  - *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,k)* or *Successor(D,k)* – Retrieve the item from D whose key is immediately before (or after) k 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) reveal 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. ∎

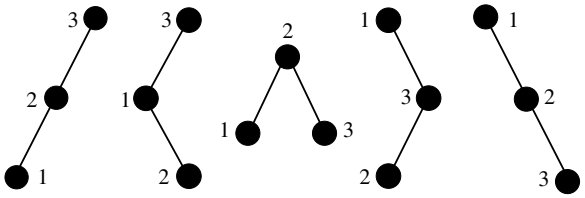

Figure 3.2: The five distinct binary search trees on three nodes

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: