the median element without traversing all the elements before it. What sorted lists *do* provide is quick termination of unsuccessful searches, for if we have not found *Abbott* by the time we hit *Costello* we can deduce that he doesn't exist. Still, searching takes linear time in the worst case.

- *Traversal operations* – The predecessor pointer problem again complicates implementing *Predecessor*. The logical successor is equivalent to the node successor for both types of sorted lists, and hence can be implemented in constant time.

- *Maximum* – The maximum element sits at the tail of the list, which would normally require $\Theta(n)$ time to reach in either singly- or doubly-linked lists.

  However, we can maintain a separate pointer to the list tail, provided we pay the maintenance costs for this pointer on every insertion and deletion. The tail pointer can be updated in constant time on doubly-linked lists: on insertion check whether `last->next` still equals `NULL`, and on deletion set `last` to point to the list predecessor of `last` if the last element is deleted.

  We have no efficient way to find this predecessor for singly-linked lists. So why can we implement maximum in $\Theta(1)$ on singly-linked lists? The trick is to charge the cost to each deletion, which *already* took linear time. Adding an extra linear sweep to update the pointer does not harm the asymptotic complexity of *Delete*, while gaining us *Maximum* in constant time as a reward for clear thinking. ∎

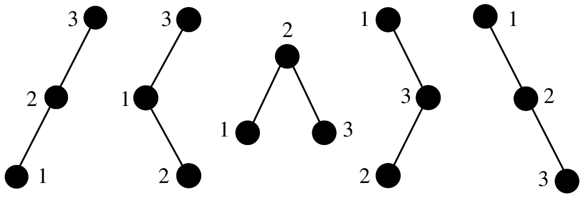

Figure 3.2: The five distinct binary search trees on three nodes

3.4 Binary Search Trees

We have seen data structures that allow fast search or flexible update, but not fast search *and* flexible update. Unsorted, doubly-linked lists supported insertion and deletion in $O(1)$ time but search took linear time in the worse case. Sorted arrays support binary search and logarithmic query times, but at the cost of linear-time update.

Binary search requires that we have fast access to *two elements*—specifically the median elements above and below the given node. To combine these ideas, we need a "linked list" with two pointers per node. This is the basic idea behind binary search trees.

A *rooted binary tree* is recursively defined as either being (1) empty, or (2) consisting of a node called the root, together with two rooted binary trees called the left and right subtrees, respectively. The order among "brother" nodes matters in rooted trees, so left is different from right. Figure 3.2 gives the shapes of the five distinct binary trees that can be formed on three nodes.

A binary *search* tree labels each node in a binary tree with a single key such that for any node labeled x, all nodes in the left subtree of x have keys $< x$ while all nodes in the right subtree of x have keys $> x$. This search tree labeling scheme is very special. For any binary tree on n nodes, and any set of n keys, there is *exactly* one labeling that makes it a binary search tree. The allowable labelings for three-node trees are given in Figure 3.2.

3.4.1 Implementing Binary Search Trees

Binary tree nodes have *left* and *right* pointer fields, an (optional) *parent* pointer, and a data field. These relationships are shown in Figure 3.3; a type declaration for the tree structure is given below:

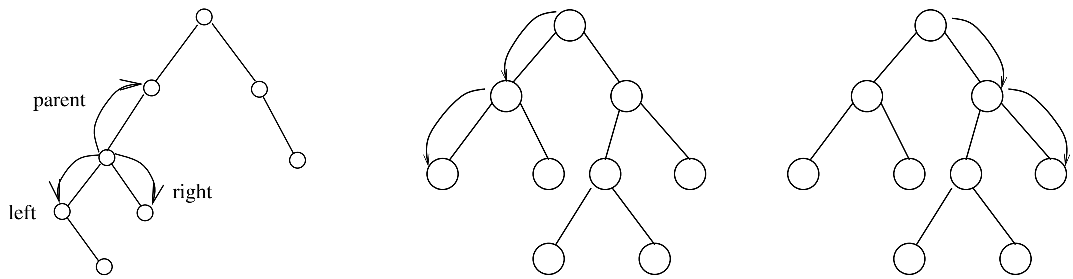

Figure 3.3: Relationships in a binary search tree (left). Finding the minimum (center) and maximum (right) elements in a binary search tree

```
typedef struct tree {
        item_type item;                 /* data item */
        struct tree *parent;            /* pointer to parent */
        struct tree *left;              /* pointer to left child */
        struct tree *right;             /* pointer to right child */
} tree;
```

The basic operations supported by binary trees are searching, traversal, insertion, and deletion.

Searching in a Tree

The binary search tree labeling uniquely identifies where each key is located. Start at the root. Unless it contains the query key x, proceed either left or right depending upon whether x occurs before or after the root key. This algorithm works because both the left and right subtrees of a binary search tree are themselves binary search trees. This recursive structure yields the recursive search algorithm below:

```
tree *search_tree(tree *l, item_type x)
{
        if (l == NULL) return(NULL);

        if (l->item == x) return(l);

        if (x < l->item)
                return( search_tree(l->left, x) );
        else
                return( search_tree(l->right, x) );
}
```

This search algorithm runs in $O(h)$ time, where h denotes the height of the tree.

Finding Minimum and Maximum Elements in a Tree

Implementing the *find-minimum* operation requires knowing where the minimum element is in the tree. By definition, the smallest key must reside in the left subtree of the root, since all keys in the left subtree have values less than that of the root. Therefore, as shown in Figure 3.3, the minimum element must be the leftmost descendent of the root. Similarly, the maximum element must be the rightmost descendent of the root.

```
tree *find_minimum(tree *t)
{
        tree *min;                      /* pointer to minimum */

        if (t == NULL) return(NULL);

        min = t;
        while (min->left != NULL)
                min = min->left;
        return(min);
}
```

Traversal in a Tree

Visiting all the nodes in a rooted binary tree proves to be an important component of many algorithms. It is a special case of traversing all the nodes and edges in a graph, which will be the foundation of Chapter 5.

A prime application of tree traversal is listing the labels of the tree nodes. Binary search trees make it easy to report the labels in sorted order. By definition, all the keys smaller than the root must lie in the left subtree of the root, and all keys bigger than the root in the right subtree. Thus, visiting the nodes recursively in accord with such a policy produces an *in-order* traversal of the search tree:

```
void traverse_tree(tree *l)
{
        if (l != NULL) {
                traverse_tree(l->left);
                process_item(l->item);
                traverse_tree(l->right);
        }

}
```

Each item is processed once during the course of traversal, which runs in $O(n)$ time, where n denotes the number of nodes in the tree.

Alternate traversal orders come from changing the position of `process_item` relative to the traversals of the left and right subtrees. Processing the item first yields a *pre-order* traversal, while processing it last gives a *post-order* traversal. These make relatively little sense with search trees, but prove useful when the rooted tree represents arithmetic or logical expressions.

Insertion in a Tree

There is only one place to insert an item x into a binary search tree T where we know we can find it again. We must replace the `NULL` pointer found in T after an unsuccessful query for the key k.

This implementation uses recursion to combine the search and node insertion stages of key insertion. The three arguments to `insert_tree` are (1) a pointer `l` to the pointer linking the search subtree to the rest of the tree, (2) the key `x` to be inserted, and (3) a `parent` pointer to the parent node containing `l`. The node is allocated and linked in on hitting the `NULL` pointer. Note that we pass the *pointer* to the appropriate left/right pointer in the node during the search, so the assignment `*l = p;` links the new node into the tree:

```
insert_tree(tree **l, item_type x, tree *parent)
{
    tree *p;                          /* temporary pointer */

    if (*l == NULL) {
           p = malloc(sizeof(tree)); /* allocate new node */
           p->item = x;
           p->left = p->right = NULL;
           p->parent = parent;
           *l = p;                    /* link into parent's record */
           return;
    }

    if (x < (*l)->item)
           insert_tree(&((*l)->left), x, *l);
    else
           insert_tree(&((*l)->right), x, *l);
}
```

Allocating the node and linking it in to the tree is a constant-time operation after the search has been performed in $O(h)$ time.

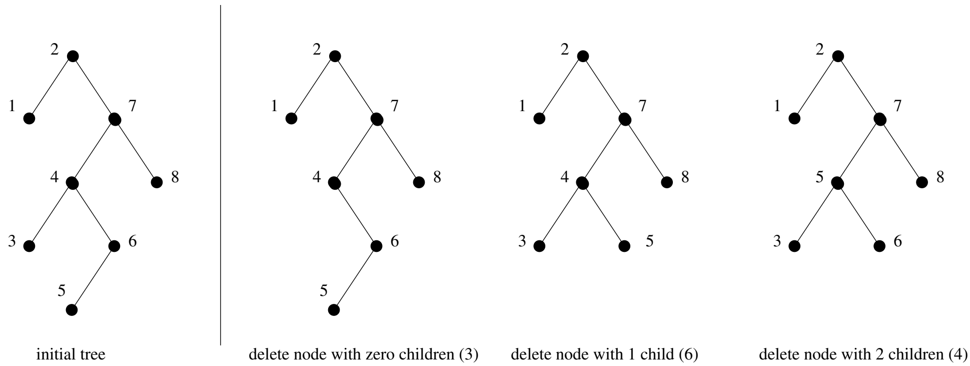

Figure 3.4: Deleting tree nodes with 0, 1, and 2 children

Deletion from a Tree

Deletion is somewhat trickier than insertion, because removing a node means appropriately linking its two descendant subtrees back into the tree somewhere else. There are three cases, illustrated in Figure 3.4. Leaf nodes have no children, and so may be deleted by simply clearing the pointer to the given node.

The case of the doomed node having one child is also straightforward. There is one parent and one grandchild, and we can link the grandchild directly to the parent without violating the in-order labeling property of the tree.

But what of a to-be-deleted node with two children? Our solution is to relabel this node with the key of its immediate successor in sorted order. This successor must be the smallest value in the right subtree, specifically the leftmost descendant in the right subtree. Moving this to the point of deletion results in a properly-labeled binary search tree, and reduces our deletion problem to physically removing a node with at most one child—a case that has been resolved above.

The full implementation has been omitted here because it looks a little ghastly, but the code follows logically from the description above.

The worst-case complexity analysis is as follows. Every deletion requires the cost of at most two search operations, each taking $O(h)$ time where h is the height of the tree, plus a constant amount of pointer manipulation.

3.4.2 How Good Are Binary Search Trees?

When implemented using binary search trees, all three dictionary operations take $O(h)$ time, where h is the height of the tree. The smallest height we can hope for occurs when the tree is perfectly balanced, where $h = \lceil \log n \rceil$. This is very good, but the tree must be perfectly balanced.

Our insertion algorithm puts each new item at a leaf node where it should have been found. This makes the shape (and more importantly height) of the tree a function of the order in which we insert the keys.

Unfortunately, bad things can happen when building trees through insertion. The data structure has no control over the order of insertion. Consider what happens if the user inserts the keys in sorted order. The operations `insert(`a`)`, followed by `insert(`b`)`, `insert(`c`)`, `insert(`d`)`, ... will produce a skinny linear height tree where only right pointers are used.

Thus binary trees can have heights ranging from $\lg n$ to n. But how tall are they on average? The average case analysis of algorithms can be tricky because we must carefully specify what we mean by *average*. The question is well defined if we consider each of the $n!$ possible insertion orderings equally likely and average over those. If so, we are in luck, because with high probability the resulting tree will have $O(\log n)$ height. This will be shown in Section 4.6 (page 123).

This argument is an important example of the power of *randomization*. We can often develop simple algorithms that offer good performance with high probability. We will see that a similar idea underlies the fastest known sorting algorithm, quicksort.

3.4.3 Balanced Search Trees

Random search trees are *usually* good. But if we get unlucky with our order of insertion, we can end up with a linear-height tree in the worst case. This worst case is outside of our direct control, since we must build the tree in response to the requests given by our potentially nasty user.

What would be better is an insertion/deletion procedure which *adjusts* the tree a little after each insertion, keeping it close enough to be balanced so the maximum height is logarithmic. Sophisticated *balanced* binary search tree data structures have been developed that guarantee the height of the tree always to be $O(\log n)$. Therefore, all dictionary operations (insert, delete, query) take $O(\log n)$ time each. Implementations of balanced tree data structures such as red-black trees and splay trees are discussed in Section 12.1 (page 367).

From an algorithm design viewpoint, it is important to know that these trees exist and that they can be used as black boxes to provide an efficient dictionary implementation. When figuring the costs of dictionary operations for algorithm analysis, we can assume the worst-case complexities of balanced binary trees to be a fair measure.

Take-Home Lesson: Picking the wrong data structure for the job can be disastrous in terms of performance. Identifying the very best data structure is usually not as critical, because there can be several choices that perform similarly.

Stop and Think: Exploiting Balanced Search Trees

Problem: You are given the task of reading n numbers and then printing them out in sorted order. Suppose you have access to a balanced dictionary data structure, which supports the operations search, insert, delete, minimum, maximum, successor, and predecessor each in $O(\log n)$ time.

1. How can you sort in $O(n \log n)$ time using only insert and in-order traversal?
2. How can you sort in $O(n \log n)$ time using only minimum, successor, and insert?
3. How can you sort in $O(n \log n)$ time using only minimum, insert, delete?

Solution: The first problem allows us to do insertion and inorder-traversal. We can build a search tree by inserting all n elements, then do a traversal to access the items in sorted order:

```
Sort1()
    initialize-tree(t)
    While (not EOF)
        read(x);
        insert(x,t)
    Traverse(t)
```

```
Sort2()
    initialize-tree(t)
    While (not EOF)
        read(x);
        insert(x,t);
    y = Minimum(t)
    While (y ≠ NULL) do
        print(y → item)
        y = Successor(y,t)
```

```
Sort3()
    initialize-tree(t)
    While (not EOF)
        read(x);
        insert(x,t);
    y = Minimum(t)
    While (y ≠ NULL) do
        print(y→item)
        Delete(y,t)
        y = Minimum(t)
```

The second problem allows us to use the minimum and successor operations after constructing the tree. We can start from the minimum element, and then repeatedly find the successor to traverse the elements in sorted order.

The third problem does not give us successor, but does allow us delete. We can repeatedly find and delete the minimum element to once again traverse all the elements in sorted order.

Each of these algorithms does a linear number of logarithmic-time operations, and hence runs in $O(n \log n)$ time. The key to exploiting balanced binary search trees is using them as black boxes. ∎

3.5 Priority Queues

Many algorithms process items in a specific order. For example, suppose you must schedule jobs according to their importance relative to other jobs. Scheduling the jobs requires sorting them by importance, and then evaluating them in this sorted order.

Priority queues are data structures that provide more flexibility than simple sorting, because they allow new elements to enter a system at arbitrary intervals. It is much more cost-effective to insert a new job into a priority queue than to re-sort everything on each such arrival.

The basic priority queue supports three primary operations:

- *Insert(Q,x)*– Given an item x with key k, insert it into the priority queue Q.
- *Find-Minimum(Q)* or *Find-Maximum(Q)*– Return a pointer to the item whose key value is smaller (larger) than any other key in the priority queue Q.
- *Delete-Minimum(Q)* or *Delete-Maximum(Q)*– Remove the item from the priority queue Q whose key is minimum (maximum).

Many naturally occurring processes are accurately modeled by priority queues. Single people maintain a priority queue of potential dating candidates—mentally if not explicitly. One's impression on meeting a new person maps directly to an attractiveness or desirability score. Desirability serves as the *key* field for inserting this new entry into the "little black book" priority queue data structure. Dating is the process of extracting the most desirable person from the data structure (Find-Maximum), spending an evening to evaluate them better, and then reinserting them into the priority queue with a possibly revised score.

Take-Home Lesson: Building algorithms around data structures such as dictionaries and priority queues leads to both clean structure and good performance.

Stop and Think: Basic Priority Queue Implementations

Problem: What is the worst-case time complexity of the three basic priority queue operations (insert, find-minimum, and delete-minimum) when the basic data structure is

- An unsorted array.
- A sorted array.
- A balanced binary search tree.

Solution: There is surprising subtlety in implementing these three operations, even when using a data structure as simple as an unsorted array. The unsorted array dictionary (discussed on page 73) implemented insertion and deletion in constant time, and search and minimum in linear time. A linear time implementation of delete-minimum can be composed from *find-minimum* followed by *delete*.

For sorted arrays, we can implement insert and delete in linear time, and minimum in constant time. However, all priority queue deletions involve only the minimum element. By storing the sorted array in reverse order (largest value on top), the minimum element will be the last one in the array. Deleting the tail element requires no movement of any items, just decrementing the number of remaining items n, and so delete-minimum can be implemented in constant time.

All this is fine, yet the following table claims we can implement find-minimum in constant time for each data structure:

	Unsorted array	Sorted array	Balanced tree
Insert(Q, x)	$O(1)$	$O(n)$	$O(\log n)$
Find-Minimum(Q)	$O(1)$	$O(1)$	$O(1)$
Delete-Minimum(Q)	$O(n)$	$O(1)$	$O(\log n)$

The trick is using an extra variable to store a pointer/index to the minimum entry in each of these structures, so we can simply return this value whenever we are asked to find-minimum. Updating this pointer on each insertion is easy—we update it if and only if the newly inserted value is less than the current minimum. But what happens on a delete-minimum? We can delete the minimum entry *have*, then do an honest find-minimum to restore our canned value. The honest find-minimum takes linear time on an unsorted array and logarithmic time on a tree, and hence can be folded into the cost of each deletion. ∎

Priority queues are very useful data structures. Indeed, they will be the hero of two of our war stories, including the next one. A particularly nice priority queue implementation (the heap) will be discussed in the context of sorting in Section 4.3 (page 108). Further, a complete set of priority queue implementations is presented in Section 12.2 (page 373) of the catalog.

3.6 War Story: Stripping Triangulations

Geometric models used in computer graphics are commonly represented as a triangulated surface, as shown in Figure 3.5(l). High-performance rendering engines have special hardware for rendering and shading triangles. This hardware is so fast that the bottleneck of rendering is the cost of feeding the triangulation structure into the hardware engine.

Although each triangle can be described by specifying its three endpoints, an alternative representation is more efficient. Instead of specifying each triangle in isolation, suppose that we partition the triangles into *strips* of adjacent triangles

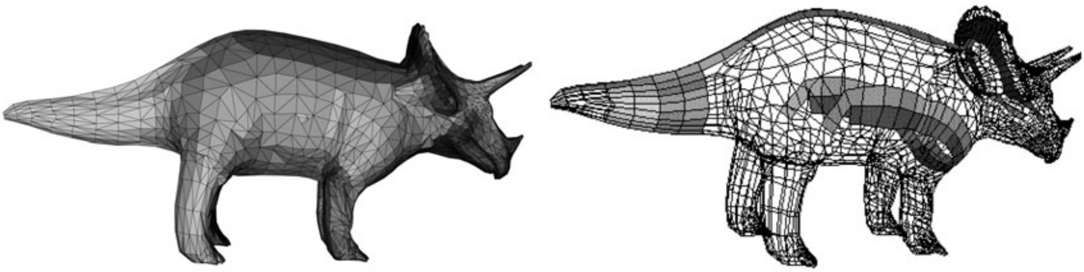

Figure 3.5: (l) A triangulated model of a dinosaur (r) Several triangle strips in the model

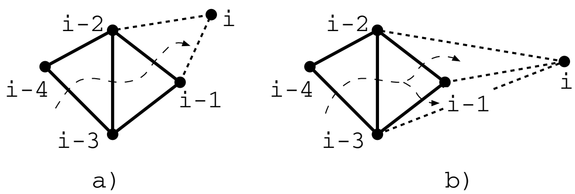

Figure 3.6: Partitioning a triangular mesh into strips: (a) with left-right turns (b) with the flexibility of arbitrary turns

and walk along the strip. Since each triangle shares two vertices in common with its neighbors, we save the cost of retransmitting the two extra vertices and any associated information. To make the description of the triangles unambiguous, the *OpenGL* triangular-mesh renderer assumes that all turns alternate from left to right (as shown in Figure 3.6).

The task of finding a small number of strips that cover each triangle in a mesh can be thought of as a graph problem. The graph of interest has a vertex for every *triangle* of the mesh, and an edge between every pair of vertices representing adjacent triangles. This *dual graph* representation captures all the information about the triangulation (see Section 15.12 (page 520)) needed to partition it into triangular strips.

Once we had the dual graph available, the project could begin in earnest. We sought to partition the vertices into as few paths or strips as possible. Partitioning it into one path implied that we had discovered a Hamiltonian path, which by definition visits each vertex exactly once. Since finding a Hamiltonian path is NP-complete (see Section 16.5 (page 538)), we knew not to look for an optimal algorithm, but concentrate instead on heuristics.

The simplest heuristic for strip cover would start from an arbitrary triangle and then do a left-right walk until the walk ends, either by hitting the boundary of

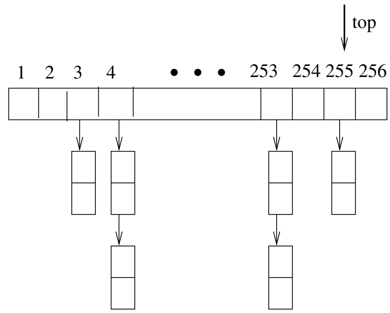

Figure 3.7: A bounded height priority queue for triangle strips

the object or a previously visited triangle. This heuristic had the advantage that it would be fast and simple, although there is no reason why it should find the smallest possible set of left-right strips for a given triangulation.

The *greedy* heuristic would be more likely to result in a small number of strips however. Greedy heuristics always try to grab the best possible thing first. In the case of the triangulation, the natural greedy heuristic would identify the starting triangle that yields the longest left-right strip, and peel that one off first.

Being greedy does not guarantee you the best possible solution either, since the first strip you peel off might break apart a lot of potential strips we might have wanted to use later. Still, being greedy is a good rule of thumb if you want to get rich. Since removing the longest strip would leave the fewest number of triangles for later strips, the greedy heuristic should outperform the naive heuristic.

But how much time does it take to find the largest strip to peel off next? Let k be the length of the walk possible from an average vertex. Using the simplest possible implementation, we could walk from each of the n vertices to find the largest remaining strip to report in $O(kn)$ time. Repeating this for each of the roughly n/k strips we extract yields an $O(n^2)$-time implementation, which would be hopelessly slow on a typical model of 20,000 triangles.

How could we speed this up? It seems wasteful to rewalk from each triangle after deleting a single strip. We could maintain the lengths of all the possible future strips in a data structure. However, whenever we peel off a strip, we must update the lengths of all affected strips. These strips will be shortened because they walked through a triangle that now no longer exists. There are two aspects of such a data structure:

- *Priority Queue* – Since we were repeatedly identifying the longest remaining strip, we needed a priority queue to store the strips ordered according to length. The next strip to peel always sat at the top of the queue. Our priority queue had to permit reducing the priority of arbitrary elements of the queue whenever we updated the strip lengths to reflect what triangles were peeled

Model name	Triangle count	Naive cost	Greedy cost	Greedy time
Diver	3,798	8,460	4,650	6.4 sec
Heads	4,157	10,588	4,749	9.9 sec
Framework	5,602	9,274	7,210	9.7 sec
Bart Simpson	9,654	24,934	11,676	20.5 sec
Enterprise	12,710	29,016	13,738	26.2 sec
Torus	20,000	40,000	20,200	272.7 sec
Jaw	75,842	104,203	95,020	136.2 sec

Figure 3.8: A comparison of the naive versus greedy heuristics for several triangular meshes

away. Because all of the strip lengths were bounded by a fairly small integer (hardware constraints prevent any strip from having more than 256 vertices), we used a bounded-height priority queue (an array of buckets shown in Figure 3.7 and described in Section 12.2 (page 373)). An ordinary heap would also have worked just fine.

To update the queue entry associated with each triangle, we needed to quickly find where it was. This meant that we also needed a ...

- *Dictionary* – For each triangle in the mesh, we needed to find where it was in the queue. This meant storing a pointer to each triangle in a dictionary. By integrating this dictionary with the priority queue, we built a data structure capable of a wide range of operations.

Although there were various other complications, such as quickly recalculating the length of the strips affected by the peeling, the key idea needed to obtain better performance was to use the priority queue. Run time improved by several orders of magnitude after employing this data structure.

How much better did the greedy heuristic do than the naive heuristic? Consider the table in Figure 3.8. In all cases, the greedy heuristic led to a set of strips that cost less, as measured by the total size of the strips. The savings ranged from about 10% to 50%, which is quite remarkable since the greatest possible improvement (going from three vertices per triangle down to one) yields a savings of only 66.6%.

After implementing the greedy heuristic with our priority queue data structure, the program ran in $O(n \cdot k)$ time, where n is the number of triangles and k is the length of the average strip. Thus the torus, which consisted of a small number of very long strips, took longer than the jaw, even though the latter contained over three times as many triangles.

There are several lessons to be gleaned from this story. First, when working with a large enough data set, only linear or near linear algorithms (say $O(n \log n)$) are likely to be fast enough. Second, choosing the right data structure is often the key to getting the time complexity down to this point. Finally, using smart heuristic like greedy is likely to significantly improve quality over the naive approach. How much the improvement will be can only be determined by experimentation.

3.7 Hashing and Strings

Hash tables are a *very* practical way to maintain a dictionary. They exploit the fact that looking an item up in an array takes constant time once you have its index. A hash function is a mathematical function that maps keys to integers. We will use the value of our hash function as an index into an array, and store our item at that position.

The first step of the hash function is usually to map each key to a big integer. Let α be the size of the alphabet on which a given string S is written. Let `char(c)` be a function that maps each symbol of the alphabet to a unique integer from 0 to $\alpha - 1$. The function

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

maps each string to a unique (but large) integer by treating the characters of the string as "digits" in a base-α number system.

The result is unique identifier numbers, but they are so large they will quickly exceed the number of slots in our hash table (denoted by m). We must reduce this number to an integer between 0 and $m-1$, by taking the remainder of $H(S)$ mod m. This works on the same principle as a roulette wheel. The ball travels a long distance around and around the circumference-m wheel $\lfloor H(S)/m \rfloor$ times before settling down to a random bin. If the table size is selected with enough finesse (ideally m is a large prime not too close to $2^i - 1$), the resulting hash values should be fairly uniformly distributed.

3.7.1 Collision Resolution

No matter how good our hash function is, we had better be prepared for collisions, because two distinct keys will occasionally hash to the same value. *Chaining* is the easiest approach to collision resolution. Represent the hash table as an array of m linked lists, as shown in Figure 3.9. The ith list will contain all the items that hash to the value of i. Thus search, insertion, and deletion reduce to the corresponding problem in linked lists. If the n keys are distributed uniformly in a table, each list will contain roughly n/m elements, making them a constant size when $m \approx n$.

Chaining is very natural, but devotes a considerable amount of memory to pointers. This is space that could be used to make the table larger, and hence the "lists" smaller.

The alternative is something called *open addressing*. The hash table is maintained as an array of elements (not buckets), each initialized to null, as shown in Figure 3.10. On an insertion, we check to see if the desired position is empty. If so,

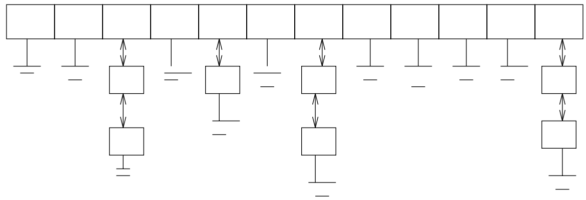

Figure 3.9: Collision resolution by chaining

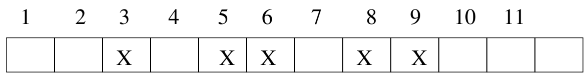

Figure 3.10: Collision resolution by open addressing

we insert it. If not, we must find some other place to insert it instead. The simplest possibility (called *sequential probing*) inserts the item in the next open spot in the table. If the table is not too full, the contiguous runs of items should be fairly small, hence this location *should* be only a few slots from its intended position.

Searching for a given key now involves going to the appropriate hash value and checking to see if the item there is the one we want. If so, return it. Otherwise we must keep checking through the length of the run.

Deletion in an open addressing scheme can get ugly, since removing one element might break a chain of insertions, making some elements inaccessible. We have no alternative but to reinsert all the items in the run following the new hole.

Chaining and open addressing both require $O(m)$ to initialize an m-element hash table to null elements prior to the first insertion. Traversing all the elements in the table takes $O(n + m)$ time for chaining, because we have to scan all m buckets looking for elements, even if the actual number of inserted items is small. This reduces to $O(m)$ time for open addressing, since n must be at most m.

When using chaining with doubly-linked lists to resolve collisions in an m-element hash table, the dictionary operations for n items can be implemented in the following expected and worst case times:

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n+m)$	$O(n+m)$
Predecessor(L, x)	$O(n+m)$	$O(n+m)$
Minimum(L)	$O(n+m)$	$O(n+m)$
Maximum(L)	$O(n+m)$	$O(n+m)$

Pragmatically, a hash table is often the best data structure to maintain a dictionary. The applications of hashing go far beyond dictionaries, however, as we will see below.

3.7.2 Efficient String Matching via Hashing

Strings are sequences of characters where the order of the characters matters, since *ALGORITHM* is different than *LOGARITHM*. Text strings are fundamental to a host of computing applications, from programming language parsing/compilation, to web search engines, to biological sequence analysis.

The primary data structure for representing strings is an array of characters. This allows us constant-time access to the ith character of the string. Some auxiliary information must be maintained to mark the end of the string—either a special end-of-string character or (perhaps more usefully) a count of the n characters in the string.

The most fundamental operation on text strings is substring search, namely:

Problem: Substring Pattern Matching
Input: A text string t and a pattern string p.
Output: Does t contain the pattern p as a substring, and if so where?

The simplest algorithm to search for the presence of pattern string p in text t overlays the pattern string at every position in the text, and checks whether every pattern character matches the corresponding text character. As demonstrated in Section 2.5.3 (page 43), this runs in $O(nm)$ time, where $n = |t|$ and $m = |p|$.

This quadratic bound is worst-case. More complicated, worst-case linear-time search algorithms do exist: see Section 18.3 (page 628) for a complete discussion. But here we give a linear *expected-time* algorithm for string matching, called the Rabin-Karp algorithm. It is based on hashing. Suppose we compute a given hash function on both the pattern string p and the m-character substring starting from the ith position of t. If these two strings are identical, clearly the resulting hash values must be the same. If the two strings are different, the hash values will *almost certainly* be different. These false positives should be so rare that we can easily spend the $O(m)$ time it takes to explicitly check the identity of two strings whenever the hash values agree.

This reduces string matching to $n-m+2$ hash value computations (the $n-m+1$ windows of t, plus one hash of p), plus what *should be* a very small number of $O(m)$ time verification steps. The catch is that it takes $O(m)$ time to compute a hash function on an m-character string, and $O(n)$ such computations seems to leave us with an $O(mn)$ algorithm again.

But let's look more closely at our previously defined hash function, applied to the m characters starting from the jth position of string S:

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

What changes if we now try to compute $H(S, j+1)$—the hash of the next window of m characters? Note that $m-1$ characters are the same in both windows, although this differs by one in the number of times they are multiplied by α. A little algebra reveals that

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

This means that once we know the hash value from the j position, we can find the hash value from the $(j+1)$st position for the cost of two multiplications, one addition, and one subtraction. This can be done in constant time (the value of α^{m-1} can be computed once and used for all hash value computations). This math works even if we compute $H(S, j) \bmod M$, where M is a reasonably large prime number, thus keeping the size of our hash values small (at most M) even when the pattern string is long.

Rabin-Karp is a good example of a randomized algorithm (if we pick M in some random way). We get no guarantee the algorithm runs in $O(n+m)$ time, because we may get unlucky and have the hash values regularly collide with spurious matches. Still, the odds are heavily in our favor—if the hash function returns values uniformly from 0 to $M-1$, the probability of a false collision should be $1/M$. This is quite reasonable: if $M \approx n$, there should only be one false collision per string, and if $M \approx n^k$ for $k \geq 2$, the odds are great we will never see any false collisions.

3.7.3 Duplicate Detection Via Hashing

The key idea of hashing is to represent a large object (be it a key, a string, or a substring) using a single number. The goal is a representation of the large object by an entity that can be manipulated in constant time, such that it is relatively unlikely that two different large objects map to the same value.

Hashing has a variety of clever applications beyond just speeding up search. I once heard Udi Manber—then Chief Scientist at Yahoo—talk about the algorithms employed at his company. The three most important algorithms at Yahoo, he said, were hashing, hashing, and hashing.

Consider the following problems with nice hashing solutions:

- *Is a given document different from all the rest in a large corpus?* – A search engine with a huge database of n documents spiders yet another webpage. How can it tell whether this adds something new to add to the database, or is just a duplicate page that exists elsewhere on the Web?

  Explicitly comparing the new document D to all n documents is hopelessly inefficient for a large corpus. But we can hash D to an integer, and compare it to the hash codes of the rest of the corpus. Only when there is a collision is D a possible duplicate. Since we expect few spurious collisions, we can explicitly compare the few documents sharing the exact hash code with little effort.

- *Is part of this document plagiarized from a document in a large corpus?* – A lazy student copies a portion of a Web document into their term paper. “The Web is a big place,” he smirks. “How will anyone ever find which one?”

  This is a more difficult problem than the previous application. Adding, deleting, or changing even one character from a document will completely change its hash code. Thus the hash codes produced in the previous application cannot help for this more general problem.

  However, we *could* build a hash table of all overlapping windows (substrings) of length w in all the documents in the corpus. Whenever there is a match of hash codes, there is likely a common substring of length w between the two documents, which can then be further investigated. We should choose w to be long enough so such a co-occurrence is very unlikely to happen by chance.

  The biggest downside of this scheme is that the size of the hash table becomes as large as the documents themselves. Retaining a small but well-chosen subset of these hash codes (say those which are exact multiples of 100) for each document leaves us likely to detect sufficiently long duplicate strings.

- *How can I convince you that a file isn't changed?* – In a closed-bid auction, each party submits their bid in secret before the announced deadline. If you knew what the other parties were bidding, you could arrange to bid \$1 more than the highest opponent and walk off with the prize as cheaply as possible. Thus the “right” auction strategy is to hack into the computer containing the bids just prior to the deadline, read the bids, and then magically emerge the winner.

  How can this be prevented? What if everyone submits a hash code of their actual bid prior to the deadline, and then submits the full bid after the deadline? The auctioneer will pick the largest full bid, but checks to make sure the hash code matches that submitted prior to the deadline. Such *cryptographic hashing* methods provide a way to ensure that the file you give me today is the same as original, because any changes to the file will result in changing the hash code.

Although the worst-case bounds on anything involving hashing are dismal, with a proper hash function we can confidently expect good behavior. Hashing is a fundamental idea in randomized algorithms, yielding linear expected-time algorithms for problems otherwise $\Theta(n \log n)$, or $\Theta(n^2)$ in the worst case.

3.8 Specialized Data Structures

The basic data structures described thus far all represent an unstructured set of items so as to facilitate retrieval operations. These data structures are well known to most programmers. Not as well known are data structures for representing more structured or specialized kinds of objects, such as points in space, strings, and graphs.

The design principles of these data structures are the same as for basic objects. There exists a set of basic operations we need to perform repeatedly. We seek a data structure that supports these operations very efficiently. These efficient, specialized data structures are important for efficient graph and geometric algorithms so one should be aware of their existence. Details appear throughout the catalog.

- *String data structures* – Character strings are typically represented by arrays of characters, perhaps with a special character to mark the end of the string. Suffix trees/arrays are special data structures that preprocess strings to make pattern matching operations faster. See Section 12.3 (page 377) for details.

- *Geometric data structures* – Geometric data typically consists of collections of data points and regions. Regions in the plane can be described by polygons, where the boundary of the polygon is given by a chain of line segments. Polygons can be represented using an array of points $(v_1, \ldots, v_n, v_1)$, such that (v_i, v_{i+1}) is a segment of the boundary. Spatial data structures such as kd-trees organize points and regions by geometric location to support fast search. For more details, see Section 12.6 (page 389).

- *Graph data structures* – Graphs are typically represented using either adjacency matrices or adjacency lists. The choice of representation can have a substantial impact on the design of the resulting graph algorithms, as discussed in Chapter 6 and in the catalog in Section 12.4.

- *Set data structures* – Subsets of items are typically represented using a dictionary to support fast membership queries. Alternately, *bit vectors* are boolean arrays such that the ith bit represents true if i is in the subset. Data structures for manipulating sets is presented in the catalog in Section 12.5. The union-find data structure for maintaining set partitions will be covered in Section 6.1.3 (page 198).

3.9 War Story: String 'em Up

The human genome encodes all the information necessary to build a person. This project has already had an enormous impact on medicine and molecular biology. Algorists have become interested in the human genome project as well, for several reasons:

- DNA sequences can be accurately represented as strings of characters on the four-letter alphabet (A,C,T,G). Biologist's needs have sparked new interest in old algorithmic problems such as string matching (see Section 18.3 (page 628)) as well as creating new problems such as shortest common superstring (see Section 18.9 (page 654)).

```
            T A T C C
          T T A T C
        G T T A T
      C G T T A
    A C G T T A T C C A
```

Figure 3.11: The concatenation of two fragments can be in S only if all sub-fragments are

- DNA sequences are very *long* strings. The human genome is approximately three billion base pairs (or characters) long. Such large problem size means that asymptotic (Big-Oh) complexity analysis is usually fully justified on biological problems.

- Enough money is being invested in genomics for computer scientists to want to claim their piece of the action.

One of my interests in computational biology revolved around a proposed technique for DNA sequencing called sequencing by hybridization (SBH). This procedure attaches a set of probes to an array, forming a *sequencing chip*. Each of these probes determines whether or not the probe string occurs as a substring of the DNA target. The target DNA can now be sequenced based on the constraints of which strings are (and are not) substrings of the target.

We sought to identify all the strings of length $2k$ that are possible substrings of an unknown string S, given the set of all length k substrings of S. For example, suppose we know that AC, CA, and CC are the only length-2 substrings of S. It is possible that $ACCA$ is a substring of S, since the center substring is one of our possibilities. However, $CAAC$ *cannot* be a substring of S, since AA is not a substring of S. We needed to find a fast algorithm to construct all the consistent length-$2k$ strings, since S could be very long.

The simplest algorithm to build the $2k$ strings would be to concatenate all $O(n^2)$ pairs of k-strings together, and then test to make sure that all $(k-1)$ length-k substrings spanning the boundary of the concatenation were in fact substrings, as shown in Figure 3.11. For example, the nine possible concatenations of AC, CA, and CC are $ACAC$, $ACCA$, $ACCC$, $CAAC$, $CACA$, $CACC$, $CCAC$, $CCCA$, and $CCCC$. Only $CAAC$ can be eliminated because of the absence of AA.

We needed a fast way of testing whether the $k-1$ substrings straddling the concatenation were members of our dictionary of permissible k-strings. The time it takes to do this depends upon which dictionary data structure we use. A binary search tree could find the correct string within $O(\log n)$ comparisons, where each comparison involved testing which of two length-k strings appeared first in alphabetical order. The total time using such a binary search tree would be $O(k \log n)$.

That seemed pretty good. So my graduate student, Dimitris Margaritis, used a binary search tree data structure for our implementation. It worked great up until the moment we ran it.

"I've tried the fastest computer in our department, but our program is too slow," Dimitris complained. "It takes forever on string lengths of only 2,000 characters. We will never get up to 50,000."

We profiled our program and discovered that almost all the time was spent searching in this data structure. This was no surprise since we did this $k-1$ times for each of the $O(n^2)$ possible concatenations. We needed a faster dictionary data structure, since search was the innermost operation in such a deep loop.

"How about using a hash table?" I suggested. "It should take $O(k)$ time to hash a k-character string and look it up in our table. That should knock off a factor of $O(\log n)$, which will mean something when $n \approx 2{,}000$."

Dimitris went back and implemented a hash table implementation for our dictionary. Again, it worked great up until the moment we ran it.

"Our program is still too slow," Dimitris complained. "Sure, it is now about ten times faster on strings of length 2,000. So now we can get up to about 4,000 characters. Big deal. We will never get up to 50,000."

"We should have expected this," I mused. "After all, $\lg_2(2,000) \approx 11$. We need a faster data structure to search in our dictionary of strings."

"But what can be faster than a hash table?" Dimitris countered. "To look up a k-character string, you must read all k characters. Our hash table already does $O(k)$ searching."

"Sure, it takes k comparisons to test the first substring. But maybe we can do better on the second test. Remember where our dictionary queries are coming from. When we concatenate $ABCD$ with $EFGH$, we are first testing whether $BCDE$ is in the dictionary, then $CDEF$. These strings differ from each other by only one character. We should be able to exploit this so each subsequent test takes constant time to perform…."

"We can't do that with a hash table," Dimitris observed. "The second key is not going to be anywhere near the first in the table. A binary search tree won't help, either. Since the keys $ABCD$ and $BCDE$ differ according to the first character, the two strings will be in different parts of the tree."

"But we can use a suffix tree to do this," I countered. "A suffix tree is a trie containing all the suffixes of a given set of strings. For example, the suffixes of $ACAC$ are $\{ACAC, CAC, AC, C\}$. Coupled with suffixes of string $CACT$, we get the suffix tree of Figure 3.12. By following a pointer from $ACAC$ to its longest proper suffix CAC, we get to the right place to test whether $CACT$ is in our set of strings. One character comparison is all we need to do from there."

Suffix trees are amazing data structures, discussed in considerably more detail in Section 12.3 (page 377). Dimitris did some reading about them, then built a nice

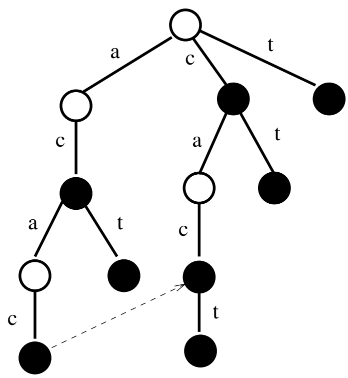

Figure 3.12: Suffix tree on $ACAC$ and $CACT$, with the pointer to the suffix of $ACAC$

suffix tree implementation for our dictionary. Once again, it worked great up until the moment we ran it.

"Now our program is faster, but it runs out of memory," Dimitris complained. "The suffix tree builds a path of length k for each suffix of length k, so all told there can be $\Theta(n^2)$ nodes in the tree. It crashes when we go beyond 2,000 characters. We will never get up to strings with 50,000 characters."

I wasn't ready to give up yet. "There is a way around the space problem, by using compressed suffix trees," I recalled. "Instead of explicitly representing long paths of character nodes, we can refer back to the original string." Compressed suffix trees always take linear space, as described in Section 12.3 (page 377).

Dimitris went back one last time and implemented the compressed suffix tree data structure. *Now* it worked great! As shown in Figure 3.13, we ran our simulation for strings of length $n = 65{,}536$ without incident. Our results showed that interactive SBH could be a very efficient sequencing technique. Based on these simulations, we were able to arouse interest in our technique from biologists. Making the actual wet laboratory experiments feasible provided another computational challenge, which is reported in Section 7.7 (page 263).

The take-home lessons for programmers from Figure 3.13 should be apparent. We isolated a single operation (dictionary string search) that was being performed repeatedly and optimized the data structure we used to support it. We started with a simple implementation (binary search trees) in the hopes that it would suffice, and then used profiling to reveal the trouble when it didn't. When an improved dictionary structure still did not suffice, we looked deeper into the kind of queries we were performing, so that we could identify an even better data structure. Finally, we didn't give up until we had achieved the level of performance we needed. In algorithms, as in life, persistence usually pays off.

String length	Binary tree	Hash table	Suffix tree	Compressed tree
8	0.0	0.0	0.0	0.0
16	0.0	0.0	0.0	0.0
32	0.1	0.0	0.0	0.0
64	0.3	0.4	0.3	0.0
128	2.4	1.1	0.5	0.0
256	17.1	9.4	3.8	0.2
512	31.6	67.0	6.9	1.3
1,024	1,828.9	96.6	31.5	2.7
2,048	11,441.7	941.7	553.6	39.0
4,096	> 2 days	5,246.7	out of	45.4
8,192		> 2 days	memory	642.0
16,384				1,614.0
32,768				13,657.8
65,536				39,776.9

Figure 3.13: Run times (in seconds) for the SBH simulation using various data structures

Chapter Notes

Optimizing hash table performance is surprisingly complicated for such a conceptually simple data structure. The importance of short runs in open addressing has led to more sophisticated schemes than sequential probing for optimal hash table performance. For more details, see Knuth [Knu98].

Our triangle strip optimizing program, *stripe*, is described in [ESV96]. Hashing techniques for plagiarism detection are discussed in [SWA03].

Surveys of algorithmic issues in DNA sequencing by hybridization include [CK94, PL94]. Our work on interactive SBH reported in the war story is reported in [MS95a].

3.10 Exercises

Stacks, Queues, and Lists

3-1. *[3]* A common problem for compilers and text editors is determining whether the parentheses in a string are balanced and properly nested. For example, the string ((())())() contains properly nested pairs of parentheses, which the strings)()(and ()) do not. Give an algorithm that returns true if a string contains properly nested and balanced parentheses, and false if otherwise. For full credit, identify the position of the first offending parenthesis if the string is not properly nested and balanced.

3-2. *[3]* Write a program to reverse the direction of a given singly-linked list. In other words, after the reversal all pointers should now point backwards. Your algorithm should take linear time.

3-3. *[5]* We have seen how dynamic arrays enable arrays to grow while still achieving constant-time amortized performance. This problem concerns extending dynamic arrays to let them both grow and shrink on demand.

(a) Consider an underflow strategy that cuts the array size in half whenever the array falls below half full. Give an example sequence of insertions and deletions where this strategy gives a bad amortized cost.

(b) Then, give a better underflow strategy than that suggested above, one that achieves constant amortized cost per deletion.

Trees and Other Dictionary Structures

3-4. *[3]* Design a dictionary data structure in which search, insertion, and deletion can all be processed in $O(1)$ time in the worst case. You may assume the set elements are integers drawn from a finite set $1, 2, .., n$, and initialization can take $O(n)$ time.

3-5. *[3]* Find the overhead fraction (the ratio of data space over total space) for each of the following binary tree implementations on n nodes:

(a) All nodes store data, two child pointers, and a parent pointer. The data field requires four bytes and each pointer requires four bytes.

(b) Only leaf nodes store data; internal nodes store two child pointers. The data field requires four bytes and each pointer requires two bytes.

3-6. *[5]* Describe how to modify any balanced tree data structure such that search, insert, delete, minimum, and maximum still take $O(\log n)$ time each, but successor and predecessor now take $O(1)$ time each. Which operations have to be modified to support this?

3-7. *[5]* Suppose you have access to a balanced dictionary data structure, which supports each of the operations search, insert, delete, minimum, maximum, successor, and predecessor in $O(\log n)$ time. Explain how to modify the insert and delete operations so they still take $O(\log n)$ but now minimum and maximum take $O(1)$ time. (Hint: think in terms of using the abstract dictionary operations, instead of mucking about with pointers and the like.)

3-8. *[6]* Design a data structure to support the following operations:

- *insert(x,T)* – Insert item x into the set T.
- *delete(k,T)* – Delete the kth smallest element from T.
- *member(x,T)* – Return true iff $x \in T$.

All operations must take $O(\log n)$ time on an n-element set.

3-9. *[8]* A *concatenate* operation takes two sets S_1 and S_2, where every key in S_1 is smaller than any key in S_2, and merges them together. Give an algorithm to concatenate two binary search trees into one binary search tree. The worst-case running time should be $O(h)$, where h is the maximal height of the two trees.

Applications of Tree Structures

3-10. *[5]* In the *bin-packing problem*, we are given n metal objects, each weighing between zero and one kilogram. Our goal is to find the smallest number of bins that will hold the n objects, with each bin holding one kilogram at most.

- The *best-fit heuristic* for bin packing is as follows. Consider the objects in the order in which they are given. For each object, place it into the partially filled bin with the smallest amount of extra room *after* the object is inserted.. If no such bin exists, start a new bin. Design an algorithm that implements the best-fit heuristic (taking as input the n weights $w_1, w_2, ..., w_n$ and outputting the number of bins used) in $O(n \log n)$ time.
- Repeat the above using the *worst-fit heuristic*, where we put the next object in the partially filled bin with the largest amount of extra room *after* the object is inserted.

3-11. *[5]* Suppose that we are given a sequence of n values $x_1, x_2, ..., x_n$ and seek to quickly answer repeated queries of the form: given i and j, find the smallest value in $x_i, \ldots, x_j$.

(a) Design a data structure that uses $O(n^2)$ space and answers queries in $O(1)$ time.

(b) Design a data structure that uses $O(n)$ space and answers queries in $O(\log n)$ time. For partial credit, your data structure can use $O(n \log n)$ space and have $O(\log n)$ query time.

3-12. *[5]* Suppose you are given an input set S of n numbers, and a black box that if given any sequence of real numbers and an integer k instantly and correctly answers whether there is a subset of input sequence whose sum is exactly k. Show how to use the black box $O(n)$ times to find a subset of S that adds up to k.

3-13. *[5]* Let $A[1..n]$ be an array of real numbers. Design an algorithm to perform any sequence of the following operations:

- *Add(i,y)* – Add the value y to the ith number.
- *Partial-sum(i)* – Return the sum of the first i numbers, i.e. $\sum_{j=1}^{i} A[j]$.

There are no insertions or deletions; the only change is to the values of the numbers. Each operation should take $O(\log n)$ steps. You may use one additional array of size n as a work space.

3-14. *[8]* Extend the data structure of the previous problem to support insertions and deletions. Each element now has both a *key* and a *value*. An element is accessed by its key. The addition operation is applied to the values, but the elements are accessed by its key. The *Partial_sum* operation is different.

- *Add(k,y)* – Add the value y to the item with key k.
- *Insert(k,y)* – Insert a new item with key k and value y.
- *Delete(k)* – Delete the item with key k.

- *Partial-sum(k)* – Return the sum of all the elements currently in the set whose key is less than k, i.e. $\sum_{x_j<y} x_i$.

The worst case running time should still be $O(n \log n)$ for any sequence of $O(n)$ operations.

3-15. *[8]* Design a data structure that allows one to search, insert, and delete an integer X in $O(1)$ time (i.e., constant time, independent of the total number of integers stored). Assume that $1 \leq X \leq n$ and that there are $m+n$ units of space available, where m is the maximum number of integers that can be in the table at any one time. (Hint: use two arrays $A[1..n]$ and $B[1..m]$.) You are not allowed to initialize either A or B, as that would take $O(m)$ or $O(n)$ operations. This means the arrays are full of random garbage to begin with, so you must be very careful.

Implementation Projects

3-16. *[5]* Implement versions of several different dictionary data structures, such as linked lists, binary trees, balanced binary search trees, and hash tables. Conduct experiments to assess the relative performance of these data structures in a simple application that reads a large text file and reports exactly one instance of each word that appears within it. This application can be efficiently implemented by maintaining a dictionary of all distinct words that have appeared thus far in the text and inserting/reporting each word that is not found. Write a brief report with your conclusions.

3-17. *[5]* A Caesar shift (see Section 18.6 (page 641)) is a very simple class of ciphers for secret messages. Unfortunately, they can be broken using statistical properties of English. Develop a program capable of decrypting Caesar shifts of sufficiently long texts.

Interview Problems

3-18. *[3]* What method would you use to look up a word in a dictionary?

3-19. *[3]* Imagine you have a closet full of shirts. What can you do to organize your shirts for easy retrieval?

3-20. *[4]* Write a function to find the middle node of a singly-linked list.

3-21. *[4]* Write a function to compare whether two binary trees are identical. Identical trees have the same key value at each position and the same structure.

3-22. *[4]* Write a program to convert a binary search tree into a linked list.

3-23. *[4]* Implement an algorithm to reverse a linked list. Now do it without recursion.

3-24. *[5]* What is the best data structure for maintaining URLs that have been visited by a Web crawler? Give an algorithm to test whether a given URL has already been visited, optimizing both space and time.

3-25. *[4]* You are given a search string and a magazine. You seek to generate all the characters in search string by cutting them out from the magazine. Give an algorithm to efficiently determine whether the magazine contains all the letters in the search string.

3-26. *[4]* Reverse the words in a sentence—i.e., "My name is Chris" becomes "Chris is name My." Optimize for time and space.

3-27. *[5]* Determine whether a linked list contains a loop as quickly as possible without using any extra storage. Also, identify the location of the loop.

3-28. *[5]* You have an unordered array X of n integers. Find the array M containing n elements where M_i is the product of all integers in X except for X_i. You may not use division. You can use extra memory. (Hint: There are solutions faster than $O(n^2)$.)

3-29. *[6]* Give an algorithm for finding an ordered word pair (e.g., "New York") occurring with the greatest frequency in a given webpage. Which data structures would you use? Optimize both time and space.

Programming Challenges

These programming challenge problems with robot judging are available at *http://www.programming-challenges.com* or *http://online-judge.uva.es*.

3-1. "Jolly Jumpers" – Programming Challenges 110201, UVA Judge 10038.

3-2. "Crypt Kicker" – Programming Challenges 110204, UVA Judge 843.

3-3. "Where's Waldorf?" – Programming Challenges 110302, UVA Judge 10010.

3-4. "Crypt Kicker II" – Programming Challenges 110304, UVA Judge 850.