5. Implementing Information Structures

5.1 Introduction

This chapter discusses some of the key principles for constructing information structures, such as lists and trees, and discusses primitive implementation in Java as an example. Such structures provide a foundation for the understanding of algorithm design considerations that play a central role in computer science, some of which will be presented in later chapters.

We have already discussed arrays extensively. Arrays are one of the key components of structural computing. The other components are *records* (as they are called in Pascal) or *structs* (as they are called in C). In Java, the class concept is used as an extension of this notion, in the sense that a class provides methods for accessing the data as well as a way to represent the data internally. Classes, coupled with arrays, are the key building blocks for constructing a wide variety of "data structures". Further discussion of the object concept and its uses appears in the following chapter.

5.2 Linked Lists

Linked lists are one of the key structuring devices in computer software. Generally speaking, lists are used to build sequences of data items *incrementally*, especially when we have no advanced notion of how large the sequence will ultimately be. Instead of having to estimate an appropriate initial size, and possible make wholesale adjustments during population of an array, lists allocate item by item, using only as much storage as is needed to hold the items plus a per-item overhead. We describe how linked lists provide a way of implementing the list abstraction found in rex, as well as implementing other list abstractions.

As an example of where linked lists are useful, consider implementing a text editor application. Suppose that the text is organized as a series of paragraphs. The editor provides a way of cutting a paragraph from one part of the document and pasting it in another. In order to make this operation fast, we would avoid storing the paragraphs as a linear array, since this cutting and pasting would entail shifting the elements of the array each time we perform an operation. Instead we would have each paragraph remember the paragraph after it by a *reference* to that paragraph. This kind of use of references is seen whenever we read a newspaper. The blocks of text for an article (which don't coincide with paragraphs necessarily) are scattered on different pages. At the end of a block is a "reference" message "continued on page ...". In a computer, references are not simply pieces of text. Instead they are implemented as memory references or pointers to the next block. Thus the process of finding the target of a reference is very fast, as it can exploit the linear addressing principle.

> The process of going from a reference to its target is called *dereferencing*.

We could effect the cut and paste operation simply by changing references rather than doing a physical cut and paste. Note that some newspapers also provide reverse references "continued from page ...". These would be called "doubly-linked lists" and are mentioned further in the exercises.

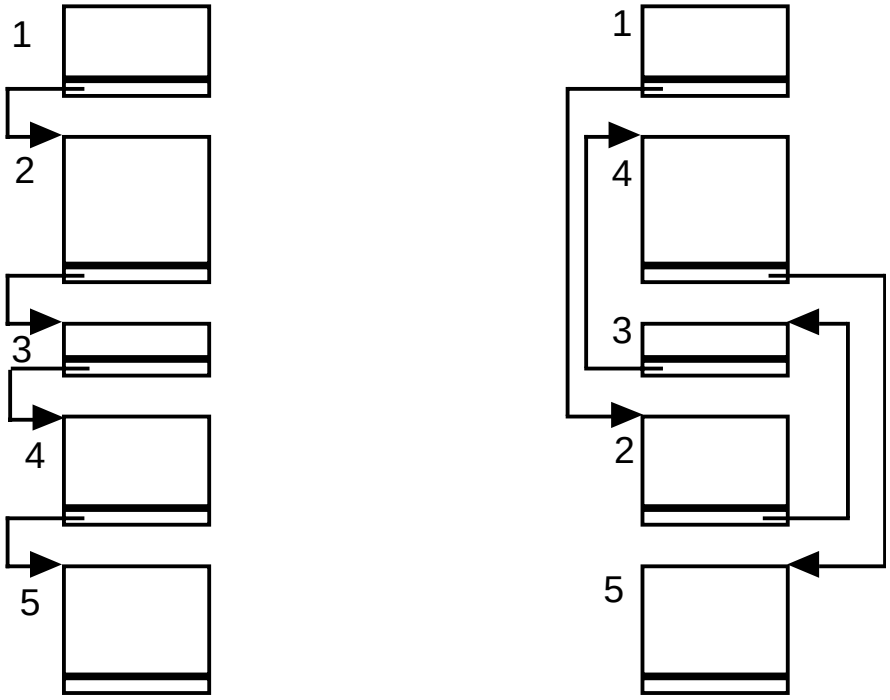

Figure 40: Exchanging paragraphs 2 and 4 by changing references

The key idea of linked lists is to provide a flexible way of connecting units of information together even though they reside in non-contiguous units in computer memory. This is accomplished by constructing a list out of objects, often called **cells**, where each cell contains both a data item and a reference to the next cell in the list. In the final cell of the list, the reference is a special *null reference* that indicates there are no further cells. **The null reference cannot be dereferenced.** In Java, attempting to dereference a null reference will result in a run-time error. In some languages, attempting to do so may produce an unpredictable result. Thus one should always make sure, one way or another, that a reference is not null before dereferencing it.

> The same test for a null reference, which tells whether we are at the end of the list, is also the one that tells whether we have the ability to dereference the reference, that is, whether there is any target.

The figure below shows how a linked-list cell is viewed. The Java code for declaring this type of cell might be:

```
class Cell
{
Item data;
Cell next;
}
```

Here `Item` refers to the type of the data item itself. This can be either a basic type or a defined type. The field `next` is the reference to the next cell. The reason it is not necessary to make any special mention that `next` is a reference is that it is implicit: *In Java, all variables representing objects are implicitly references.* Because the type of object being defined is named *cell* and *cell* is mentioned in the definition, this can also be viewed as **recursive type definition**:

```
Cell_reference = Item x Cell_reference | null
```

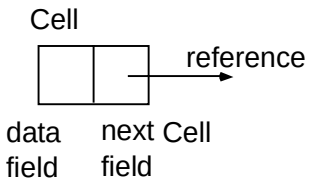

Figure 41: List cell structure

The figure below shows how we depict the case where the value of next is the null reference. This form is used because the next field doesn't point to anything.

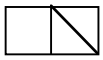

Figure 42: Representing the case of the last element, i.e. the next reference does not point to anything

The following is an example of a linked list with data elements a, b, c, d.

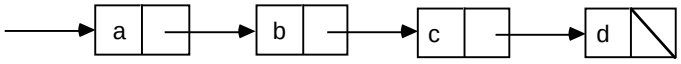

Figure 43: A linked list of four elements

Of course, such structures have already been mentioned in Chapter 2. It is also possible for the elements in the list to be *references* to the actual data elements. This is especially useful in the case that the elements are of non-uniform size, as we might have with a list of strings.

We distinguish between two varieties of lists, but many variations in between are also possible:

Closed lists:
A closed list is a linked list, the cells of which are not normally shared by other lists.

Open lists:
An open list is a linked list in which the cells are shareable by other open lists.

5.3 Open Lists

In an open list, sharing is encouraged, to economize on storage space and to reduce overhead from copying. However, open lists complicate matters of reclaiming unused storage, since we cannot simply delete a cell just because we are done with it in one: such a cell might also be one or more other lists. Java takes care of such reclamation automatically by computing whether each cell is accessible or not when storage becomes in short supply. Cells that are no longer accessible are recycled (deallocated and made available for other uses).

Part of the reason we emphasize open lists here is that they correspond in a natural way to the implementation of lists in rex and related languages. *In simplest terms, a list can be viewed as a reference to a cell.* The empty list is identified with the null reference. For this reason, we could simply rename the `Cell` class previously presented to be a list class. The distinction between list and cell in this simple implementation is purely one of viewpoint. In more complex closed-list implementations to be described later, it will be important to distinguish between lists and cells.

```
class List
{
Item First;       // data in the first cell
List Rest;        // reference to next cell and rest of the list
}
```

To make this more convincing, we show how to implement the rex functions `cons`, `first`, and `rest`.

Function cons constructs a new list from an existing list and a new first element. Were it to be defined anew in rex, the definition would be:

```
cons(E, L) = [E | L];
```

With the definition of List used previously, this function would be defined by including it as a "static method" within the class definition, since Java does not have functions as

such. A static method is one that applies to all objects in the class in general, rather than a particular object.

```
class List
{
Item First;
List Rest;

// return a new list (reference to cell) created from an existing
// list (referenced by Rest) and a data item

static List cons(Item First, List Rest)
  {
  List result = new List;
  result.First = First;
  result.Rest = Rest;
  return result;
  }

}
```

A more elegant way to accomplish this same effect is to introduce a constructor for a List that takes the First and Rest values as arguments. A constructor is called in the context of a new operator, which creates a new List. Adding the constructor, we could rewrite cons:

```
class List
{
Item First;
List Rest;

// construct a List from First and Rest

List(Item First, List Rest)
  {
  this.First = First;
  this.Rest = Rest;
  }


// return a new list (reference to a cell) created from an item
// First and an existing list Rest

static List cons(Item First, List Rest)
  {
  return new List(First, Rest);
  }
}
```

The functions `first` and `rest` would be defined in rex as follows:

```
first( [E | L] ) = E;    // return the first element in a list

rest( [E | L] ) = L;     // return the list of all but the first
```

We now add corresponding functions to the Java implementation:

```
class List
{
Item First;
List Rest;

// construct a List from First and Rest

List(Item First, List Rest)
  {
  this.First = First;
  this.Rest = Rest;
  }


// return a new list (reference to a cell) created from an item
// First and an existing list Rest

static List cons(Item First, List Rest)
  {
  return new List(First, Rest);
  }


// return the first element of a non-empty list

static Item first(List L)
  {
  return L.First;
  }


// return all but the first element of a non-empty list

static List rest(List L)
  {
  return L.Rest;
  }


// return indication of whether list is empty

static boolean isEmpty(List L)
  {
  return L == null;
  }

static boolean nonEmpty(List L)
  {
  return L != null;
  }
}
```

We took the liberty of also adding the functions `isEmpty` and `nonEmpty` to the set of functions being developed, as they will be useful in the following discussion.

Now let's use these definitions by presenting the implementation of some typical rex functions. Consider the definition of function `length` that, as we recall, returns the length of its list argument. The rex definition is:

```
length( [ ] ) => 0;

length( [F | R] ) => length(R) + 1;
```

The translation into Java, which could go inside the class definition above, is:

```
static int length(List L)
  {
  if( isEmpty(L) )  return 0;

  return length(rest(L)) + 1;
  }
```

Notice that each rex rule corresponds to a section of Java code. First we check whether the first rule applies by seeing if the list is empty. If it is not empty, we apply the function recursively and add 1.

As an alternate to implementation of the `length` function, we could use an iterative, non-recursive, solution:

```
static int length(List L)
  {
  int result = 0;

  while( nonEmpty(L) )
    {
    L = rest(L);         // "peel" the first element from the list
    result++;            // record that element in the length
    }

  return result;
  }
```

Although this version is non-recursive, it is perhaps more difficult to understand at a glance, as it introduces another variable to worry about. Depending on the compiler, however, this might well be the preferred way of doing things.

Note that the `length` function should not, and does not, modify its argument list. It merely changes the value of the local variable `L` which is a *reference* to a cell.

Now let's try another example, the function `append`. First in rex:

```
append( [ ], M ) => M;

append( [A | L], M ) => [A | append(L, M)];
```

then in Java:

```
static List append(List L, List M)
  {
  if( isEmpty(L) ) return M;

  return cons(first(L), append(rest(L), M));
  }
```

Notice that the pattern is very similar to the recursive implementation of length. In the case of `append` however, there is no clear and clean way in which the function could be implemented iteratively rather than recursively.

Finally, let's look at a function which was implemented with an accumulator argument earlier: `reverse`. In rex we employed an auxiliary function with two arguments, one of which was the accumulator.

```
reverse( L ) = reverse( L, [ ] );

reverse( [ ], R ) => R;

reverse( [A | L], R ) => reverse( L, [A | R]);
```

A literal translation into Java would be to have two functions corresponding to the two rex functions:

```
static List reverse(List L)
  {
  return reverse(L, null);
  }

static List reverse(List L, List R)
  {
  if( isEmpty(L) ) return R;

  return reverse(rest(L), cons(first(L), R));
  }
```

In the case of `reverse`, we can get rid of the need for the auxiliary function by using iteration. An alternate Java definition is:

```
static List reverse(List L)
  {
  List result = null;
  while( nonEmpty(L) )
    {
    result = cons(first(L), result);
    L = rest(L);
    }
  return result;
  }
```

This version is probably the preferred one, despite it being slightly removed from the original rex definition, since it does not introduce the complication of an auxiliary function.

Exercises

(You may wish to develop rex versions of these solutions first, then translate them to Java.)

1 •• Construct a Java function which will test whether an element occurs in an argument list.

2 •• Construct a Java function which will add an element onto the end of a list, returning a totally new list (leaving the original intact).

3 •• Construct a Java function which will produce as an open list the digits of an argument number in a given radix.

4 •• Construct a Java function which will produce a number given the list of digits in a given radix as an argument.

5 ••• Construct a Java function which will produce a sorted list of the elements from its argument list.

6 ••• Construct a Java function which will produce the list of all subsets of an argument list viewed as a set.

7 ••• Construct a Java function which will return, from a sorted list, a sorted list of the same elements with no duplicates.

5.4 Closed Lists

Some of the techniques for open lists can be used to implement closed lists. Recall that while open lists generally encouraging tail-sharing, closed lists provide a way to prevent. While open lists provide a nice mathematical programming style, dealing with closed lists, e.g. using destructive modification, should also be part of our repertoire. In some cases we use closed lists to save space. Rather than create a new list: we modify the elements or references directly in place. Closed lists can also save time: To append one list to another, we can get by just by modifying references rather than recreating the first list as function `append` does. Because modifying lists is more error prone than creating new ones, we must be more careful if we decide to do any form of sharing. Usually it is best to avoid sharing whenever lists are being modified destructively.

In a sense, a closed list can be implemented by putting a *wrapper* around an open list for which no sharing is to take place. In the absence of sharing, it makes sense to do things which we wouldn't wish to do with open lists, such as keep track of the *last* cell in the list and modify it destructively.

The usual way to provide a wrapper is through a list **header**, a particular object which identifies the list and through which initial accesses are made. Auxiliary information, such as the length of the list, can also be maintained in the header.

With open lists, we may or may not have a header. Our initial primitive exposition was without, and corresponds to implementations in rex and related languages.

The figure below shows a closed list, where auxiliary information, namely a reference to the last cell in the list, is maintained. A type definition for the header might be:

```
class closedList
{
Cell head;
Cell tail;
}
```

where `Cell` is as previously declared.

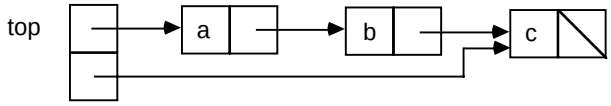

Figure 44: Example of a closed list with 3 elements: a, b, c.

Common uses of closed lists are data containers which maintain objects in a certain order and allow addition or removal only according to a pre-specified discipline:

stack - data are removed in the opposite order of insertion

queue - data are removed in the same order of insertion

We will say more about such containers in later chapters. The figures below depict these uses for linked lists. We leave it to the reader to provide code for the appropriate data operations for these abstractions.

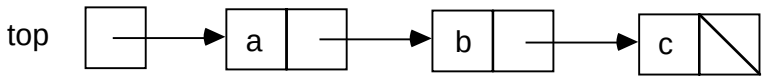

Figure 45: A stack implemented as a closed list.
The header contains a reference to the top cell.

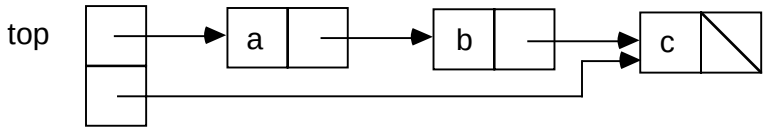

Figure 46: A queue implemented as a closed list.
The oldest cell is removed first.
Insertions take place after the youngest cell.

A more complete presentation of a closed list implementation will come once we have introduced object-oriented concepts in *Object-Oriented Computing*. For now, we will be content with a simple example of what can be done.

Appending Closed Lists

We will use the form of closed list described earlier, with a header that points to both the first and last element in the list. If the list is empty, we will assume that both of these references are null. Before writing any code, it is helpful to draw of picture of what is to happen. Then a series of statements can be constructed to carry out the plan. Finally, special cases, such as empty lists, must be dealt with to make sure the code works for them.

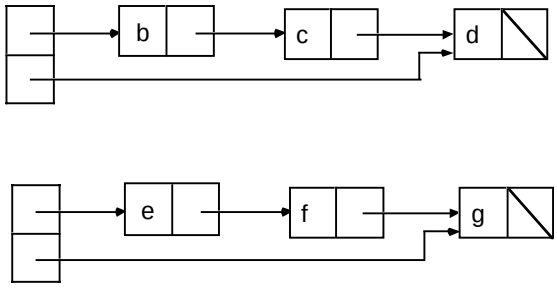

Figure 47: Two closed lists before appending

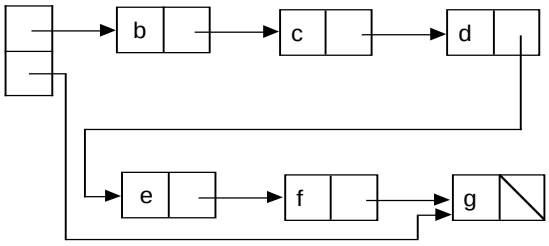

**Figure 48: Closed list after appending second to first.
The second list is no longer shown as a separate entity,
as it would not be advisable to use it once its cells are implicitly shared.**

Assume the following structural definition for the types closedList and cell:

```
class closedList
{
Cell head;
Cell tail;
}

class Cell
{
Item data;
Cell next;
}
```

In order to effect the appending of list `M` to list `L`, we need to do the following:

```
L.tail.next = M.head;  // connect the tail of L to the head of M

L.tail = M.tail;       // install the tail of M as the new tail of L
```

We also have to deal with the null cases. If `L` is empty, then `L.tail` is `null`, so we certainly don't want to dereference it. However, in this case we need to set `L.head` to `M.head`. On the other hand, if `M` is empty, then `M.head` is `null`, so setting `L.tail.next` to `M.head` does no harm. But in this case, `M.tail` will also be `null`. We want to leave `L.tail` as it was, pointing to the tail of `L`. So the final code, packaged as a procedure which modifies `L`, is:

```
void append(closedList L, closedList M)
{
if( L.tail == null )
  L.head = M.head;         // L is null, make L's head be M's
else
  L.tail.next = M.head;    // L is not null, connect L to M

if( M.head != null )
  L.tail = M.tail;         // M is not null, make L's tail be M's
}
```

Exercises

1 •• Construct a procedure *find* which takes a closed list and an argument of type `Item` and returns a reference to the first occurrence of a cell containing that element, or null if the element does not occur.

2 ••• Construct a procedure *reverse* which reverses a closed list in place. Be sure to handle the empty list case.

3 ••• Construct a procedure *insert* which destructively inserts an item into a closed list given a reference to the cell before which it is to be inserted. Assume that if this reference is null, the intention is to insert it at the end of the list.

4 ••• Construct a procedure *delete* which destructively removes an item in a closed list given a reference to the cell to be deleted.

5 ••• A *doubly-linked list* (DLL) is a form of closed list in which each cell has two references, pointing to both the next cell in the list and the previous cell in the list (the latter reference is 0 if there is no previous cell).

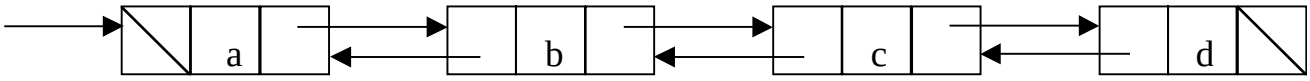

Figure 49: A doubly-linked list of four elements

Give a Java definition for a) the cell of a DLL, and b) a DLL. (Take into account the possibility of a DLL with no elements.)

Develop a set of procedures that do each of the following:

6 •• Find an item in a DLL based on its value. The result is a reference to the cell, or 0 if no such value was found.

7 •• Delete the cell pointed to by a given reference.

8 •• Insert a new cell with a given value following the cell identified by a reference.

9 •• Insert a new cell with a given value before the cell identified by a reference.

10 •• Concatenate two DLL's to form a new DLL.

11 •• Create a DLL with the same values as are contained in an open list.

12 •• Create an open list with the same values as are in a DLL.

13 ••• Think of some applications where a DLL is a more appropriate structure than an ordinary linked list.

14 ••• A *ring* is like a doubly-linked list in which the first and last elements are linked together, as suggested below. This type of structure is used, for example, in some text editors where searches for the next occurrence of a specified string wrap around from the end of the text to the beginning.

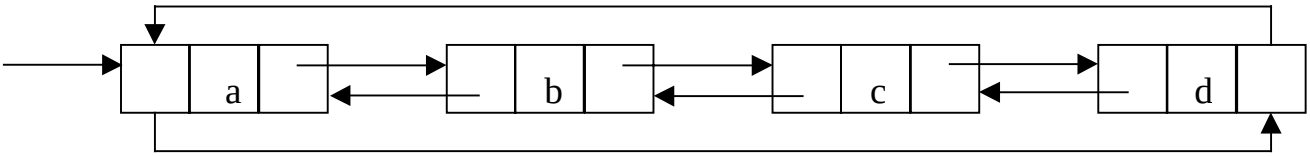

Figure 50: A ring of four elements

Repeat the previous two exercises substituting "ring" for DLL.

15 ••• A *labeled binary tree* (LBT) is structure constructed from nodes similar to those in a doubly-linked list, but the references have an entirely different use. An LBT is a branching version of an open list. Each cell has a data item (called the "label") and two references which themselves represent LBT's.

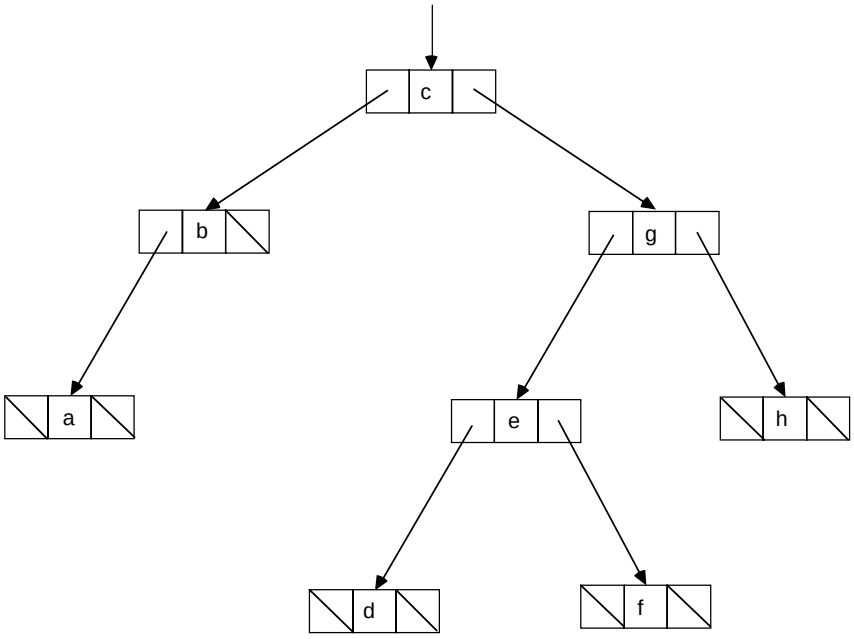

Figure 51: A labeled binary tree

16 ••• Develop a set of abstractions similar to the abstractions *cons*, *first*, *rest*, *null*, etc. for open lists.

17 ••• A *traversal* of an LBT is defined to be a linear list, the elements of which are in one-to-one correspondence with the nodes of the LBT. There are several standard types of traversals, enumerated below. Develop functions which produce each form of traversal from an LBT.

In each of the following cases, the traversal of an empty tree (represented by a null reference) is the empty list.

In an *in-order traversal*, the elements are ordered so that the root element is between an in-order traversal of the left sub-tree and the right sub-tree. An in-order traversal for the tree in the diagram is:

(a b c d e f g h)

since *c* is the root element, (a b) is an in-order traversal of the left sub-tree of the root, and (d e f g h) is an in-order traversal of the right sub-tree of the root. (These facts are established by applying the definition recursively.)

In a *pre-order traversal*, the elements are ordered so that the root element is first, followed by a pre-order traversal of the left sub-tree, then a pre-order traversal of the right sub-tree. For the example above, a pre-order traversal is

(c b a g e d f h)

In a *post-order traversal*, the elements are ordered so that the root is last, and is preceded by a post-order traversal of the left sub-tree, then a post-order traversal of the right sub-tree. For the example above, a post-order traversal is

(a b d f e h g c)

18 •••• In a *level-order* or *breadth-first traversal*, the elements are ordered so that the root is first, the roots of the two sub-trees are next, then the roots of their sub-trees, left-to-right, etc. For the example above, the level-order traversal is

(a b g a e h d f)

Develop a function that produces the level-order traversal of a LBT.

19 ••••• Show that the information in a traversal by itself is insufficient to re-establish the LBT from which it came. Is it possible to use two different traversals to re-establish the LBT? If not, demonstrate. If so, which pairs of traversals work? For those pairs, develop a function that constructs the tree given the traversals.

20 ••• Develop a formula for the number of null references in an LBT as a function of the number of nodes N. Prove your formula by induction.

5.5 Hashing

The principle of hashing combines arrays and lists to achieve an astounding effect: efficient time access to a large volume of data based on key words, numbers, or phrases stored in the data. We present here just one of many variations on the concept. The lists appear to be somewhat closed, but are essentially simple open lists with headers. Typically all addition can take place at the front end. As such, the lists are functioning as write-only stacks, the latter being discussed in more generality in the next chapter.

The problem addressed by hashing is to access "records", e.g. structs, according to some "key" value. The keys could be large numbers or strings for example. If a large number of such records are stored in an array, it can take considerable time to search the array to find the one corresponding to a given key. On the other extreme, we could use linear addressing to access an array by using the key as an index. However, for many such indices there will typically be no record, so much memory space could be wasted with unused locations. It would not be feasible to create such an array for more than a few hundred million keys given current computer technology.

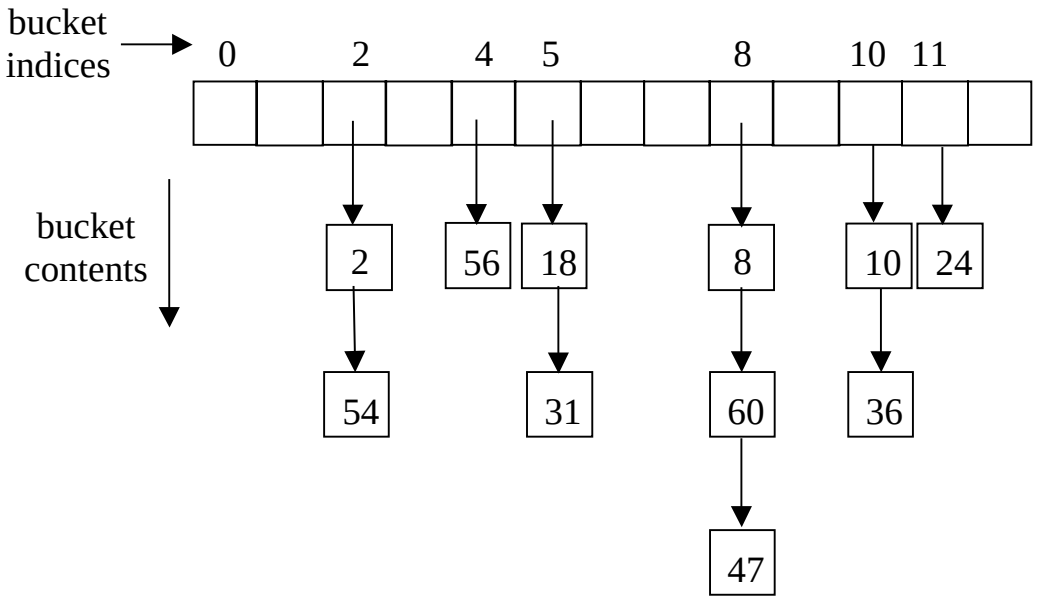

Figure 52: Array of 13 buckets, each a linked list, used for hashing. The numbers in the buckets represent key values.

Hashing "folds" the indexing array so that the same location is used for multiple records. These records are linked together in a list. The records corresponding to any one location are called a "bucket". The bucket is searched linearly. The trick to fast access is to keep the buckets small. This can be done by keeping the index array nominally large and having a way of distributing the records more-or-less "randomly" into the buckets. Based on only the key, we have to know how to find the bucket, both to insert the record in the bucket in the first place and to find out if a record with a given key is in the bucket. The overall structure, as illustrated in the figure, is typically called a *hash table*.

For the example above, we simply took the key value modulo the table size, 13, and used the result as an index. Thus the bucket for key 18 is 18 % 13 ==> 5, while the bucket for key 47 is 47 % 13 ==> 8. Typically such a simple bucket computation will not assure very random distributions. So rather than taking the raw key value mod the table size, we agree in advance on a function

h: key_values → integers

and use

```
h(k) % table_size
```

as our index of the bucket for key *k*. This kind of function is called a *hash function*. By careful choice of *h*, we can get very random distributions and handle arbitrarily large key values. We can even use strings or other structures as key values, by considering those structures to be numerals in a certain radix.

Example Hash Function

The following hash function, *hash_pdg* (for "pretty darn good") works effectively on strings, producing an unsigned long. Before using the resulting value to index the hash table, the value produced by the function is taken modulo the table size. This insures that indices are within range. The function works by using the integer values of successive characters in the string. An accumulator *h* is initialized to 0. Each character is added to *h* multiplied by a constant to obtain a new value of *h*. The multiplier has been chosen to randomize the result as much as possible.

```
unsigned long hash_pdg(char str[ ])
{
int multiplier = 131;
unsigned long h = 0;
int N = str.length();
for( int i = 0; i < N; i++ )
  {
  h = h*multiplier + str[i];
  }
return h;
}
```

The origin of the function is G. H. Gonnet and R. Baeza-Yates, 1991.

5.6 Principle of Virtual Contiguity

We conclude this chapter with a reference-based structure quite different from linked lists. This is an array-like structure for simulating large arrays from smaller ones. The key idea here is to approach the performance availed by the linear addressing principle, without the need for having a single contiguous array.

This principle is used in the structure of so-called virtual memory computers, which are now commonplace. We explained above how we need to have data stored in contiguous memory locations if we are to exploit the linear addressing principle. This requirement can present a problem when very large arrays are involved: it could happen that, at the time a request for a large array is made, the memory has become temporarily "fragmented". That is, there is enough total memory available in terms of the number of storage locations, but no contiguous block that is large enough to hold an array of desired size. The principle of virtual contiguity can be used to "piece together" smaller blocks, with a slight penalty in access time.

Suppose we need to allocate an array requiring 10^6 bytes of memory but there is no block available of that amount. Suppose that there are 100 blocks of 10^4 bytes each available in various blocks. The principle of virtual contiguity allows us to piece these blocks together to give us our 10^6 blocks. This piecing is done by adding a second level of indexing, as implemented by an index array 100 addresses in length. Call the virtual array A and the index array T (for "table"). The values T[0] .... T[99] hold the base addresses of our 100 blocks of 10^4 bytes each. Thus, to access A[i], we first compute i / 100 (using integer division) to find out which block to use, then use i % 100 to access within this block. In equations:

$$\&A[i] \equiv T[i / 100] + i \% 100$$

where &A[i] means the *address* of A[i] in memory.

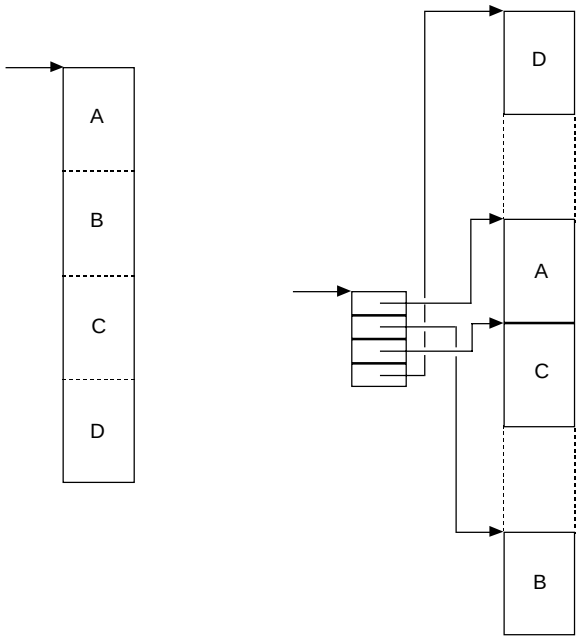

Figure 53: Virtual contiguity:
left: array as perceived by program;
right: array as implemented in linear address-space memory

Virtual Memory

In a true virtual memory system, an additional twist is used along with the principle of virtual contiguity: a table entry T[i] can contain either a memory address or a disk address (as determined by an additional bit in each word). The block being referenced need not be in memory at the time the reference is attempted; instead it is on disk and is brought in on demand. This allows us to "time-share" a relatively small amount of main memory by swapping blocks to and from the disk, giving the illusion of a very large amount of memory. The cost paid for this is a slightly slower overall access time, plus a large penalty if the desired block has to be brought in from disk.

In a virtual memory system, blocks are referred to as **pages** and the array T is called a **page table**. Systems are designed so that they try to keep the most-likely-to-be referenced pages in memory whenever possible. The workability of such schemes relies on what is called the **principle of locality**: programs tend to refer to the same pages over and over again in a nominal time interval. Obviously a virtual memory system does not strictly follow the linear addressing principle of uniform access time when a page is not present on disk. Nonetheless, most people design algorithms as if the linear addressing principle still held, relying on the principle of locality to make linear addressing a good

approximation. Fortunately for the applications programmer, the mechanisms implementing virtual memory are carried out transparently by the system.

Exercises

1. •• Write a program which will do a fast spelling check by using a dictionary stored as a hash table. Populate the table from a dictionary file, such as `/usr/dict/words` which is available in most UNIX® systems. Compare the speed of your program to one that searches the dictionary sequentially.

2. •• Implement a system of arrays that uses the principle of virtual contiguity.

5.7 Chapter Review

Define the following terms:

append
bucket
cell
class
closed list
dereferencing
doubly-linked
hash function
hashing
header
labeled binary tree
level-order
linear addressing principle
linked list
null reference
open list
page
pre-order traversal
post-order traversal
queue
recursive type
ring
virtual memory

5.8 Further Reading

G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures, 2nd ed.*, Addison-Wesley, 1991. [Concise reference on a wide range of algorithmic techniques, with code. Moderate to Difficult]