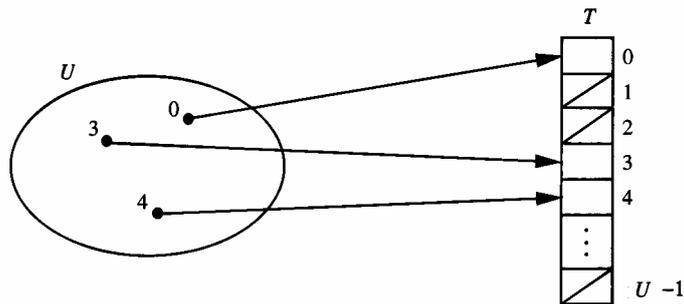


Hashing

Hashing Procedures

Let us denote the set of all possible key values (i.e., the universe of keys) used in a dictionary application by U . Suppose an application requires a dictionary in which elements are assigned keys from the set of small natural numbers. That is, $U \subset \mathbb{Z}^+$ and $|U|$ is relatively small.

If no two elements have the same key, then this dictionary can be implemented by storing its elements in the array $T[0, \dots, |U| - 1]$. This implementation is referred to as a direct-access table since each of the requisite DICTONARY ADT operations - *Search*, *Insert*, and *Delete* - can always be performed in $\mathcal{O}(1)$ time by using a given key value to index directly into T , as shown:

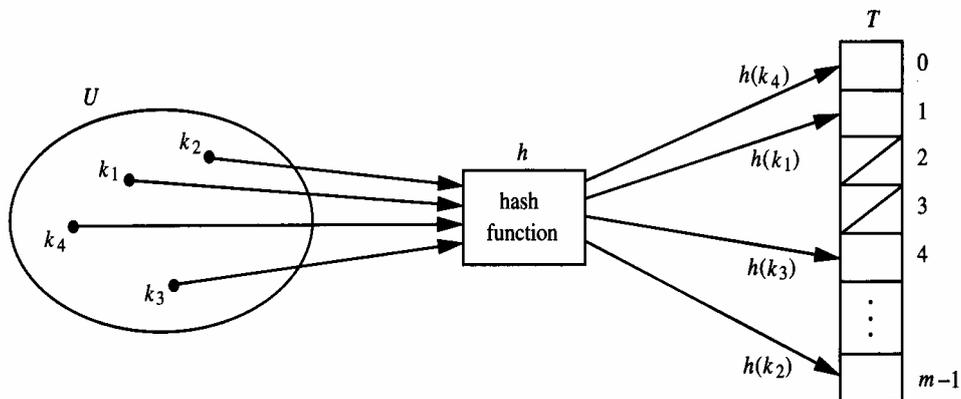


The obvious shortcoming associated with direct-access tables is that the set U rarely has such "nice" properties. In practice, $|U|$ can be quite large. This will lead to wasted memory if the number of elements actually stored in the table is small relative to $|U|$.

Furthermore, it may be difficult to ensure that all keys are unique. Finally, a specific application may require that the key values be real numbers, or some symbols which cannot be used directly to index into the table.

An effective alternative to direct-access tables are hash tables. A hash table is a sequentially mapped data structure that is similar to a direct-access table in that both attempt to make use of the random-access capability afforded by sequential mapping.

However, instead of using a key value to directly index into the hash table, the index is computed from the key value using a hash function, which we will denote using h . This situation is depicted as follows:



In this figure $h(k_i)$ is the index, or hash value, computed by h when it is supplied with key $k_i \in U$. We will say that k_i hashes to slot $T[h(k_i)]$ in hash table T . If we can ensure that all keys have unique hash values, then the DICTONARY ADT operations can be implemented almost in the same way as for regular arrays.

The advantages of this approach are that, if we pick the hash function properly, the size of the hash table m can be chosen so as to be proportional to the number of elements actually stored in the table n , and the key values will not be restricted to the set of small natural numbers.

Furthermore, if the hash function itself can be computed in $\Theta(1)$ time, then each of the DICTONARY ADT operations can be implemented in $\Theta(1)$ time. Of course, this strategy relies on proper selection of the hash function.

An ordinary hash function h performs a mapping from the universe of keys U to slots in the hash table $T[0, \dots, m - 1]$:

$$h : U \rightarrow [0, 1, \dots, m - 1]$$

Since $|U|$ is generally much larger than m , h is unlikely to perform a one-to-one mapping. In other words, it is very probable that:

$$\text{for two keys } k_i \text{ and } k_j, \text{ where } i \neq j, h(k_i) = h(k_j).$$

This situation, where two different keys hash to the same slot, is referred to as a collision. Since two elements cannot be stored in the same slot in a hash table, the *Insert* operation must resolve collisions by relocating an element so that it can be found by subsequent *Search* and *Delete* operations. This will increase the running time of all three operations.

There is an interesting space-time trade-off associated with hash tables. By making the table size m larger, the chances of collisions are generally reduced. However, if m is too large most of the hash table slots will never be utilized.

In general, m should be proportional to n , the number of elements that must be stored in the hash table. If we let α denote the load factor of a hash table (i.e., the ratio of the number of elements currently stored in the table, to the size of the table), then a rule of thumb that works well in practice is:

to choose m so that α never exceeds 0.8 while using the hash table

The development of efficient strategies for resolving collisions is an important issue.

But the issues related to the design of "good" hash functions, and various methods for creating them are important too.

Hash Functions

The most important properties of a good hash function are that it can be computed very quickly (i.e., only a few simple operations are involved), while at the same time minimizing collisions. After all, any hash function that never yields a collision, and whose computation takes $\Theta(1)$ time, can be used to implement all DICTONARY ADT operations in $\Theta(1)$ time.

In order to minimize collisions, a hash function should not be biased towards any particular slot in the hash table. Ideally, a hash function will have the property that each key is equally likely to hash to any of the m slots in the hash table. This behavior is referred to as *simple uniform hashing*, which implies that independently drawing keys from U are uniformly distributed:

$$\sum_{k \in S_j} P(k) = \frac{1}{m} \quad \text{for } j = 0, 1, \dots, m - 1$$

If this condition holds, then the average running time of any DICTIONARY ADT operation is $\Theta(1)$.

The difficulty in designing good hash functions is that we usually do not know the distribution P of values of U .

Let k represents an arbitrary key, m represents the size of the hash table, and n represents the number of elements stored in the hash table. Let us assume that the universe of keys is some subset of the natural numbers. It is typically quite easy to transform values from some other set to natural numbers.

There is a number of specific techniques used to create hash functions. Although a wide variety of hash functions have been suggested, the ones presented next have proved to be most useful in practice.

Division Method

Hash functions that make use of the division method generate hash values by computing the remainder of k divided by m :

$$h(k) = k \bmod m$$

With this hash function, $h(k)$ will always compute a value that is an integer in the range

$$0, 1, \dots, m - 1$$

The choice of m is critical to the performance of the division method. For instance choosing m as a power of 2 is usually ill-advised, since $h(k)$ is simply the p least significant bits of k whenever $m = 2^p$. In this case, the distribution of keys in the hash table is based only on a portion of the information contained in the keys.

For similar reasons, choosing m as a power of 10 should be avoided: when $m = 10^p$, $h(k)$ is simply the last p digits in the decimal representation of k .

In general, the best choices for m when using the division method turn out to be prime numbers that do not divide $r^b \pm a$, where b and a are small natural numbers, and r is the radix of the character set that is used.

As an example of a properly chosen value for m , if we must store $n = 725$ alphabetic strings, and each character is encoded using its ASCII representation, the reasonable table size is $m = 907$, since this is a prime number which is not close to a power of 128, and the load factor will be roughly 0.8 when all strings have been stored.

Multiplication Method

Hash functions that make use of the multiplication method generate hash values in two steps. First the fractional part of the product of k and some real constant A , $0 < A < 1$, is computed. This result is then multiplied by m before applying the floor function to obtain the hash value:

$$h(k) = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$$

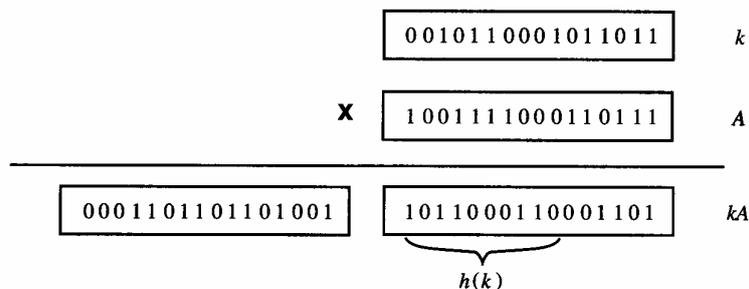
The hash values must be integers in the range $0, 1, \dots, m - 1$. One choice for A that often does a good job of distributing keys throughout the hash table is the inverse of the golden ratio

$$A = 2 / (\sqrt{5} + 1) = 0.61803399\dots$$

The multiplication method exhibits a number of nice mathematical features. Because the hash values depend on all bits of the key, permutations of a key are no more likely to collide than any other pair of keys. Furthermore, keys such as "ptr1" and "ptr2" that are very similar, and therefore have transformed key values that are numerically close to each other, will yield hash values that are widely separated.

A particularly nice property of the multiplication method is that it can be easily approximated using fixed-point arithmetic, exploring fixed-point representation of the numbers as well as a floating-point representation.

The analysis of these representations suggests the following approach for computing hash values using the multiplication method. If b is the number of bits in a machine word, choose the table size to be a power of 2 such that $m = 2^p$, where $p < b$. Represent key values using b -bit integers, and approximate A as a b -bit fixed-point fraction. Perform the fixed-point multiplication kA saving only the low-order b -bit word. The high-order p bits of this word, when interpreted as an integer, is the hash value $h(k)$:



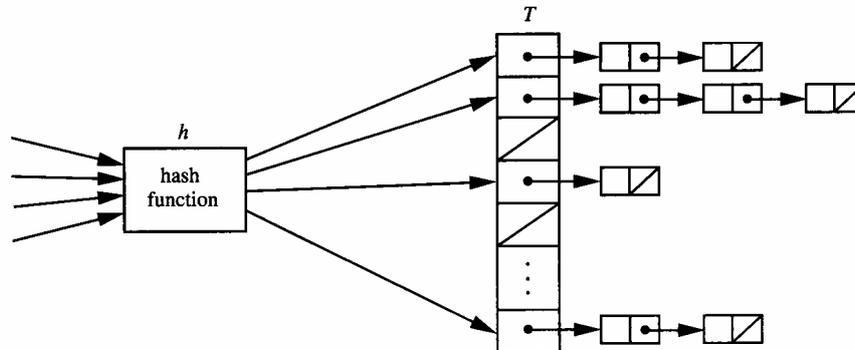
Collision Resolution Strategies

Although hash functions have to minimize collisions, in most applications the collisions will occur. Therefore the manner in which collisions are resolved will directly affect the efficiency of the DICTIONARY ADT operations. It is also important to recognize that a given collision resolution strategy has a more subtle impact on efficiency - if collision resolution is not handled intelligently, it may actually cause additional collisions in the future, thereby impacting the running time of future operations.

There is a number of important collision resolution strategies. The strategies can involve constructing additional data structures for storing the data elements, and then attaching these data structures to the hash table in some fashion.

Separate Chaining

One of the simplest collision resolution strategies, called separate chaining, involves placing all elements that hash to the same slot in a linked list (i.e., a chain). Thus every element stored in a given linked list will have the same key. In this case the slots in the hash table will no longer store data elements, but rather pointers to linked lists:



This strategy is easily extended to allow for any dynamic data structure, not just linked lists. Note that with separate chaining, the number of items that can be stored is only limited by the amount of available memory.

If unordered linked lists are used in this strategy, then the *Insert* operation can be implemented in $\Theta(1)$ time, independent of collisions---each new element is simply added to the head of a specific list. The same cannot be said for the *Search* and *Delete* operations. It is easy to see that in the worst case, both of these operations will take time that is proportional to the length of the longest list.

That is, in the worst case, all n elements hash to the same slot, and the element we are searching for (or deleting) is stored at the tail of this list. This leads to worst-case running times of $\Theta(n)$ for both of these operations. Of course, hash tables should not be selected for a given application based on their worst-case performance.

Open Addressing

In open addressing all data elements are *stored in the hash table itself*. In this case, collisions are resolved by computing a sequence of hash slots. This sequence is successively examined, or probed, until an empty hash table slot is found, in the case of *Insert*, or the desired key is found in the case of *Search* or *Delete*.

The advantage of this approach is that it avoids the use of pointers. The memory saved by not storing pointers can be used to construct a larger hash table if necessary. Thus, using the same amount of memory we can construct a larger hash table, which potentially leads to fewer collisions and therefore faster DICTIONARY ADT operations.

In open addressing, the ordinary hash functions are modified so that they use both a key and a probe number when computing a hash value. This additional information is used to construct the probe sequence. More specifically, in open addressing, hash functions perform the mapping

$$h : U \times [0, 1, \dots, \infty] \rightarrow [0, 1, \dots, m - 1]$$

and produce the probe sequence

$$\langle h(k, 0), h(k, 1), h(k, 2), \dots \rangle$$

Because the hash table contains m slots, there can be at most m unique values in a probe sequence.

Note, however, that for a given probe sequence we are allowing the possibility of $h(k, i) = h(k, j)$ for $i \neq j$. Therefore it is possible for a probe sequence to contain more than m values.

Inserting an element using open addressing involves probing the hash table using the computed probe sequence until an empty array slot is found, or some stopping criteria is met, as shown in the following pseudocode:

```

OpenHash::Insert(array  $T$ , element  $x$ )  ▷ open addressing
1   $i \leftarrow 0$ 
2  do
3       $idx \leftarrow h(\text{key}[x], i)$ 
4       $i \leftarrow i + 1$ 
5       $stop \leftarrow f(i)$   ▷ stopping criteria is some function of  $i$ 
6  while  $T[idx] \neq \text{EMPTY}$  or  $T[idx] \neq \text{DELETED}$  or  $stop = \text{false}$  do
7  if  $stop = \text{true}$  then  ▷ an unsuccessful search
8      return false
9  else  ▷ a successful search
10      $T[idx] \leftarrow x$ 
11     return true

```

Initially all hash table locations store the *empty* value; however, if an element is stored in the table and later deleted, we will mark the vacated slot using the *deleted* symbol rather than the *empty* symbol.

Searching for (or deleting) an element involves probing the hash table until the desired key is found. Note that the same sequence of probes used to insert an element must also be used when searching for (or deleting) it.

The use of *deleted* (rather than *empty*) to mark locations that have had an element deleted increases the efficiency of future *Search* operations. To see why, note that if these locations were instead marked with the *empty* symbol, we would always have to assume that an element had been deleted and continue probing through the entire probe sequence whenever an *empty* was encountered. However, if the *deleted* symbol is used, then a search can terminate whenever an *empty* value is encountered. In this case, we know that the element being searched for is not in the hash table.

Some of the set of specific open addressing strategies:

Linear Probing

This is one of the simplest probing strategies to implement; however, its performance tends to decrease rapidly with increasing load factor.

If the first location probed is j , and c_l is a positive constant, the probe sequence generated by linear probing is

$$\langle j, (j + c_l - 1) \bmod m, (j + c_l - 2) \bmod m, \dots \rangle$$

Given any ordinary hash function h' , a hash function that uses linear probing is easily constructed using

$$h(k, i) = (h'(k) + c_1 i) \bmod m$$

where $i = 0, 1, \dots, m - 1$ is the probe number. Thus the argument supplied to the *mod* operator is a linear function of the probe number.

It should be noted that some choices for c_1 and m work better than others. For example, if we choose m arbitrarily and $c_1 = 1$, then every slot in the hash table can be examined in m probes. However, if we choose m to be an even number and $c_1 = 2$, then only half the slots can be examined by any given probe sequence.

In general, c_1 needs to be chosen so that it is relatively prime to m if all slots in the hash table are to be examined by the probe sequence.

The use of linear probing leads to a problem known as *clustering* – elements tend to clump (or cluster) together in the hash table in such a way that they can only be accessed via a long probe sequence (i.e., after a large number of collisions). This results from the fact that once a small cluster emerges in the hash table, it becomes a "target" for collisions during subsequent insertions.

There are two factors in linear probing that lead to clustering. First, every probe sequence is related to every other probe sequence by a simple cyclic shift, this leads to a specific form of clustering called *primary clustering*: because any two probe sequences are related by a cyclic shift, they will overlap after a sufficient number of probes. Second factor is less severe form of clustering, called *secondary clustering*, results from the fact that if two keys have the same initial hash value $h(k_1, 0) = h(k_2, 0)$, then they will generate the same probe sequence- $h(k_i, i) = h(k_2, i)$ for $i = 1, 2, \dots, m - 1$.

The probe sequence in linear probing is completely determined by the initial hash value, and since there are m of these, the number of unique probe sequences is m . This is far fewer than the $m!$ possible unique probe sequences over m elements. This fact, coupled with the clustering problems, conspire to make linear probing a poor approximation to uniform hashing whenever n approaches m .

Quadratic Probing

This is a simple extension of linear probing in which one of the arguments supplied to the *mod* operation is a quadratic function of the probe number. More specifically, given any ordinary hash function h' , a hash function that uses quadratic probing can be constructed using

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where c_1 and c_2 are positive constants. Once again, the choices for c_1 , c_2 , and m are critical to the performance of this method.

Since the left-hand argument of the mod operation in equation is a non-linear function of the probe number, probe sequences cannot be generated from other probe sequences via simple cyclic shifts. This eliminates the primary clustering problem, and tends to make quadratic probing work better than linear probing.

However, as with linear probing, the initial probe $h(k, 0)$ determines the entire probe sequence, and the number of unique probe sequences is m . Thus, secondary clustering is still a problem, and quadratic probing only offers a good approximation to uniform hashing if m is large relative to n .

Double Hashing

Given two ordinary hash functions h'_1 and h'_2 , double hashing computes a probe sequence using the hash function

$$h(k, i) = (h'_1(k) + i h'_2(k)) \bmod m$$

Note that the initial probe $h(k, 0) = h'_1(k) \bmod m$, and that successive probes are offset from previous probes by the amount $h'_2(k) \bmod m$. Thus the probe sequence depends on k through both h'_1 and h'_2 . This approach alleviates primary and secondary clustering by making the second and subsequent probes in a sequence independent of the initial probe.

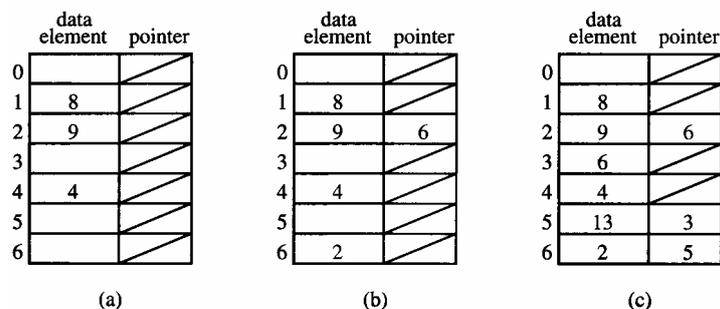
The probe sequences produced by this method have many of the characteristics associated with randomly chosen sequences, which makes the behavior of double hashing a good approximation to uniform hashing.

Coalesced Hashing

This form of collision resolution is similar to the separate chaining approach, except that all data elements are stored in the hash table itself. This is accomplished by allowing each slot in the hash table to store not only a data element, but also a pointer.

These pointers may store either the null value, or the address of some other location within the hash table. Thus, starting from a pointer stored in any non-empty slot, a chain is formed by following this pointer to the slot it points to, reading the pointer contained in the new slot, and continuing in this fashion until a null pointer is reached.

During an insertion, a collision is resolved by inserting the data element into the largest-numbered empty slot in the hash table, and then linking this element to the end of the chain that contains its hash address:



A variation on coalesced hashing sets aside a portion of the hash table, called the cellar, for handling collisions. The portion of the hash table that is not part of the cellar is referred to as the address region. A hash function is selected so that its range is restricted to the address region.

Whenever a collision occurs during an insertion, it is resolved by storing the data element in the next available slot in the cellar. In practice, this approach appears to slightly improve search time; however, the difficulty of determining the appropriate size for the cellar is introduced. Empirical studies have shown that allocating **14 percent** of the hash table to the cellar leads to good performance:

		data element	pointer
address region	0		/
	1	8	/
	2	9	/
	3		/
	4	4	/
	5		/
cellar	6		/
	7		/
	8		/
	9		/

(a)

		data element	pointer
address region	0		/
	1	8	/
	2	9	9
	3		/
	4	4	/
	5		/
cellar	6		/
	7		/
	8		/
	9	2	/

(b)

		data element	pointer
address region	0		/
	1	8	/
	2	9	9
	3		/
	4	4	/
	5		/
cellar	6	13	8
	7		/
	8	6	/
	9	2	/

(c)

Table Overflow

Up to this point, we have assumed the hash table size m will always be large enough to accommodate the data sets we are working with. In practice, however, we must consider the possibility of an insertion into a full table (i.e., table overflow).

If separate chaining is being used, this is typically not a problem since the total size of the chains is only limited by the amount of available memory in the free store. Thus we will restrict our discussion to table overflow in *open address* hashing.

Two techniques that circumvent the problem of table overflow by allocating additional memory will be considered. In both cases, it is best not to wait until the table becomes completely full before allocating more memory; instead, memory will be allocated whenever the load factor α exceeds a certain threshold which we denote by α_t .

Table Expansion

The simplest approach for handling table overflow involves allocating a larger table whenever an insertion causes the load factor to exceed α_t , and then moving the contents of the old table to the new one. The memory of the old table can then be reclaimed.

Using the technique of implementing lists by arrays for hash tables the method can be suggested but it is also complicated by the fact that the output of hash functions is dependent on the table size.

This means that after the table is expanded (or contracted), every data element needs to be "rehashed" into the new table. The additional overhead due to rehashing tends to make this method too slow. An alternative approach is considered next.

Extendible Hashing

Extendible hashing limits the overhead due to rehashing by splitting the hash table into blocks. The hashing process then proceeds in two steps: The low-order bits of a key are first checked to determine which block a data element will be stored in (i.e., all data elements in a given block will have identical low-order bits), and then the data element is actually hashed into a particular slot in that block using the methods discussed.

The addresses of these blocks are stored in a directory table. In addition, a value b is stored with the table - this gives the number of low-order bits to use during the first step of the hashing process.

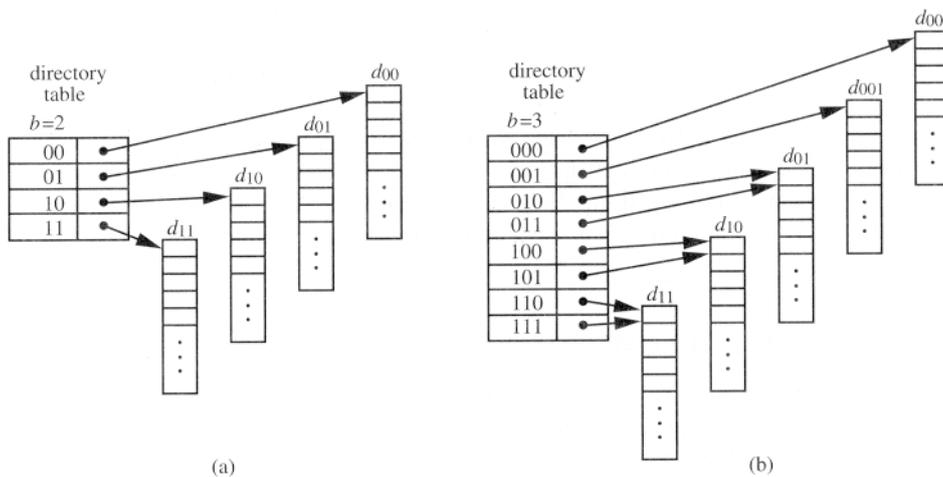
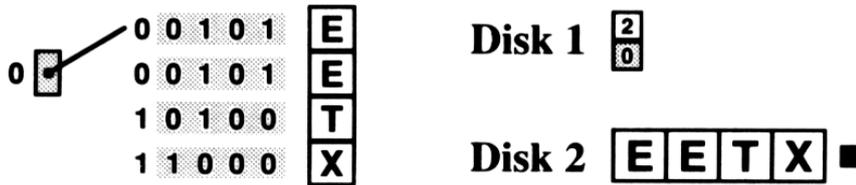


Table overflow can now be handled as follows. Whenever the load factor α , of any one block d is exceeded, an additional block d' the same size as d is created, and the elements originally in d are rehashed into both d and d' using $b + 1$ low-order bits in the first step of the hashing process. Of course, the size of the directory table must be doubled at this point, since the value of b is increased by one. This process is demonstrated in figure above.

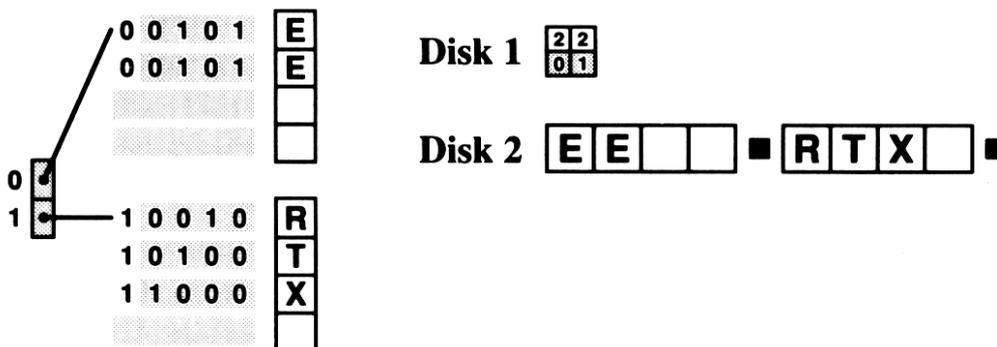
Extendible hashing, if applied to disk memory, gives an alternative to B-trees. This method involves two disk accesses for each search in typical applications while at the same time allowing efficient insertion. The records are stored on pages (clusters) which are split into two pieces when they fill up.

The example, for the string of input data:

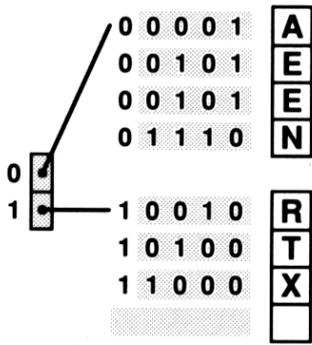
“EXTERNALSEARCHINGEXAMPLE”



The first page



Directory split



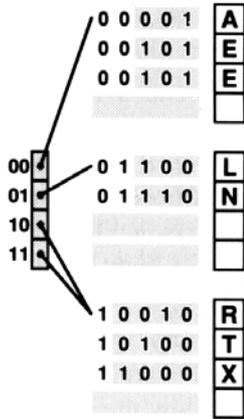
Disk 1

2	2
0	1

Disk 2

A	E	E	N	■	R	T	X	■
---	---	---	---	---	---	---	---	---

First page again full



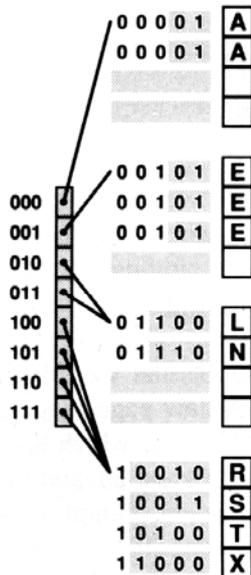
Disk 1

2	2	2	2
0	2	1	1

Disk 2

A	E	E	■	R	T	X	■	L	N	■
---	---	---	---	---	---	---	---	---	---	---

Second split



Disk 1

2	3	2	2	2	2	2	2
0	0	2	2	1	1	1	1

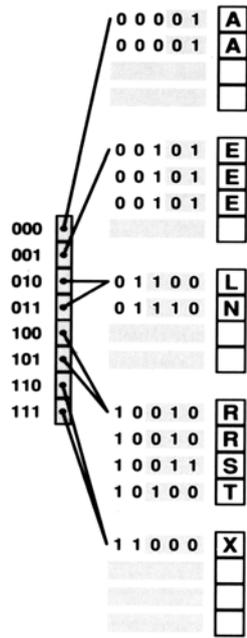
Disk 2

A	A	■	R	S	T	X	■	L	N	■
---	---	---	---	---	---	---	---	---	---	---

Disk 3

E	E	E	■
---	---	---	---

Third split



Disk 1

2	3	2	2	2	2	3	3
0	0	2	2	1	1	1	1

 ■

Disk 2

A	A		
R	R	S	T
L	N		

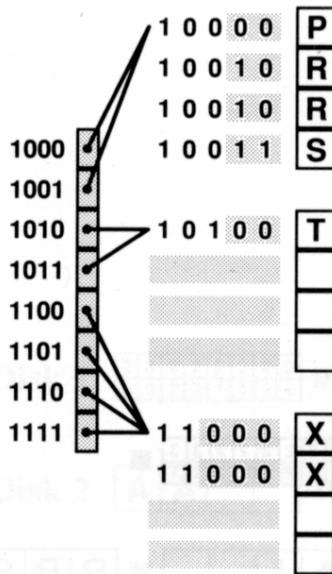
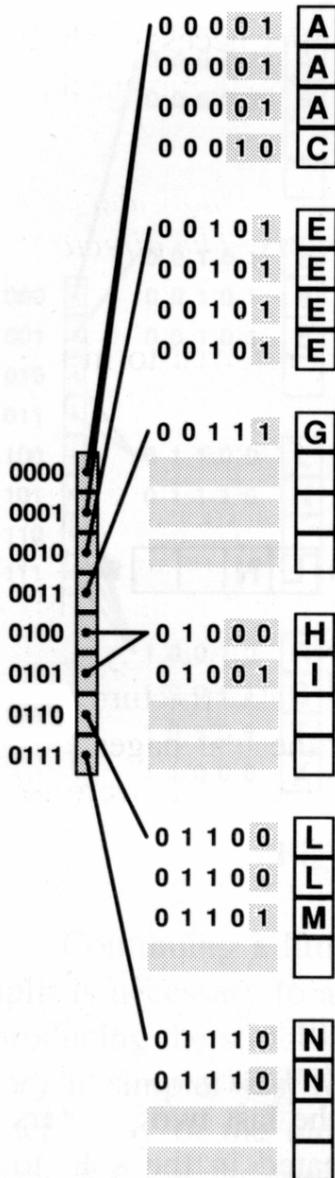
 ■

Disk 3

E	E	E	
X			

 ■

Fourth split



Disk 1

2	2	3	4	2	2	3	4
0	0	0	0	2	2	2	2

 ■

2	2	4	4	3	3	3	3
1	1	1	1	1	1	1	1

 ■

Disk 2

A	A	A	C
P	R	R	S
H	I		

 ■

Disk 3

E	E	E	E
X	X		
L	L	M	

 ■

Disk 4

G			
T			
N	N		

 ■

Extendible hashing access

Property: With pages that can hold M records, extendible hashing may be expected to require about $1.44 (N / M)$ pages for a file of N records. The directory may be expected to have about $N^{1+1/M} / M$ entries.

The analysis of algorithm is complicated and beyond the scope of material.