State $q_2$ searches left for an X and enters state $q_0$ upon finding it, moving right, to the leftmost 0, as it changes state. As $M$ searches right in state $q_1$, if a B or X is encountered before a 1, then the input is rejected; either there are too many 0's or the input is not in $0^*$.

State $q_0$ has another role. If, after state $q_2$ finds the rightmost X, there is a Y immediately to its right, then the 0's are exhausted. From $q_0$, scanning Y, state $q_3$ is entered to scan over Y's and check that no 1's remain. If the Y's are followed by a B, state $q_4$ is entered and acceptance occurs; otherwise the string is rejected. The function $\delta$ is shown in Fig. 7.2. Figure 7.3 shows the computation of $M$ on input 0011. For example, the first move is explained by the fact that $\delta(q_0, 0) = (q_1, X, R)$; the last move is explained by the fact that $\delta(q_3, B) = (q_4, B, R)$. The reader should simulate $M$ on some rejected inputs such as 001101, 001, and 011.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>Symbol</th>
<th>X</th>
<th>Y</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>(q_1, X, R)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(q_3, Y, R)</td>
<td>-</td>
</tr>
<tr>
<td>$q_1$</td>
<td>(q_2, 0, R)</td>
<td>(q_2, Y, L)</td>
<td>-</td>
<td>(q_2, Y, R)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>(q_2, 0, L)</td>
<td>-</td>
<td>(q_0, X, R)</td>
<td>(q_2, Y, L)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(q_2, Y, R)</td>
<td>(q_4, B, R)</td>
<td>-</td>
</tr>
<tr>
<td>$q_4$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 7.2 The function $\delta$.

| $q_0, q_1, q_2, q_3, q_4$ | - | - | - | - | - | - |
| $q_0011$ | X | q_1, 011 | X, q_0, 11 | X, q_0, Y, 1 | - |
| $q_2$ | X, Y, 1 | X, q_0, Y, 1 | X, X, q_1, Y | X, X, Y, q_1 | - |
| $q_3$ | X, q_2, Y | X, q_2, Y, Y | X, q_0, Y, Y | X, q_0, Y, q_3 | - |
| $q_4$ | X, Y, q_2 | X, Y, Y, q_2 | - | - | - |

Fig. 7.3 A computation of $M$.

### Computable Languages and Functions

A language that is accepted by a Turing machine is said to be recursively enumerable (r.e.). The term "enumerable" derives from the fact that it is precisely these languages whose strings can be enumerated (listed) by a Turing machine. "Recursively" is a mathematical term predating the computer, and its meaning is similar to what the computer scientist would call "recursion." The class of r.e. languages is very broad and properly includes the CFL's.

The class of r.e. languages includes some languages for which we cannot mechanically determine membership. If $L(M)$ is such a language, then any Turing machine recognizing $L(M)$ must fail to halt on some input not in $L(M)$. If $w$ is in $L(M)$, $M$ eventually halts on input $w$. However, as long as $M$ is still running on some input, we can never tell whether $M$ will eventually accept if we let it run long enough, or whether $M$ will run forever.

It is convenient to single out a subclass of the r.e. sets, called the recursive sets, which are those languages accepted by at least one Turing machine that halts on all inputs (note that halting may or may not be preceded by acceptance). We shall see in Chapter 8 that the recursive sets are a proper subclass of the r.e. sets. Note also that by the algorithm of Fig. 6.8, every CFL is a recursive set.

**The Turing machine as a computer of integer functions**

In addition to being a language acceptor, the Turing machine may be viewed as a computer of functions from integers to integers. The traditional approach is to represent integers in unary; the integer $i \geq 0$ is represented by the string $0^i$. If a function has $k$ arguments, $i_1, i_2, \ldots, i_k$, then these integers are initially placed on the tape separated by Y's, as $0^i10^i21 \cdots 10^k$.

If the TM halts (whether or not in an accepting state) with a tape consisting of $0^m$ for some $m$, then we say that $f(i_1, i_2, \ldots, i_k) = m$, where $f$ is the function of $k$ arguments computed by this Turing machine. Note that one TM may compute a function of one argument, a different function of two arguments, and so on. Also note that if TM $M$ computes function $f$ of $k$ arguments, then $f$ needs not have a value for all different $k$-tuples of integers $i_1, \ldots, i_k$.

If $f(i_1, \ldots, i_k)$ is defined for all $i_1, \ldots, i_k$, then we say $f$ is a total recursive function. A function $f(i_1, \ldots, i_k)$ computed by a Turing machine is called a partial recursive function. In a sense, the partial recursive functions are analogous to the r.e. languages, since they are computed by Turing machines that may or may not halt on a given input. The total recursive functions correspond to the recursive languages, since they are computed by TM's that always halt. All common arithmetic functions on integers, such as multiplication, $n!$, $\log_2 n$ and $2^n$ are total recursive functions.

**Example 7.2** Proper subtraction $m \sim n$ is defined to be $m - n$ for $m \geq n$, and zero for $m < n$.

The TM $M = (\{q_0, q_1, \ldots, q_6\}, \{0, 1\}, \{0, 1, B\}, \delta, q_0, B, \emptyset)$ defined below, starts with $0^n1^m$ on its tape, halts with $0^n1^m$ on its tape. $M$ repeatedly replaces its leading 0 by blank, then searches right for a 1 followed by a 0 and changes the 0 to 1. Next, $M$ moves left until it encounters a blank and then repeats the cycle. The repetition ends if

1. Searching right for a 0, $M$ encounters a blank. Then, the n 0's in $0^n1^m$ have all been changed to 1's, and $n + 1$ of the $m$ 0's have been changed to B. $M$ replaces the $n + 1$ 1's by a 0 and $n$ B's, leaving $m - n$ 0's on its tape.
ii) Beginning the cycle, $M$ cannot find a 0 to change to a blank, because the first $m$ 0's already have been changed. Then $n \geq m$, so $m - n = 0$. $M$ replaces all remaining 1's and 0's by $B$.

The function $\delta$ is described below.

1) $\delta(q_0, 0) = (q_1, B, R)$
Begin the cycle. Replace the leading 0 by $B$.

2) $\delta(q_1, 0) = (q_2, 0, R)$
$\delta(q_1, 1) = (q_2, 1, R)$
Search right, looking for the first 1.

3) $\delta(q_2, 1) = (q_2, 1, R)$
$\delta(q_2, 0) = (q_3, 1, L)$
Search right past 1's until encountering a 0. Change that 0 to 1.

4) $\delta(q_3, 0) = (q_3, 0, L)$
$\delta(q_3, 1) = (q_3, 1, L)$
$\delta(q_3, B) = (q_0, B, R)$
Move left to a blank. Enter state $q_0$ to repeat the cycle.

5) $\delta(q_4, B) = (q_5, B, L)$
$\delta(q_4, 1) = (q_5, B, L)$
$\delta(q_4, 0) = (q_4, 0, L)$
$\delta(q_4, B) = (q_0, 0, R)$
If in state $q_4$ a $B$ is encountered before a 0, we have situation (i) described above. Enter state $q_4$ and move left, changing all 1's to $B$'s until encountering a $B$. This $B$ is changed back to a 0, state $q_6$ is entered, and $M$ halts.

6) $\delta(q_6, 1) = (q_7, B, R)$
$\delta(q_6, 0) = (q_7, B, R)$
$\delta(q_7, B) = (q_6, B, R)$
If in state $q_6$ a 1 is encountered instead of a 0, the first block of 0's has been exhausted, as in situation (ii) above. $M$ enters state $q_7$ to erase the rest of the tape, then enters $q_6$ and halts.

A sample computation of $M$ on input 0010 is:

$q_0010 \rightarrow Bq_10 \rightarrow B0q_11 \rightarrow B01q_20 \rightarrow$
$B0q_30 \rightarrow Bq_30 \rightarrow Bq_01 \rightarrow q_3 \rightarrow B01q_2 \rightarrow B01q_3 \rightarrow$
$BBq_41 \rightarrow BBq_41 \rightarrow Bq_4 \rightarrow B0q_6$

On input 0100, $M$ behaves as follows:

$q_00100 \rightarrow Bq_1100 \rightarrow B1q_200 \rightarrow Bq_3110 \rightarrow$
$q_4 \rightarrow B110 \rightarrow Bq_0110 \rightarrow BBq_410 \rightarrow BBBBBq_5 \rightarrow$
$BBBBBBq_6$

### 7.4 TECHNIQUES FOR TURING MACHINE CONSTRUCTION

Designing Turing machines by writing out a complete set of states and a next-move function is a noticeably un rewarding task. In order to describe complicated Turing machine constructions, we need some "higher level" conceptual tools. In this section, we shall discuss the principal ones.

#### Storage in the finite control

The finite control can be used to hold a finite amount of information. To do so, the state is written as a pair of elements, one exercising control and the other storing a symbol. It should be emphasized that this arrangement is for conceptual purposes only. No modification in the definition of the Turing machine has been made.

**Example 7.3** Consider a Turing machine $M$ that looks at the first input symbol, records it in its finite control, and checks that the symbol does not appear elsewhere on its input. Note that $M$ accepts a regular set, but $M$ will serve for demonstration purposes:

$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, \{q_0, B\}, B, F)$

where $Q = \{q_0, q_1\} \times \{0, 1, B\}$.
That is, $Q$ consists of the pairs $[q_0, 0]$, $[q_0, 1]$, $[q_0, B]$, $[q_1, 0]$, $[q_1, 1]$, and $[q_1, B]$. The set $F$ is $\{[q_1, B]\}$. The intention is that the first component of the state controls the action, while the second component "remembers" a symbol.

We define $\delta$ as follows.

1) a) $\delta([q_0, B], 0) = ([q_0, 0], 0, R)$, b) $\delta([q_0, B], 1) = ([q_1, 1], 1, R)$.

Initially, $q_0$ is the control component of the state, and $M$ moves right. The first component of $M$'s state becomes $q_1$, and the first symbol seen is stored in the second component.

2) a) $\delta([q_1, 0], 0) = ([q_1, 0], 1, R)$, b) $\delta([q_1, 1], 0) = ([q_1, 1], 0, R)$.

If $M$ has a 0 stored and sees a 1 or vice versa, then $M$ continues to move to the right.
indicating the position of the i\textsuperscript{th} tape head for $1 \leq i \leq k$. The details are left for an exercise.

### Off-line Turing machines

An off-line Turing machine is a multitape TM whose input tape is read-only. Usually we surround the input by endmarkers $\dagger$ on the left and $\ddagger$ on the right. The Turing machine is not allowed to move the input tape head off the region between $\dagger$ and $\ddagger$. It should be obvious that the off-line TM is just a special case of the multitape TM, and therefore is no more powerful than any of the models we have considered. Conversely, an off-line TM can simulate any TM $M$ by using one more tape than $M$. The first thing the off-line TM does is copy its own input onto the extra tape, and then simulates $M$ as if the extra tape were $M$'s input. The need for off-line TM's will become apparent in Chapter 12, when we consider limiting the amount of storage space to less than the input length.

### 7.6 Church's Hypothesis

The assumption that the intuitive notion of "computable function" can be identified with the class of partial recursive functions is known as Church's hypothesis or the Church-Turing thesis. While we cannot hope to "prove" Church's hypothesis as long as the informal notion of "computable" remains an informal notion, we can give evidence for its reasonableness. As long as our intuitive notion of "computable" places no bound on the number of steps or the amount of storage, it would seem that the partial recursive functions are intuitively computable, although some would argue that a function is not "computable" unless we can bound the computation in advance or at least establish whether or not the computation eventually terminates.

What is less clear is whether the class of partial recursive functions includes all "computable" functions. Logicians have presented many other formalisms such as the $\lambda$-calculus, Post systems, and general recursive functions. All have been shown to define the same class of functions, i.e., the partial recursive functions. In addition, abstract computer models, such as the random access machine (RAM), also give rise to the partial recursive functions.

The RAM consists of an infinite number of memory words, numbered 0, 1, ..., each of which can hold any integer, and a finite number of arithmetic registers capable of holding any integer. Integers may be decoded into the usual sorts of computer instructions. We shall not define the RAM model more formally, but it should be clear that if we choose a suitable set of instructions, the RAM may simulate any existing computer. The proof that the Turing machine formalism is as powerful as the RAM formalism is given below. Some other formalisms are discussed in the exercises.

---

### Simulation of random access machines by Turing machines

**Theorem 7.6** A Turing machine can simulate a RAM, provided that the elementary RAM instructions can themselves be simulated by a TM.

**Proof** We use a multitape TM $M$ to perform the simulation. One tape of $M$ holds the words of the RAM that have been given values. The tape looks like

$$\#0v_0\#1v_1\#10\#v_2\#\ldots$$

where $v_i$ is the contents, in binary, of the $i$\textsuperscript{th} word. At all times, there will be some finite number of words of the RAM that have been used, and $M$ needs only to keep a record of values up to the largest numbered word that has been used so far.

The RAM has some finite number of arithmetic registers. $M$ uses one tape to hold each register's contents, one tape to hold the location counter, which contains the number of the word from which the next instruction is to be taken, and one tape as a memory address register on which the number of a memory word may be placed.

Suppose that the first 10 bits of an instruction denote one of the standard computer operations, such as LOAD, STORE, ADD, and so on, and that the remaining bits denote the address of an operand. While we shall not discuss the details of implementation for all standard computer instructions, an example should make the techniques clear. Suppose the location counter tape of $M$ holds number $i$ in binary. $M$ searches its first tape from the left, looking for $\#i\#\#$. If a blank is encountered before finding $\#i\#$, there is no instruction in word $i$, so the RAM and $M$ halt. If $\#i\#$ is found, the bits following $*$ up to the next $\#$ are examined. Suppose the first 10 bits are the code for "ADD to register 2," and the remaining bits are some number $j$ in binary. $M$ adds 1 to $i$ on the location counter tape and copies $j$ onto the memory address tape. Then $M$ searches for $\#j\#\$ on the first tape, again starting from the left (note that $\#0\#$ marks the left end). If $\#j\#$ is not found, we assume word $j$ holds 0 and go on to the next instruction of the RAM. If $\#j\#v_j\#$ is found, $v_j$ is added to the contents of register 2, which is stored on its own tape. We then repeat the cycle with the next instruction.

Observe that although the RAM simulation used a multitape Turing machine, by Theorem 7.2 a single tape TM would suffice, although the simulation would be more complicated.

---

### 7.7 Turing Machines as Enumerators

We have viewed Turing machines as recognizers of languages and as computers of functions on the nonnegative integers. There is a third useful view of Turing machines, as generating devices. Consider a multitape TM $M$ that uses one tape as an output tape, on which a symbol, once written, can never be changed, and whose
tape head never moves left. Suppose also that on the output tape, \( M \) writes strings over some alphabet \( \Sigma \), separated by a marker symbol \#\#. We can define \( G(M) \), the language generated by \( M \), to be the set of \( w \) in \( \Sigma^* \) such that \( w \) is eventually printed between a pair of \#'s on the output tape.

Note that unless \( M \) runs forever, \( G(M) \) is finite. Also, we do not require that words be generated in any particular order, or that any particular word be generated only once. If \( L = G(M) \) for some TM \( M \), then \( L \) is an r.e. set, and conversely. The recursive sets also have a characterization in terms of generators; they are exactly the languages whose words can be generated in order of increasing size. These equivalences will be proved in turn.

**Characterization of r.e. sets by generators**

**Lemma 7.1** If \( L = G(M_1) \) for some TM \( M_1 \), then \( L \) is an r.e. set.

**Proof** Construct TM \( M_2 \) with one more tape than \( M_1 \). \( M_2 \) simulates \( M_1 \) using all but \( M_2 \)'s input tape. Whenever \( M_1 \) prints \# on its output tape, \( M_2 \) compares its input with the word just generated. If they are the same, \( M_2 \) accepts; otherwise \( M_2 \) continues to simulate \( M_1 \). Clearly \( M_2 \) accepts an input \( x \) if and only if \( x \) is in \( G(M_1) \). Thus \( L(M_2) = G(M_1) \).

The converse of Lemma 7.1 is somewhat more difficult. Suppose \( M_1 \) is a recognizer for some r.e. set \( L \subseteq \Sigma^* \). Our first (and unsuccessful) attempt at designing a generator for \( L \) might be to generate the words in \( \Sigma^* \) in some order \( w_0, w_1, w_2, \ldots \), run \( M_1 \) on \( w_n \), and if \( M_1 \) accepts, generate \( w_n \). Then run \( M_1 \) on \( w_{n+1} \), generating \( w_{n+1} \) if \( M_1 \) accepts, and so on. This method works if \( M_1 \) is guaranteed to halt on all inputs. However, as we shall see in Chapter 8, there are languages \( L \) that are r.e. but not recursive. If such is the case, we must contend with the possibility that \( M_1 \) never halts on some \( w_i \). Then \( M_2 \) never considers \( w_{i+1}, w_{i+2}, \ldots \), and so cannot generate any of these words, even if \( M_1 \) accepts them.

We must therefore avoid simulating \( M_1 \) indefinitely on any one word. To do this we fix an order for enumerating words in \( \Sigma^* \). Next we develop a method of generating all pairs \((i, j)\) of positive integers. The simulation proceeds by generating a pair \((i, j)\) and then simulating \( M_1 \) on the \( i \)th word, for \( j \) steps.

We fix a canonical order for \( \Sigma^* \) as follows. List words in order of size, with words of the same size in "numerical order." That is, let \( \Sigma = \{a_0, a_1, \ldots, a_n\} \), and imagine that \( a_i \) is the "digit" \( i \) in base \( k \). Then the words of length \( n \) are the numbers 0 through \( k^n - 1 \) written in base \( k \). The design of a TM to generate words in canonical order is not hard, and we leave it as an exercise.

**Example 7.9** If \( \Sigma = \{0, 1\} \), the canonical order is \( \epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots \).

Note that the seemingly simpler order in which we generate the shortest representation of 0, 1, 2, \ldots in base \( k \) will not work as we never generate words like \( a_0 a_0 a_1 \), which have "leading 0's."

Next consider generating pairs \((i, j)\) such that each pair is generated after some finite amount of time. This task is not so easy as it seems. The naive approach, \((1, 1), (1, 2), (1, 3), \ldots \), never generates any pairs with \( i > 1 \). Instead, we shall generate pairs in order of the sum \( i + j \), and among pairs of equal sum, in order of increasing \( i \). That is, we generate \((1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), (1, 4), \ldots \), the pair \((i, j)\) is the \( \left(\frac{(i + j - 1)(i + j - 2)}{2} + i\right) \)th pair generated. Thus this ordering has the desired property that there is a finite time at which any particular pair \((i, j)\) is generated.

A TM generating pairs \((i, j)\) in this order in binary is easy to design, and we leave its construction to the reader. We shall refer to such a TM as the pair generator in the future. Incidentally, the ordering used by the pair generator demonstrates that pairs of integers can be put into one-to-one correspondence with the integers themselves, a seemingly paradoxical result that was discovered by Georg Cantor when he showed that the rationals (which are really the ratios of two integers) are equinumerous with the integers.

**Theorem 7.7** A language is r.e. if and only if it is \( G(M_2) \) for some TM \( M_2 \).

**Proof** With Lemma 7.1 we have only to show how an r.e. set \( L = L(M_1) \) can be generated by a TM \( M_2 \). \( M_2 \) simulates the pair generator. When \( (i, j) \) is generated, \( M_2 \) produces the \( i \)th word \( w_i \) in canonical order and simulates \( M_1 \) on \( w_i \) for \( j \) steps. If \( M_1 \) accepts on the \( j \)th step (counting the initial ID as step 1), then \( M_2 \) generates \( w_i \).

Surely \( M_2 \) generates no word not in \( L \). If \( w \) is in \( L \) let \( w \) be the \( i \)th word in canonical order for the alphabet of \( L \), and let \( M_1 \) accept \( w \) after exactly \( j \) moves. As it takes only a finite amount of time for \( M_2 \) to generate any particular word in canonical order or to simulate \( M_1 \) for any particular number of steps, we know that \( M_2 \) will eventually produce the pair \((i, j)\). At that stage, \( w \) will be generated by \( M_2 \). Thus \( G(M_2) = L \).

**Corollary** If \( L \) is an r.e. set, then there is a generator for \( L \) that enumerates each word in \( L \) exactly once.

**Proof** \( M_2 \) described above has that property, since it generates \( w_i \) only when considering the pair \((i, j)\), where \( j \) is exactly the number of steps taken by \( M_1 \) to accept \( w_i \).

**Characterization of recursive sets by generators**

We shall now show that the recursive sets are precisely those sets whose words can be generated in canonical order.
Lemma 7.2 If L is recursive, then there is a generator for L that prints the words of L in canonical order and prints no other words.

Proof Let \( L = L(M_1) \subseteq \Sigma^* \), where \( M_1 \) halts on every input. Construct \( M_2 \) to generate L as follows. \( M_2 \) generates (on a scratch tape) the words in \( \Sigma^* \), one at a time, in canonical order. After generating some word \( w \), \( M_2 \) simulates \( M_1 \) on \( w \). If \( M_1 \) accepts \( w \), \( M_2 \) generates \( w \). Since \( M_1 \) is guaranteed to halt, we know that \( M_2 \) will finish processing each word after a finite time and will therefore eventually consider each particular word in \( \Sigma^* \). Clearly \( M_2 \) generates L in canonical order. \( \square \)

The converse of Lemma 7.2, that if L can be generated in canonical order then L is recursive, is also true. However, there is a subtlety of which we should be aware. In Lemma 7.2 we could actually construct \( M_2 \) from \( M_1 \). However, given a TM \( M \) generating L in canonical order, we know a halting TM recognizing L exists, but there is no algorithm to exhibit that TM.

Suppose \( M_1 \) generates L in canonical order. The natural thing to do is to construct a TM \( M_2 \) that on input \( w \) simulates \( M_1 \) until \( M_1 \) either generates \( w \) or a word beyond \( w \) in canonical order. In the former case, \( M_2 \) accepts \( w \), and in the latter case, \( M_2 \) halts without accepting \( w \). However, if L is finite, \( M_1 \) may never halt after generating the last word in L, so \( M_1 \) may generate neither \( w \) nor any word beyond. In this situation \( M_2 \) would not halt. This problem arises only when L is finite, even though we know every finite set is accepted by a Turing machine that halts on all inputs. Unfortunately, we cannot determine whether a TM generates a finite set or, if finite, which finite set it is. Thus we know that a halting Turing machine accepting L, the language generated by \( M_1 \), always exists, but there is no algorithm to exhibit the Turing machine.

Theorem 7.8 \( L \) is recursive if and only if \( L \) is generated in canonical order.

Proof The “only if” part was established by Lemma 7.2. For the “if” part, when \( L \) is finite, \( M_2 \) described above is a halting Turing machine for \( L \). Clearly, when \( L \) is finite, there is a finite automaton accepting \( L \), and thus \( L \) can be accepted by a TM that halts on all inputs. Note that in general we cannot exhibit a particular halting TM that accepts \( L \), but the theorem merely states that one such TM exists.

7.8 RESTRICTED TURING MACHINES EQUIVALENT TO THE BASIC MODEL

In Section 7.5 we considered generalizations of the basic TM model. As we have seen, these generalizations have no more computational power than the basic model. We conclude this chapter by considering some models that at first appear less powerful than the TM but indeed are just as powerful. For the most part, these models will be variations of the pushdown automaton defined in Chapter 5.

In passing, we note that a pushdown automaton is equivalent to a nondeterministic TM with a read-only input on which the input head cannot move left, plus a storage tape with a rather peculiar restriction on the tape head. Whenever the storage tape head moves left, it must print a blank. Thus the storage tape to the right of the head is always completely blank, and the storage tape is effectively a stack, with the top at the right, rather than the left as in Chapter 5.

Multistack machines

A deterministic two-stack machine is a deterministic Turing machine with a read-only input and two storage tapes. If a head moves left on either tape, a blank is printed on that tape.

Lemma 7.3 An arbitrary single-tape Turing machine can be simulated by a deterministic two-stack machine.

Proof The symbols to the left of the head of the TM being simulated can be stored on one stack, while the symbols on the right of the head can be placed on the other stack. On each stack, symbols closer to the TM’s head are placed closer to the top of the stack than symbols farther from the TM’s head. \( \square \)

Counter machines

We can prove a result stronger than Lemma 7.3. It concerns counter machines, which are off-line Turing machines whose storage tapes are semi-infinite, and whose tape alphabets contain only two symbols, Z and B (blank). Furthermore, the symbol Z, which serves as a bottom of stack marker, appears initially on the cell scanned by the tape head and may never appear on any other cell. An integer i can be stored by moving the tape head i cells to the right of Z. A stored number can be incremented or decremented by moving the tape head left or right. We can test whether a number is zero by checking whether Z is scanned by the head, but we cannot directly test whether two numbers are equal.

An example of a counter machine is shown in Fig. 7.11; \$ and \# are customarily used for end markers on the input. Here Z is the nonblank symbol on each tape. An instantaneous description of a counter machine can be described by the state, the input tape contents, the position of the input head, and the distance of the storage heads from the symbol Z (shown here as \( d_1 \) and \( d_2 \)). We call these distances the counts on the tapes. The counter machine, then, can only really store a count on each tape and tell if that count is zero.

Lemma 7.4 A four-counter machine can simulate an arbitrary Turing machine.

Proof From Lemma 7.3, it suffices to show that two counter tapes can simulate one stack. Let a stack have \( k - 1 \) tape symbols, \( Z_1, Z_2, \ldots, Z_{k-1} \). Then we can