3 Abstract Data Types

Abstract data types (ADTs) provide both a language for talking about and tools for operating on complex data structures. Each is defined by the types of objects that it can store and the operations that can be performed. Unlike a function that takes an input and produces an output, an ADT is more dynamic, periodically receiving information and commands to which it must react in a way that reflects its history. In an object-oriented language, these are implemented with objects, each of which has its own internal variables and operations. A user of an ADT has no access to its internal structure except through the operations provided. This is referred to as *information hiding* and provides a clean boundary between the user and the ADT. One person can use the ADT to develop other algorithms without being concerned with how it is implemented or worrying about accidentally messing up the data structure. Another can implement and modify the ADT without knowing how it is used or worrying about unexpected effects on the rest of the code. A general purpose ADT—not just the code, but also the understanding and the mathematical theory—can be reused in many applications. Having a limited set of operations guides the implementer to use techniques that are efficient for these operations yet may be slow for the operations excluded. Conversely, using an ADT such as a stack in your algorithm automatically tells someone attempting to understand your algorithm a great deal about the purpose of this data structure. Generally, the running time of an operation is not a part of the description of an ADT, but is tied to a particular implementation. However, it is useful for the user to know the relative expense of using operations so that he can make his own choices about which ADTs and which operations to use.

This chapter will treat the following ADTs: lists, stacks, queues, priority queues, graphs, trees, and sets. From the user's perspective, these consist of a data structure and a set of operations with which to access the data. From the perspective of the data structure itself, it is a ongoing system that continues to receive a stream of commands to which it must react dynamically. ADTs have a set of invariants or integrity constraints (both public and hidden) that must be true every time the system is entered or left. Imagining a big loop around the system allows us to regard them as a kind of loop invariant.

3.1 Specifications and Hints at Implementations

The following are examples frequently used ADTs.

Simple Types: Integers, floating point numbers, strings, arrays, and records are abstract data types provided by all programming languages.

The List ADT:

Specification: A list consists of an ordered sequence of elements. Unlike arrays, they contain no empty positions. Elements can be *inserted*, *deleted*, *read*, *modified*, and *searched* for.

Array Implementations: There are different implementations that have tradeoffs in the running time, memory requirements, and difficulty of implementing. The obvious implementation of a list is to put the elements in an array. If the elements are packed one after the other, then the ith element can be accessed in $\Theta(1)$ time, but inserting or deleting an element requires $\Theta(n)$ time because all the elements need to be shifted. Alternatively, blank spaces could be left between the elements. This leaves room to insert or delete elements in $\Theta(1)$ time, but finding the ith element might now take $\Theta(n)$ time.

Linked List Implementations: A problem with the array implementation is that the array needs to be allocated some fixed size of memory when initialized. An alternative implementation, which can be expanded or shrunk in size as needed, uses a linked list. This implementation has the disadvantage of requiring $\Theta(n)$ time to access a particular element. See Section 3.2.

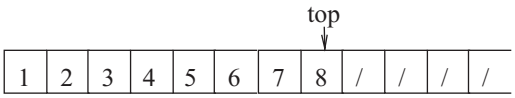
Tree Implementations: A nice balance between the advantages of array and the linked list implementations is data structure called a *heap*. Heaps can do every operation in $\Theta(\log n)$ time. See Section 10.4. Adelson-Velsky–Landis (AVL) trees and red–black trees have similar properties.

The Stack ADT:

Specification: A stack ADT is the same as a list ADT, except its operations are limited. It is analogous to a stack of plates. A *push* is the operation of adding a new element to the top of the stack. A *pop* is the operation of removing the top element from the stack. The rest of the stack is hidden from view. This order is referred to as *last in, first out* (LIFO).

Use: Stacks are the key data structure for recursion and parsing. Having the operations limited means that all operations can implemented easily and be performed in constant time.

Array Implementation: The hidden invariants in an array implementation of a stack are that the elements in the stack are stored in an array starting with the bottom of the stack and that a variable *top* indexes the entry of the array containing the top element. It is not difficult to implement push and pop so that they maintain these invariants. The stack grows to the right as elements are pushed and shrinks to the left as elements are popped. For the code, see Exercise 3.1.1.



Linked List Implementation: As with lists, stacks are often implemented using linked lists. See Section 3.2.

The Queue ADT:

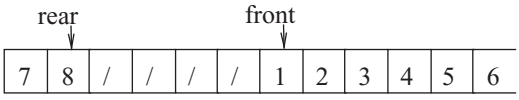
Specification: The queue ADT is also the same as a list ADT, except with a different limited set of operations. A queue is analogous to a line-up for movie tickets. One is able to *insert* an element at the *rear* and *remove* the element that is at the *front*. This order is *first in first out* (FIFO).

Queue Use: An operating system will have a queue of jobs to run and a *network hub* will have a queue of packets to transmit. Again all operations can be implement easily to run in constant time.

Array Implementation:

Trying Small Steps: If the front element is always stored at index 1 of the array, then when the current front is removed, all the remaining elements would need to shift by one to take its place. To save time, once an element is placed in the array, we do not want to move it until it is removed. The effect is that the rear moves to the right as elements arrive, and the front moves to the right as elements are removed. We use two different variables, *front* and *rear*, to index their locations. As the queue migrates to the right, eventually it will reach the end of the array. To avoid getting stuck, we will treat the array as a circle, indexing modulo the size of the array. This allows the queue to migrate around and around as elements arrive and leave.

Hidden Invariants: The elements are stored in order from the entry indexed by *front* to that indexed by *rear* possibly wrapping around the end of the array.



Extremes: It turns out that the cases of a completely empty and a completely full queue are indistinguishable, because with both *front* will be one to the left of *rear*. The easiest solution is not to let the queue get completely full.

Code: See Exercises 3.1.2 and 3.1.3.

Linked List Implementation: Again see Section 3.2.

The Priority Queue ADT:

Specification: A priority queue is still analogous to a line-up for movie tickets. However, in these queues the more important elements are allowed to move to the front of the line. When *inserting* an element, its priority must be specified. This priority can later be changed. When *removing*, the element with the highest priority in the queue is removed and returned. Ties are broken arbitrarily.

Tree Implementations: Heaps, AVL trees, and red–black trees can do each operation in $\Theta(\log n)$ time. See Sections 4.1, 10.2, and 10.4.

The Set ADT:

Specification: A *set* is basically a bag within which you can put any elements that you like. It is the same as a list, except that the elements cannot be repeated or ordered.

Indicator Vector Implementation: If the universe of possible elements is sufficiently small, then a good data structure is to have a Boolean array indexed with each of these possible elements. An entry being true will indicate that the corresponding element is in the set. All set operations can be done in constant time, i.e., in a time independent of the number of items in the set.

Hash Table Implementation: Surprisingly, even if the universe of possible elements is infinite, a similar trick can be done, using a data structure called a *hash table*. A pseudorandom function H is chosen that maps possible elements of the set to the entries $[1, N]$ in the table. It is a deterministic function in that it is easy to compute and always maps an element to the same entry. It is pseudorandom in that it appears to map each element into a random place. Hopefully, all the elements that are in your set happen to be placed into different entries in the table. In this case, one can determine whether or not an element is contained in the set, ask for an arbitrary element from the set, determine the number of elements in the set, iterate through all the elements, and add and delete elements—all in constant time, i.e., independently of the number of items in the set. If collisions occur, meaning that two of your set elements get mapped to the same entry, then there are a number of possible methods to rehash them somewhere else.

The Set System ADT:

Specification: A set system allows you to have a set (or list) of sets. Operations allow the *creation*, *union*, *intersection*, *complementation*, and *subtraction* of sets. The *find* operator determines which set a given element is contained in.

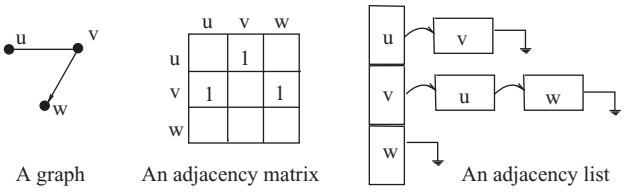
List-of-Indicator-Vectors or Hash-Table Implementations: One way to implement these is to have a list of elements implemented using an array or a linked list where each of these elements is an implementation of a set. What remains is to implement operations that operate on multiple sets. Generally, these operations take $\Theta(n)$ time.

Union–Find Set System Implementation: Another quite surprising result is that on disjoint sets, the union and find operations can be done on average in a constant amount of time for all practical purposes. See the end of this section.

The Dictionary ADT: A dictionary associates a meaning with each word. Similarly, a dictionary ADT associates data with each *key*.

Graphs:

Specification: A *graph* is set of nodes with edges between them. They can represent networks of roads between cities or friendships between people. The key information stored is which pairs of nodes are connected by an edge. Sometimes data, such as weight, cost, or length, can be associated with each edge or with each node. Though a drawing implicitly places each node at some location on the page, a key abstraction of a graph is that the location of a node is not specified. The basic operations are to determine whether an edge is in a graph, to add or delete an edge, and to iterate through the neighbors of a node. There is a huge literature of more complex operations that one might want to do. For example, one might want to determine which nodes have paths between them or to find the shortest path between two nodes. See Chapter 14.



Adjacency Matrix Implementation: This consists of an $n \times n$ matrix with $M(u, v) = 1$ if $\langle u, v \rangle$ is an edge. It requires $\Theta(n^2)$ space (corresponding to the number of potential edges) and $\Theta(1)$ time to access a given edge, but $\Theta(n)$ time to find the edges adjacent to a given node, and $\Theta(n^2)$ to iterate through all the nodes. This is only a problem when the graph is large and sparse.

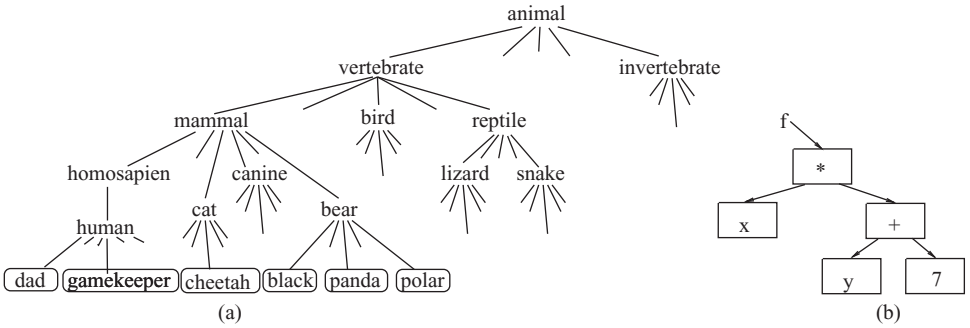


Figure 3.1: Classification tree of animals and a tree representing the expression $f = x \times (y + 7)$.

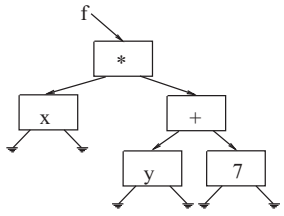
Adjacency List Implementation: It lists for each node the nodes adjacent to it. It requires $\Theta(E)$ space (corresponding to the number of actual edges) and can iterate quickly through the edges adjacent to a give node, but requires time proportional to the degree of a node to access a specific edge.

Trees:

Specification: Data is often organized into a hierarchy. A person has children, who have children of their own. The boss has people under her, who have people under them. The abstract data type for organizing this data is a *tree*.

Uses: There is a surprisingly large list of applications for trees. For two examples see Figure 3.1 and Section 10.5.

Pointer Implementation: Trees are generally implemented by having each node point to each of its children:



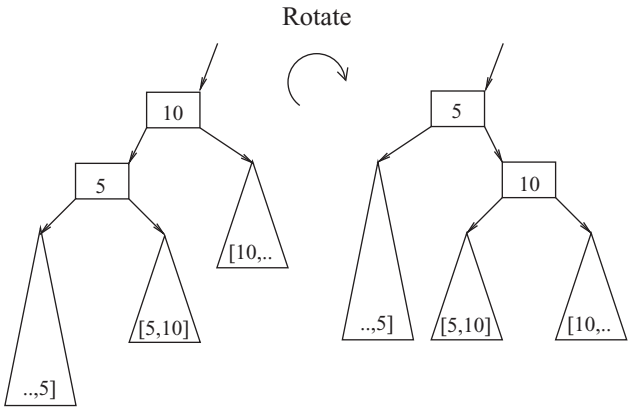
Orders: Imposing rules on how the nodes can be ordered speeds up certain operations.

Binary Search Tree: A binary search tree is a data structure used to store keys along with associated data. The nodes are ordered so that for each node, all the keys in its left subtree are smaller than its key, and all those in the right subtree are larger. Elements can be found in such a tree, using binary search, in $O(height)$ instead of $O(n)$ time. See Sections 4.1 and 10.2.

Heaps: A heap requires that the key of each node be bigger than those of both its children. This allows one to find the maximum key in $O(1)$ time. All updates can be done in $O(\log n)$ time. Heaps are useful for a sorting algorithm known as heap sort and for the implementation of priority queues. See Section 10.4.

Balanced Trees: If a binary tree is balanced, it takes less time to traverse down it, because it has height at most $\log_2 n$. It is too much work to maintain a perfectly balanced tree as nodes are added and deleted. There are, however, a number of data structures that are able to add and delete in $O(\log_2 n)$ time while ensuring that the tree remains almost balanced. Here are two.

AVL Trees: Every node has a *balance factor* of -1, 0, or 1, defined as the difference between the heights of its left and right subtrees. As nodes are added or deleted, this invariant is maintained using rotations like the following (see Exercise 3.1.5):



Red–Black Trees: Every node is either red or black. If a node is red, then both its children are black. Every path from the root to a leaf contains the same number of black nodes. See Exercise 3.1.6.

Balanced Binary Search Tree: By storing the elements in a balanced binary search tree, insertions, deletions, and searches can be done in $\Theta(\log n)$ time.

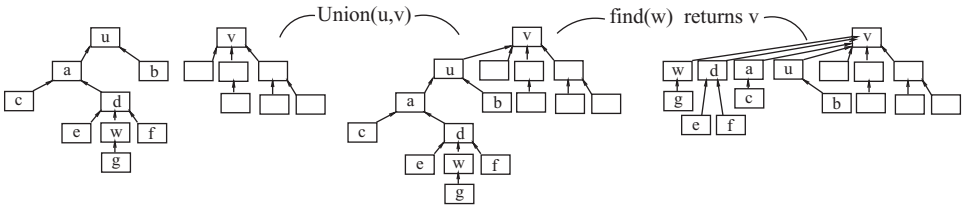
***Union–Find* Set System:** This data structure maintains a number of disjoint sets of elements.

Operations: (1) *Makeset*(v), which creates a new set containing the specified element v; (2) *Find*(v), which determines the *name* of the set containing a specified element (each set is given a distinct but arbitrary name); and (3) *Union*(u, v), which merges the sets containing the specified elements u and v.

Use: One application of this is in the minimum-spanning-tree algorithm in Section 16.2.3.

Running Time: On average, for all practical purposes, each of these operations can be completed in a constant amount of time. More formally, the total time to do m of these operations on n elements is $\Theta(m\alpha(n))$, where α is the *inverse Ackermann's function*. This function is so slow growing that even if n equals the number of atoms in the universe, then $\alpha(n) \leq 4$. See Section 9.3.

Implementation: The data structure used is a rooted tree for each set, containing a node for each element in the set. The difference is that each node points to its parent instead of to its children. The name of the set is the contents of the root node. *Find*(w) is accomplished by tracing up the tree from w to the root u. *Union*(u, v) is accomplished by having node u point to node v. From then on, *Find*(w) for a node w in u's tree will trace up and find v instead. What makes this fast on average is that whenever a *Find* operation is done, all nodes that are encountered during the find are changed to point directly to the root of the tree, collapsing the tree into a shorter tree.

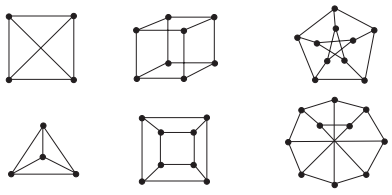


EXERCISE 3.1.1 *Implement the push and pop operations on a stack using an array as described in Section 3.1.*

EXERCISE 3.1.2 *Implement the insert and remove operations on a queue using an array as described in Section 3.1.*

EXERCISE 3.1.3 *When working with arrays, as in Section 3.1, what is the difference between "rear* $=$ *(rear* $+ 1)$ *mod MAX" and "rear* $=$ *(rear mod MAX)* $+ 1$*," and when should each be used?*

Figure: The top row shows three famous graphs: the complete graph on four nodes, the cube, and the Peterson graph. The bottom row shows the same three graphs with their nodes laid out differently.



EXERCISE 3.1.4 *For each of the three pairs of graphs, number the nodes in such the way that* $\langle i, j \rangle$ *is an edge in one if and only if it is an edge in the other.*

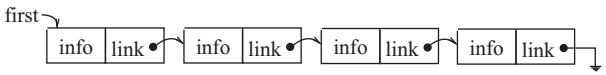
EXERCISE 3.1.5 *(See solution in Part Five.) Prove that the height of an AVL tree with n nodes is* $\Theta(\log n)$.

EXERCISE 3.1.6 *Prove that the height of a red–black tree with n nodes is* $\Theta(\log n)$.

3.2 Link List Implementation

As said, a problem with the array implementation of the list ADT is that the array needs to be allocated some fixed size of memory when it is initialized. A solution to this is to implement these operations using a linked list, which can be expanded in size as needed. This implementation is particularly efficient when the operations are restricted to those of a stack or a queue.

List ADT Specification: A list consists of an ordered sequence of elements. Unlike arrays, it has no empty positions. Elements can be *inserted*, *deleted*, *read*, *modified*, and *searched* for. There are tradeoffs in the running time. Arrays can access the *i*th element in $\Theta(1)$ time, but require $\Theta(n)$ time to insert an element. A *linked list* is an alternative implementation in which the memory allocated can grow and shrink dynamically with the needs of the program. Linked lists allow insertions in $\Theta(1)$ time, but require $\Theta(n)$ time to access the *i*th element. Heaps can do both in $\Theta(\log n)$ time.



Hidden Invariants: In a linked list, each node contains the information for one element and a pointer to the next. The variable *first* points to the first node, and *last* to the last. The last node has its pointer variable contain the value *nil*. When the list contains no nodes, *first* and *last* also point to nil.

Notation: A pointer, such as *first*, is a variable that is used to store the address of a block of memory. The information stored in the *info* field of such a block is denoted by *first.info* in Java and *first*−> *info* in C. We will adopt the first notation. Similarly, *first.link* denotes the pointer field of the node. Being a pointer itself, *first.link.info* denotes the information stored in the second node of the linked list, and *first.link.link.info* in the third.

Adding a Node to the Front: Given a list ADT and new *Info* to store in an element, this operation is to insert an element with this information into the front the list.

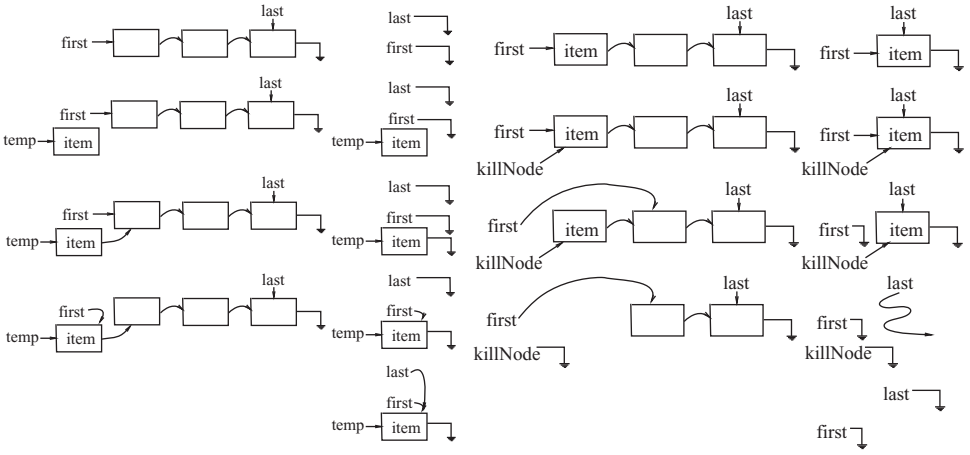


Figure 3.2: *Adding and removing a node from the front of a linked list.*

General Case: We need the following steps (with pseudocode given to the right) for a large and general linked list. See Figure 3.2.

- Allocate space for the new node. New *temp*
- Store the information for the new element. *temp.info* = *Info*
- Point the new node at the rest of the list. *temp.link* = *first*
- Point *first* at the new node. *first* = *temp*

Special Case: The main special case is an empty list. Sometimes we are lucky and the code written for the general case also works for such special cases. Inserting a node starting with both *first* and *last* pointing to *nil*, everything works except for *last*. Add the following to the bottom of the code.

- Point *last* to the new and only node.

```
if( last = nil ) then
   last = temp
end if
```

Whenever adding code to handle a special case, be sure to check that the previously handled cases still are handled correctly.

Removing Node from Front: Given a list ADT, this operation is to remove the element in the front the list and to return the information *Info* stored within it.

General Case:

- Point a temporary variable *killNode* to point to the node to be removed. *killNode* = *first*
- Move *first* to point to the second node. *first* = *first.link*
- Save the value to be returned. *Info* = *killNode.info*
- Deallocate the memory for the first node. free *killNode*
- Return the value. return(*item*)

Special Cases: If the list is already empty, a node cannot be removed. The only other special case occurs when there is one node pointed to by both *first* and *last*. At the end of the code, *first* points to nil, which is correct for an empty list. However, *last* still points to the node that has been deleted. This can be solved by adding the following to the bottom of the code:

- The list becomes empty.

```
if( first = nil ) then
   last = nil
end if
```

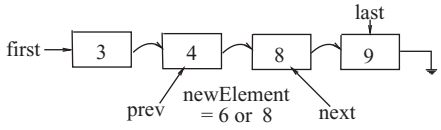
Note that the value of *first* and *last* change. If the routine *Pop* passes these parameters in by value, the routine needs to be written to allow this to happen.

Testing Whether Empty: A routine that returns whether the list is empty returns *true* if *first* = *nil* and *false* otherwise. It does not look like this routine does much, but it serves two purposes. It hides these implementation details from the user, and by calling this routine instead of doing the test directly, the user's code becomes more readable. See Exercise 3.2.1.

Adding Node to End: See Exercise 3.2.2.

Removing Node from End: It is easy to access the last node and delete it, because *last* is pointing at it. However, in order to maintain this invariant, *last* must be pointed at the node that had been the second-to-last node. It takes $\Theta(n)$ time to *walk* down the list from the first node to find this second-to-last node. Luckily, neither stacks nor queues need this operation. For a faster implementation see Exercise 3.2.3.

Walking Down the Linked List: Now suppose that the elements in the linked lists are sorted by the field *info*. When given an *info* value *newElement*, our task is to point the pointer *next* at the first element in the list with that value. The pointer *prev* is to point to the previous element in the list. This needs to be saved, because if it is needed, there is no back pointer to back up to it. If such an element does not exist, then *prev* and *next* are to sandwich the location where this element would go. For example, if *newElement* had either the value 6 or the value 8, the result of the search would be



- Walk down the list
- maintaining the two pointers

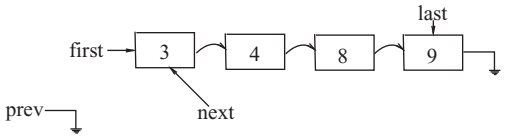
```
loop
   ⟨loop-inv⟩: prev and next point to
      consecutive nodes before or at
      our desired location.
```

- until the desired location is found — exit when $next = nil$ or $next.info \geq newElement$
- pointing *prev* where *next* is pointing — $prev = next$
- and pointing *next* to the next node. — $next = next.link$

end loop

Running Time: This can require $O(n)$ time, where n is the length of the list.

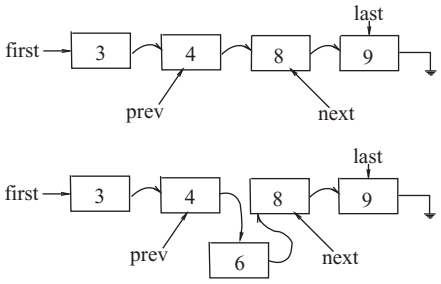
Initialize the Walk: To initially establish the loop invariant, *prev* and *next* must sandwich the location before the first node. We do this as follows:



- Sandwich the location before the first node. — $prev = nil$; $next = first$

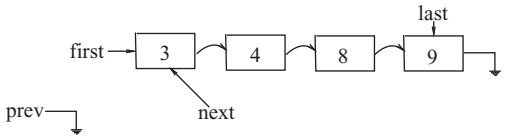
Adding a Node:

Into the Middle: The general case to consider first is adding the node into the middle of the list.



- Allocate space for the new node. — new *temp*
- Store the information for the new element. — $temp.info = item$
- Point the previous node to the new node. — $prev.link = temp$
- Point the new node to the next node. — $temp.link = next$

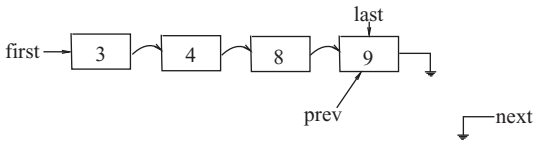
At the Beginning: If the new node belongs at the beginning of the list (say value 2), then $prev.link = temp$ would not work, because *prev* is not pointing at a node. We will replace this line with the following:



- If the new node is to be the first node,
- point *first* at the new node
- else
- point the previous node to the new node.

if *prev* = *nil* then
    first = *temp*
else
    prev.link = *temp*
end if

At the End: Now what if the new node is to be added on the end (e.g. value 12)? The variable *last* will no longer point at the last node. Adding the following code to the bottom will solve the problem:



- If the new node will be the last node,
- point *last* at the new node.

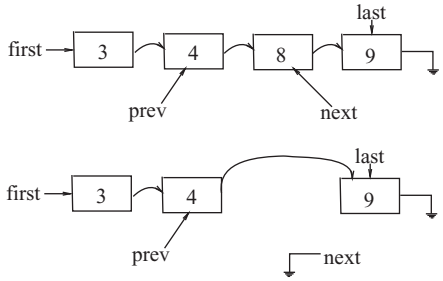
if *prev* = *last* then
    last = *temp*
end if

To an Empty List: Another case to consider is when the initial list is empty. In this case, all the variables, *first*, *last*, *prev*, and *next*, will be nil. The new code works in this case as is.

Compete Code for Adding a Node: One needs to put all of these pieces together into one *insert* routine. See Exercise 3.2.5.

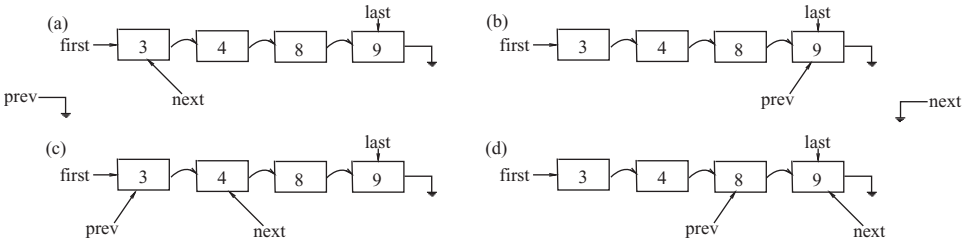
Deleting a Node:

From the Middle: Again the general case to consider first is deleting the node from the middle of the list. We must maintain the linked list *before* destroying the node. Otherwise, we will drop the list.



- Bypass the node being deleted. *prev.link* = *next.link*
- Deallocate the memory pointed to by *next*. free *next*

From the Beginning or the End: As before, you need to consider all the potential special cases. See Exercise 3.2.6.



EXERCISE 3.2.1 *Implement testing whether a linked list is empty.*

EXERCISE 3.2.2 *Implement adding a node to the end of a linked list.*

EXERCISE 3.2.3 *Double pointers: Describe how this operation can be done in* $\Theta(1)$ *time if there are pointers in each node to both the previous and the next node.*

EXERCISE 3.2.4 *(See solution in Part Five.) In the code for walking down the linked list, what effect, if any, would it have if the order of the exit conditions were switched to "exit when next.info* $\geq$ *newElement or next* $=$ *nil"?*

EXERCISE 3.2.5 *Implement the complete code insert that, when given an info value newElement, inserts a new element where it belongs into a sorted linked list. This involves only putting together the pieces just provided.*

EXERCISE 3.2.6 *Implement the complete code Delete that, when given an info value newElement, finds and deletes the first element with this value, if it exists. This involves also considering the four special cases listed for deleting a node from the beginning or the end of a linked list.*

3.3 Merging with a Queue

Merging consists of combining two sorted lists, A and B, into one completely sorted list, C. Here A, B, and C are each implemented as queues. The loop invariant maintained is that the k smallest of the elements are sorted in C. (This is a classic more-of-the-output loop invariant. It is identical to that for selection sort.) The larger elements are still in their original lists A and B. The next smallest element will be either

first element in A or the first element in B. Progress is made by removing the smaller of these two first elements and adding it to the back of C. In this way, the algorithm proceeds like two lanes of traffic merging into one. At each iteration, the first car from one of the incoming lanes is chosen to move into the merged lane. This increases k by one. Initially, with $k = 0$, we simply have the given two lists. We stop when $k = n$. At this point, all the elements will be sorted in C. Merging is a key step in the merge sort algorithm presented in Section 9.1.

algorithm *Merge*(*list* : A, B)

⟨ ***pre-cond***⟩***:*** A and B are two sorted lists.
⟨ ***post-cond***⟩***:*** C is the sorted list containing the elements of the other two.

```
begin
    loop
        ⟨loop-invariant⟩: The k smallest of the elements are sorted in C.
            The larger elements are still in their original lists A and B.
        exit when A and B are both empty
        if( the first in A is smaller than the first in B or B is empty ) then
            nextElement = Remove first from A
        else
            nextElement = Remove first from B
        end if
        Add nextElement to C
    end loop
    return( C )
end algorithm
```

3.4 Parsing with a Stack

One important use of stack is for parsing.

Specifications:

Preconditions: An input instance consists of a string of brackets.

Postconditions: The output indicates whether the brackets match. Moreover, each left bracket is allocated an integer 1, 2, 3, . . . , and each right bracket is allocated the integer from its matching left bracket.

Example:

Input:	(	[	{	}	(	)	]	(	)	{	(	)	}	)
Output:	1	2	3	3	4	4	2	5	5	6	7	7	6	1

The Loop Invariant: Some prefix of the input instance has been read, and the correct integer allocated to each of these brackets. (Thus, it is a more-of-the-input loop invariant.) The left brackets that have been read and not matched are stored along with their integers in left-to-right order in a stack, with the rightmost on top. The variable c indicates the next integer to be allocated to a left bracket.

Maintaining the Loop Invariant: If the next bracket read is a left bracket, then it is allocated the integer c. Not being matched, it is pushed onto the stack. c is incremented. If the next bracket read is a right bracket, then it must match the rightmost left bracket that has been read in. This will be on the top of the stack. The top bracket on the stack is popped. If it matches the right bracket, i.e., we have ( ), {}, or [], then the right bracket is allocated the integer for this left bracket. If not, then an error message is printed.

Initial Conditions: Initially, nothing has been read and the stack is empty.

Ending: If the stack is empty after the last bracket has been read, then the string has been parsed.

Code:

```
algorithm Parsing(s)
⟨pre-cond⟩: s is a string of brackets.
⟨post-cond⟩: Prints out a string of integers that indicate how the brackets
    match.
begin
    i = 0, c = 1
    loop
        ⟨loop-invariant⟩: Prefix s[1, i] has been allocated integers, and
            its left brackets are on the stack.
        exit when i = n
        if(s[i + 1] is a left bracket) then
            print(c)
            push(⟨s[i + 1], c⟩)
            c = c + 1
        elseif(s[i + 1] = right bracket) then
            if( stackempty( ) ) return("Cannot parse")
            ⟨left, d⟩ = pop( )
            if(left matches s[i + 1]) then print(d)
            else return("Cannot parse")
```

```
            else
                return("Invalid input character")
            end if
            i = i + 1
        end loop
        if( stackempty( ) ) return("Parsed") else return("Cannot parse")
    end algorithm
```


Parsing only "( )": If you only need to parse one type of brackets and you only want to know whether or not the brackets match, then you do not need the stack in the above algorithm, only an integer storing the number of left brackets in the stack.

Parsing with Context-Free Grammars: To parse more complex sentences see Chapter 12 and Section 19.8.