10. Radix Sorting

☐ The "keys" used to define the order of the records in files for many sorting applications can be very complicated. (For example, consider the ordering function used in the telephone book or a library catalogue.) Because of this, it is reasonable to define sorting methods in terms of the basic operations of "comparing" two keys and "exchanging" two records. Most of the methods we have studied can be described in terms of these two fundamental operations. For many applications, however, it is possible to take advantage of the fact that the keys can be thought of as numbers from some restricted range. Sorting methods which take advantage of the digital properties of these numbers are called radix *sorts.* These methods do not just compare keys: they process and compare pieces of keys.

Radix sorting algorithms treat the keys as numbers represented in a base-M number system, for different values of M (the radix) and work with individual digits of the numbers. For example, consider an imaginary problem where a clerk must sort a pile of cards with three-digit numbers printed on them. One reasonable way for him to proceed is to make ten piles: one for the numbers less than 100, one for the numbers between 100 and 199, etc., place the cards in the piles, then deal with the piles individually, either by using the same method on the next digit or by using some simpler method if there are only a few cards. This is a simple example of a radix sort with $M = 10$. We'll examine this and some other methods in detail in this chapter. Of course, with most computers it's more convenient to work with $M = 2$ (or some power of 2) rather than $M = 10$.

Anything that's represented inside a digital computer can be treated as a binary number, so many sorting applications can be recast to make feasible the use of radix sorts operating on keys which are binary numbers. Unfortunately, Pascal and many other languages intentionally make it difficult to write a program that depends on the binary representation of numbers.

(The reason is that Pascal is intended to be a language for expressing programs in a machine-independent manner, and different computers may use different representations for the same numbers.) This philosophy eliminates many types of "bit-flicking" techniques in situations better handled by fundamental Pascal constructs such as records and sets, but radix sorting seems to be a casualty of this progressive philosophy. Fortunately, it's not too difficult to use arithmetic operations to simulate the operations needed, and so we'll be able to write (inefficient) Pascal programs to describe the algorithms that can be easily translated to efficient programs in programming languages that support bit operations on binary numbers.

Given a (key represented as a) binary number, the fundamental operation needed for radix sorts is extracting a contiguous set of bits from the number. Suppose we are to process keys which we know to be integers between 0 and 1000. We may assume that these are represented by ten-bit binary numbers. In machine language, bits are extracted from binary numbers by using bitwise "and" operations and shifts. For example, the leading two bits of a ten-bit number are extracted by shifting right eight bit positions, then doing a bitwise "and" with the mask 0000000011. In Pascal, these operations can be simulated with **div** and **mod.** For example, the leading two bits of a ten-bit number x are given by (x **div 256)mod** 4. In general, "shift x right k bit positions" can be simulated by computing x **div** 2^k, and "zero all but the j rightmost bits of x" can be simulated by computing x **mod** 2^j. In our description of the radix sort algorithms, we'll assume the existence of a **function** *bits*(x, k, j: integer): integer which combines these operations to return the j bits which appear k bits from the right in x by computing (x **div** 2^k) **mod 23.** For example, the rightmost bit of x is returned by the call *bits*(x, *0, 1*). This function can be made efficient by precomputing (or defining as constants) the powers of 2. Note that a program which uses only this function will do radix sorting whatever the representation of the numbers, though we can hope for much improved efficiency if the representation is binary and the compiler is clever enough to notice that the computation can actually be done with machine language "shift" and "and" instructions. Many Pascal implementations have extensions to the language which allow these operations to be specified somewhat more directly.

Armed with this basic tool, we'll consider two different types of radix sorts which differ in the order in which they examine the bits of the keys. We assume that the keys are not short, so that it is worthwhile to go to the effort of extracting their bits. If the keys are short, then the distribution counting method in Chapter 8 can be used. Recall that this method can sort N keys known to be integers between 0 and $M - 1$ in linear time, using one auxiliary table of size M for counts and another of size N for rearranging records. Thus, if we can afford a table of size 2^b, then b-bit keys can easily be sorted in linear time. Radix sorting comes into play if the keys are sufficiently long (say $b = 32$) that this is not possible.

The first basic method for radix sorting that we'll consider examines the bits in the keys from left to right. It is based on the fact that the outcome of "comparisons" between two keys depend: only on the value of the bits at the first position at which they differ (reading from left to right). Thus, all keys with leading bit 0 appear before all keys with leading bit 1 in the sorted file; among the keys with leading bit 1, all keys with second bit 0 appear before all keys with second bit 1, and so forth. The left-to-right radix sort, which is called radix exchange *sort,* sorts by systematically dividing up the keys in this way.

The second basic method that we'll consider, called *straight radix sort,* examines the bits in the keys from right to left. It is based on an interesting principle that reduces a sort on b-bit keys to b sorts on l-bit keys. We'll see how this can be combined with distribution counting to produce a sort that runs in linear time under quite generous assumptions.

The running times of both basic radix sorts for sorting N records with b bit keys is essentially Nb. On the one hand, one can think of this running time as being essentially the same as N log N, since if the numbers are all different, b must be at least $\log N$. On the other hand, both methods usually use many fewer than Nb operations: the left-to-right method because it can stop once differences between keys have been found; and the right-to-left method, because it can process many bits at once.

Radix Exchange Sort

Suppose we can rearrange the records of a file so that all those whose keys begin with a 0 bit come before all those whose keys begin with a 1 bit. This immediately defines a recursive sorting method: if the two subfiles are sorted independently, then the whole file is sorted. The rearrangement (of the file) is done very much like the partitioning n Quicksort: scan from the left to find a key which starts with a 1 bit, scan from the right to find a key which starts with a 0 bit, exchange, and continue the process until the scanning pointers cross. This leads to a recursive sorting procedure that is very similar to Quicksort:

```
procedure radixexchange(l, r, b: integer);
  var t, i, j: integer;
  begin
  if (r>l) and (b>=0) then
    begin
    i:=l; j:=r;
    repeat
      while (bits(a[i], b, 1)=0) and (i<j) do i:=i+1;
      while (bits(a[j], b, 1)=1) and (i<j) do j:=j-1;
      t:=a[i]; a[i]:=a[j]; a[j]:=t;
    until j=i;
    if bits(a[r], b, 1)=0 then j:=j+1;
    radixexchange(l, j-1, b-1);
    radixexchange(j, r, b-1) ;
    end
  end ;
```

For simplicity, assume that a [*1..N*] contains positive integers less than 2^{32} (that is, they could be represented as 31-bit binary numbers). Then the call *radixexchange(1,* N, 30) will sort the array. The variable *b* keeps track of the bit being examined, ranging from 30 (leftmost) down to 0 (rightmost). (It is normally possible to adapt the implementation of bits to the machine representation of negative numbers so that negative numbers are handled in a uniform way in the sort.)

This implementation is obviously quite similar to the recursive implementation of Quicksort in the previous chapter. Essentially, the partitioning in radix exchange sort is like partitioning in Quicksort except that the number 2^b is used instead of some number from the file as the partitioning element. Since 2^b may not be in the file, there can be no guarantee that an element is put into its final place during partitioning. Also, since only one bit is being examined, we can't rely on sentinels to stop the pointer scans; therefore the tests (i<j) are included in the scanning loops. As with Quicksort, an extra exchange is done for the case *j=i,* but it is not necessary to undo this exchange outside the loop because the "exchange" is a[i] with itself. Also as with Quicksort, some care is necessary in this algorithm to ensure that the nothing ever "falls between the cracks" when the recursive calls are made. The partitioning stops with j=i and all elements to the right of a[i] having 1 bits in the bth position and all elements to the left of a[i] having 0 bits in the bth position. The element a[i] itself will have a **1** bit unless all keys in the file have a 0 in position b. The implementation above has an extra test just after the partitioning loop to cover this case.

The following table shows how our sample file of keys is partitioned and sorted by this method. This table is can be compared with the table given in Chapter 9 for Quicksort, though the operation of the partitioning method is completely opaque without the binary representation of the keys.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	E	O	L	M	I	N	G	E	A	X	T	P	R	S
A	E	A	E	G	I	N	M	L	O					
A	A	E	E	G										
A	A													
A	A													
		E	E	G										
		E	E											
					I	N	M	L	0					
						L	M	N	0					
						L	M							
								N	0					
										S	T	P	R	X
										S	R	P	T	
										P	R	S		
											R	S		
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

The binary representation of the keys used for this example is a simple five-bit code with the ith letter in the alphabet represented by the binary representation of the number i. This is a simplified version of real character codes, which use more bits (seven or eight) and represent more characters (upper/lower case letters, numbers, special symbols). By translating the keys in this table to this five-bit character code, compressing the table so that the subfile partitioning is shown "in parallel" rather than one per line, and then transposing rows and columns, we can see how the leading bits of the keys control partitioning:

```
A 00001   A 0│0001   A 00│001   A 000│01   A 0000│1   A 00001
s 10011   E 0│0101   E 00│101   A 000│01   A 0000│1   A 00001
0 01111   0 0│1111   A 00│001   E 001│01   E 0010│1   E 00101
R 10010   L 0│1100   E 00│101   E 001│01   E 0011│1   E 00101
T 10100   M 0│1101   G 00│111   G 001│11   G 0011│1
I 01001   I 0│1001   I 01│001   I 010│01       0
N 01110   N 0│1110   N 01│110   N 011│10   L 0111│0   L 01100
G 00111   G 0│0111   M 01│101   M 011│01   M 0111│1   M 01101
E 00101   E 0│0101   L 01│100   L 011│00   N 01110    N 01110
x 11000   A 0│0001   0 01│111   0 011│11   0 01111    0 01111
A 00001   x 1│1000   s 10│011   s 100│11   P 100 0
M 01101   T 1│0100   T 10│100   R 100│10   R 10010    R 10010
P 10000   P 1│0000   P 10│000   P 100│00   s 10011    s 10011
L 01100   R 1│0010   R 10│010   T 101│00
E 00101   s 1│0011   x 11│000
```

One serious potential problem for radix sort not brought out in this example is that degenerate partitions (with all keys having the same value for the bit being used) can happen frequently. For example, this arises commonly in real files when small numbers (with many leading zeros) are being sorted. It also occurs for characters: for example suppose that 32-bit keys are made up from four characters by encoding each in a standard eight-bit code then putting them together. Then degenerate partitions are likely to happen at the beginning of each character position, since, for example, lower case letters all begin with the same bits in most character codes. Many other similar effects are obviously of concern when sorting encoded data.

From the example, it can be seen that once a key is distinguished from all the other keys by its left bits, no further bits are examined. This is a distinct advantage in some situations, a disadvantage in others. When the keys are truly random bits, each key should differ from the others after about lg N bits, which could be many fewer than the number of bits in the keys. This is because, in a random situation, we expect each partition to divide the subfile in half. For example, sorting a file with 1000 records might involve only examining about ten or eleven bits from each key (even if the keys are, say, 32-bit keys). On the other hand, notice that all the bits of equal keys are examined. Radix sorting simply does not work well on files which contain many equal keys. Radix exchange sort is actually slightly faster than Quicksort if the keys to be sorted are comprised of truly random bits, but Quicksort can adapt better to less random situations.

Straight Radix Sort

An alternative radix sorting method is tc examine the bits from right to left. This is the method used by old computer-card-sorting machines: a deck of cards was run through the machine 80 times, once for each column, proceeding from right to left. The following example shows how a right-to-left bit-by-bit radix sort works on our file of sample keys.

A 00001	R 10010	T 10100	X 11000	P 10000	A 00001
s 10011	T 10100	x 11000	P 10000	A 00001	A 00001
0 01111	N 01110	P 10000	A 00001	A 00001	E 00101
R 10010	x 11000	L 01100	I 01001	R 10010	E 00101
T 10100	P 10000	A 00001	A 00001	s 10011	G 00111
I 01001	L 01100	I 01001	R 10010	T 10100	I 01001
N 01110	A 00001	E 00101	S 10011	E 00101	L 01100
G 00111	s 10011	A 00001	T 10100	E 00101	M 01101
E 00101	0 01111	M 01101	L 01100	G 00111	N 01110
x 11000	I 01001	E 00101	E 00101	x 11000	0 01111
A 00001	G 00111	R 10010	M 01101	I 01001	P 10000
M 01101	E 00101	N 01110	E 00101	L 01100	R 10010
P 10000	A 00001	s 10011	N 01110	M 01101	s 10011
L 01100	M 01101	0 01111	0 01111	N 01110	T 10100
E 00101	E 00101	G 00111	G 00111	0 01111	x 11000

The ith column in this table is sorted on the trailing i bits of the keys. The ith column is derived from the $(i - 1)$st column by extracting all the keys with a 0 in the ith bit, then all the keys with a 1 in the ith bit.

It's not easy to be convinced that the method works; in fact it doesn't work at all unless the one-bit partitioning process is stable. Once stability has been identified as being important, a trivial proof that the method works can be found: after putting keys with ith bit 0 before those with ith bit 1 (in a stable manner) we know that any two keys appear in proper order (on the basis of the bits so far examined) in the file either because their ith bits are different, in which case partitioning puts them in the proper order, or because their ith bits are the same, in which case they're in proper order because of stability. The requirement of stability means, for example, that the partitioning method used in the radix exchange sort can't be used for this right-to-left sort.

The partitioning is like sorting a file with only two values, and the distribution counting sort that we looked at in Chapter 8 is entirely appropriate for this. If we assume that $M = 2$ in the distribution counting program and replace *a[i]* by *bits(a[i], k, 1)*, then that program becomes a method for sorting the elements of the array a on the bit *k* positions from the right and putting the result in a temporary array *t*. But there's no reason to use A4 = 2; in fact we should make M as large as possible, realizing that we need a table of M counts. This corresponds to using m bits at a time during the sort, with $M = 2^m$. Thus, straight radix sort becomes little more than a generalization of distribution counting sort, as in the following implementation for sorting *a[1..N]* on the *b* rightmost bits:

```
procedure straightradix( b: integer) ;
  var i, j, pass: integer;
  begin
  for pass:=0 to (b div m)-1 do
    begin
    for j:=O to M-l do count[j] :=0;
    for i:=1 to N do
      count[bits(a[i],pass*m,  m)] :=count[bits(a[i], pass*m,  m)]+1;
    for j:=1 to M-l do
      count[j]:=count[j-1]+count[j];
    for i:=N downto 1 do
      begin
      t[count[bits(a[i], pass*m, m)]]:=a[i];
      count[bits(a[i], pass*m, m)]:=count[bits(a[i], pass*m, m)]-1;
      end ;
    for i:=1 to N do a[i]:=t[i];
    end ;
  end;
```

For clarity, this procedure uses two calls on bits to increment and decrement count, when one would suffice. Also, the correspondence $M = 2^m$ has been preserved in the variable names, though some versions of "pascal" can't tell the difference between *m* and *M*.

The procedure above works properly only if *b* is a multiple of *m*. Normally, this is not a particularly restrictive assumption for radix sort: it simply corresponds to dividing the keys to be sorted into an integral number of equal size pieces. When $m=b$ we have distribution counting sort; when $m=1$ we have *straight radix sort,* the right-to-left bit-by-bit radix sort described in the example above.

The implementation above moves the file from a to t during each distribution counting phase, then back to a in a simple loop. This "array copy" loop could be eliminated if desired by making two copies of the distribution counting code, one to sort from a into t, the other to sort from t into a.

A Linear *Sort*

The straight radix sort implementation given in the previous section makes b/m passes through the file. By making m large, we get a very efficient sorting method, as long as we have $M = 2^m$ words of memory available. A reasonable choice is to make m about one-fourth the word-size *(b/4),* so that the radix sort is four distribution counting passes. The keys are treated as base-M numbers, and each (base--M) digit of each key is examined, but there are only four digits per key. (This directly corresponds with the architectural organization of many computers: one typical organization is to have 32-bit words, each consisting of four 8-bit bytes. The bits procedure then winds up extracting particular bytes from words in this case, which obviously can be done very efficiently on such computers.) Now, each distribution counting pass is linear, and since there are only four of them, the entire sort is linear, certainly the best performance we could hope for in a sort.

In fact, it turns out that we can get by with only two distribution counting passes. (Even a careful reader is likely to have difficulty telling right from left by this time, so some caution is called for in trying to understand this method.) This can be achieved by taking advantage of the fact that the file will be almost sorted if only the leading $b/2$ bits of the bbit keys are used. As with Quicksort, the sort can be completed efficiently by using insertion sort on the whole file afterwards. This method is obviously a trivial modification to the implementation above: to do a right-to-left sort using the leading half of the keys, we simply start the outer loop at pass=b div $(2*m)$ rather than *pass=1.* Then a conventional insertion sort can be used on the nearly-ordered file that results. To become convinced that a file sorted on its leading bits is quite well-ordered, the reader should examine the first few columns of the table for radix exchange sort above. For example, insertion sort run on the the file sorted on the first three bits would require only six exchanges.

Using two distribution counting passes (with m about one-fourth the word size), then using insertion sort to finish the job will yield a sorting method that is likely to run faster than any of the others that we've seen for large files whose keys are random bits. Its main disadvantage is that it requires an extra array of the same size as the array being sorted. It is possible to eliminate the extra array using linked-list techniques, but extra space proportional to N (for the links) is still required.

A linear sort is obviously desirable for many applications, but there are reasons why it is not the panacea that it might seem. First, it really does depend on the keys being random bits, randomly ordered. If this condition is not satisfied, severely degraded performance is likely. Second, it requires extra space proportional the size of the array being sorted. Third, the "inner loop" of the program actually contains quite a few instructions, so even though it's linear, it won't be as much faster than Quicksort (say) as one might expect, except for quite large files (at which point the extra array becomes a real liability). The choice between Quicksort and radix sort is a difficult one that is likely to depend not only on features of the application such as key, record, and file size, but also on features of the programming and machine environment that relate to the efficiency of access and use of individual bits. Again, such tradeoffs need to be studied by an expert and this type of study is likely to be worthwhile only for serious sorting applications.

□

Exercises

1. Compare the number of exchanges used by radix exchange sort with the number of exchanges used by Quicksort for the file 001,011,101,110, 000,001,010,111,110,010.

2. Why is it not as important to remove the recursion from the radix exchange sort as it was for Quicksort?

3. Modify radix exchange sort to skip leading bits which are identical on all keys. In what situations would this be worthwhile?

4. True or false: the running time of straight radix sort does not depend on the order of the keys in the input file. Explain your answer.

5. Which method is likely to be faster for a file of all equal keys: radix exchange sort or straight radix sort?

6. True or false: both radix exchange sort and straight radix sort examine all the bits of all the keys in the file. Explain your answer.

7. Aside from the extra memory requirement, what is the major disadvantage to the strategy of doing straight radix sorting on the leading bits of the keys, then cleaning up with insertion sort afterwards?

8. Exactly how much memory is required to do a 4-pass straight radix sort of N b-bit keys?

9. What type of input file will make radix exchange sort run the most slowly (for very large N)?

10. Empirically compare straight radix sort with radix exchange sort for a random file of 1000 32-bit keys.