Hashing

The *DYNAMIC SET* ADT

The ***set*** is a fundamental structure in mathematics.

Computer science view to ***a set***:

- it groups objects together;
- the objects in a set are called the elements or members of the set;
- these elements are taken from the universal set ***U***, which contains all possible set elements;
- all the members of- a given set are unique.

The number of elements contained in a set ***S*** is referred to as the cardinality of ***S***, denoted by **|*S*|**. It is often refered to a set with cardinality ***n*** as an ***n-set***.

The elements of a set are not ordered. Thus, {1, 2, 3} and {3, 2, 1} represent the same set.

Mathematical operations with sets:

- an element ***x*** is (or is not) a member of the set ***S***;
- the empty set;
- two sets ***A*** and ***B*** are equal (or not);
- an ***A*** is said to be a subset of ***B*** (the empty set is a subset of every set);
- the union of ***A*** and **B**;
- the intersection of ***A*** and ***B***;
- the difference of ***A*** and ***B***;
- the Cartesian product of two sets.

In computer science it is often useful to consider set-like structures in which the ordering of the elements is important, such sets will be refered as an ordered n-tuple, like (a_l, a_2, …, a_n).

The concept of a set serves as the basis for a wide variety of useful abstract data types. A large number of computer applications involve the manipulation of sets of data elements. Thus, it makes sense to investigate data structures and algorithms that support efficient implementation of various operations on sets.

Another important difference between the mathematical concept of a set and the sets considered in computer science:

- a set in mathematics is unchanging, while the sets in CS are considered to change over time as data elements are added or deleted.

Thus, sets are refered here as ***dynamic sets***. In addition, we will assume that each element in a dynamic set contains an identifying field called a key, and that ***a total ordering relationship*** exists on these keys.

It will be assumed that no two elements of a dynamic set contain the same key.

The concept of ***a dynamic set*** as an ***DYNAMIC SET*** ADT is to be specified, that is, as a collection of data elements, along with the legal operations defined on these data elements.

If the DYNAMIC SET ADT is implemented properly, application programmers will be able to use dynamic sets without having to understand their implementation details. The use of ADTs in this manner simplifies design and development, and promotes reusability of software components.

A list of general operations for the ***DYNAMIC SET*** ADT. In each of these operations, ***S*** represents a specific dynamic set:

1. ***Search(S, k)***. Returns the element with key ***k*** in ***S***, or the ***null*** value if an element with key ***k*** is not in ***S***.
2. ***Insert(S, x)***. Adds element ***x*** to ***S***. If this operation is successful, the boolean value ***true*** is returned; otherwise, the boolean value ***false*** is returned.
3. ***Delete(S, k)***. Removes the element with key ***k*** in ***S***. If this operation is successful, the boolean value ***true*** is returned; otherwise, the boolean value ***false*** is returned.
4. ***Minimum(S)***. Returns the element in dynamic set ***S*** that has the smallest key value, or the ***null*** value if ***S*** is empty.
5. ***Maximum(S)***. Returns the element in ***S*** that has the largest key value, or the ***null*** value if ***S*** is empty.
6. ***Predecessor(S, k)***. Returns the element in ***S*** that has the largest key value less than ***k***, or the ***null*** value if no such element exists.
7. ***Successor(S, k)***. Returns the element in S that has the smallest key value greater than ***k***, or the ***null*** value if no such element exists.

In addition, when considering the ***DYNAMIC SET*** ADT (or any modifications of this ADT) we will assume the following operations are available:

1. ***Empty(S)***. Returns a boolean value, with ***true*** indicating that ***S*** is an empty dynamic set, and ***false*** indicating that ***S*** is not.
2. ***MakeEmpty(S)***. Clears ***S*** of all elements, causing ***S*** to become an empty dynamic set.

Since these last two operations are often trivial to implement, they generally are to be omitted.

In many instances an application will only require the use of ***a few DYNAMIC SET*** operations. Some groups of these operations are used so frequently that they are given special names:

- the ADT that supports ***Search, Insert,*** and ***Delete*** operations is called the ***DICTIONARY*** ADT;
- the ***STACK***, ***QUEUE***, and ***PRIORITY QUEUE*** ADTs are all special types of dynamic sets.

A variety of data structures will be described in forthcoming considerations that they can be used to implement either the ***DYNAMIC SET*** ADT, or ADTs that support specific subsets of the ***DYNAMIC SET*** ADT operations.

Each of the data structures described can be analyzed in order to determine ***how efficiently*** they support the implementation of these operations. In each case, the analysis is to be performed in terms of ***n***, the number of data elements stored in the dynamic set.

This analysis will demonstrate that there is ***no optimal data structure*** for implementing dynamic sets.

Rather, the best implementation choice will depend upon:

- which operations need to be supported,
- the frequency with which specific operations are used,
- and possibly many other factors.

As always, ***the more we know*** about how a specific application will use data, the better we can ***fine tune*** the associated data structures so that this data can be accessed efficiently.

Dictionary ADT

The ***Dictionary ADT*** is the Dynamic set ADT with operations: ***search, insert, delete***.

The classical problem with ***dictionary*** is of automatically creating a rough index for an electronic document. Given an input file in ASCII format, there is a need to produce an output file that contains an alphabetical list of keywords, along with the page numbers on which they appear in the input document.

One way of producing an index would be to read through the entire document, storing each ***novel*** word that is encountered in a list. Next a user would remove from the list those words that should not appear in the index. Finally the document would be read again, noting the pages on which each word in the list appears. This process requires ***two passes*** through (i.e., readings of) the input document.

It is possible to construct an index in ***a single pass*** by keeping track of the page numbers on which words occur as they are encountered. The problem with this approach is that the user is ***not given*** the opportunity to ***cull*** words from the list, which means that commonly occurring words (e.g., “and”, “the”, “and”, “it”, …, etc.) would appear in the index.

To solve this problem it is necessary to augment one-pass method with a dictionary containing words that should not appear in the index (this dictionary is “LeaveOut”). The question then becomes:

- what words should appear in LeaveOut?

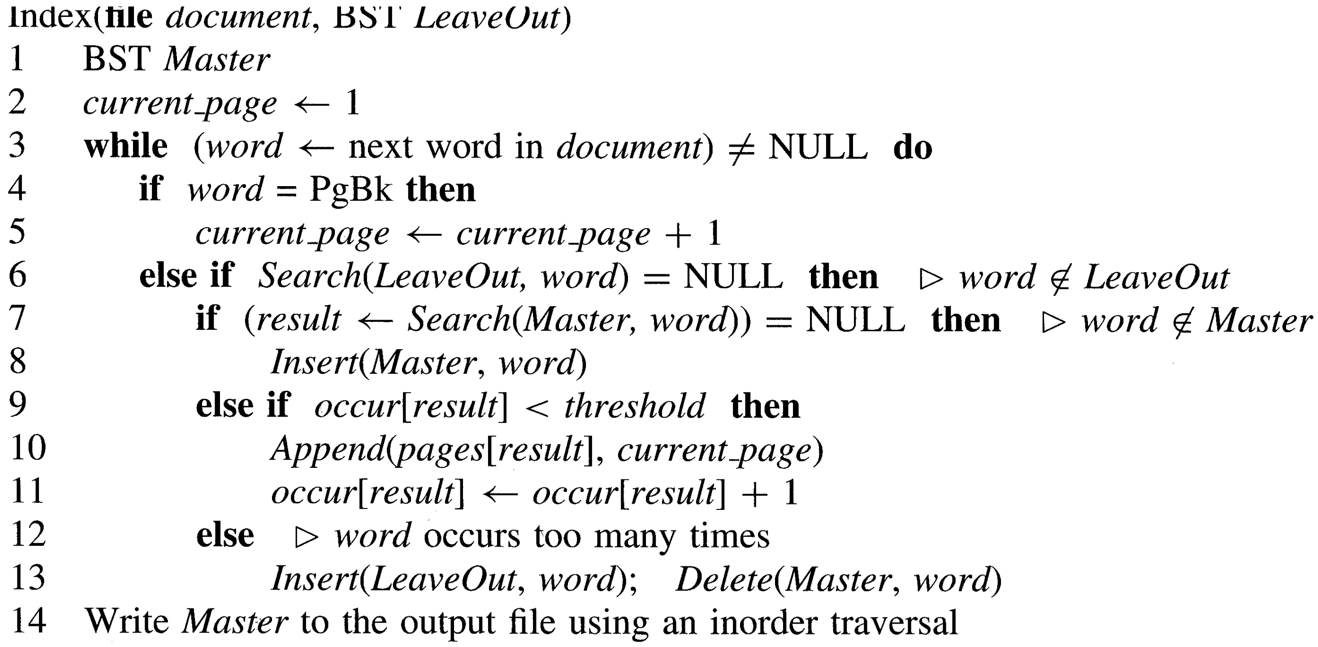
Obviously, words such as ***“and”, “the”, “and”, “it”***, should appear in ***LeaveOut***, and the user can add many other common words to this dictionary. However, certain words such as "tree" and "algorithm", may occur frequently in some texts, and may occur very infrequently in other documents.

Let us assume that the words are stored, which have to be included into the index, in a dictionary called ***Master***, and that along with a word, the list of page numbers on which it occurs, is stored, and also the number of times they occur. The modified one-pass approach then works as follows:

- each time a word is read from the input document, it is checked to see if it appears in ***LeaveOut***;
- if it does, then ignore this word and read the next one;
- if it does not, then it is first to be searched in ***Master*** to see if it contains this word;
- if the word is not present, it is added to ***Master***, along with the page number on which it appeared, and then it is initialized its number of occurrences to one;
- if the word it is searched for, is found in ***Master***, then simply the list of page numbers on which the word occurs, is apdated, and the number of occurrences in incremented by one;
- after this it is checked to see if the number of occurrences of the word has exceeded the threshold;
- if it has, then the word is deleted from ***Master*** and is added to ***LeaveOut***.

The method just described entails quite a bit of time searching the ***LeaveOut*** and ***Master*** dictionaries. For this reason, these dictionaries can be implemented using binary search trees, which should perform much better than lists in this application.

Pseudocode for one-pass index creation method is given next. In this pseudocode it is assumed that ***Master*** and ***LeaveOut*** store compound data objects consisting of a key (which is a function of a word), a list called pages, which stores page numbers, and a variable occur, which keeps track of the number of occurrences of a given word. This pseudocode also assumes that a special word, ***PgBk***, will be used in the input document to indicate a page break.



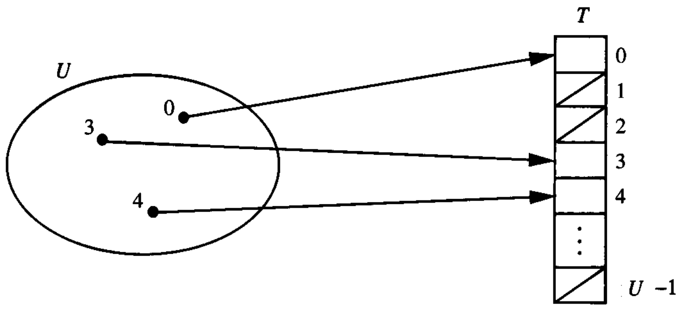
```
Index(file document, BST LeaveOut)
1   BST Master
2   current_page $\leftarrow$ 1
3   while (word $\leftarrow$ next word in document) $\neq$ NULL do
4       if word = PgBk then
5           current_page $\leftarrow$ current_page + 1
6       else if Search(LeaveOut, word) = NULL then    $\triangleright$ word $\notin$ LeaveOut
7           if (result $\leftarrow$ Search(Master, word)) = NULL then    $\triangleright$ word $\notin$ Master
8               Insert(Master, word)
9           else if occur[result] < threshold then
10              Append(pages[result], current_page)
11              occur[result] $\leftarrow$ occur[result] + 1
12          else    $\triangleright$ word occurs too many times
13              Insert(LeaveOut, word);   Delete(Master, word)
14  Write Master to the output file using an inorder traversal
```

The loop spanning lines 3-14 of Index() performs a single pass through the input document, implementing the steps just described. Specifically, line 3 reads the words in the input document sequentially until it reaches the end of the file (i.e., encounters a null value). If the current word is not in ***LeaveOut***, then ***Master*** is searched on line 7. If the result of this search is the null value (which means the word is not currently in ***Master***) then the word is inserted in ***Master***. If the search on line 7 is successful and the number of occurrences of the word does not exceed the threshold, then the list of page numbers is updated on line 10, and the number of occurrences is incremented on line 11. If, on the other hand, the number of occurrences of the current word exceeds the threshold, then this word is inserted into ***LeaveOut***, and deleted from ***Master*** on line 12. After all words in the input document have been processed, the ***Master*** dictionary is written to an output file using an inorder traversal. If the key values for words will be choosen appropriately, this will output the words in alphabetical order.

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 Ì 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, … , |U| - 1]***. This implementation is referred to as a direct-access table since each of the requisite DICTIONARY ADT operations - ***Search, Insert,*** and ***Delete*** - can always be performed in ***Q(1)*** time by using a given key value to index directly into ***T***, as shown:

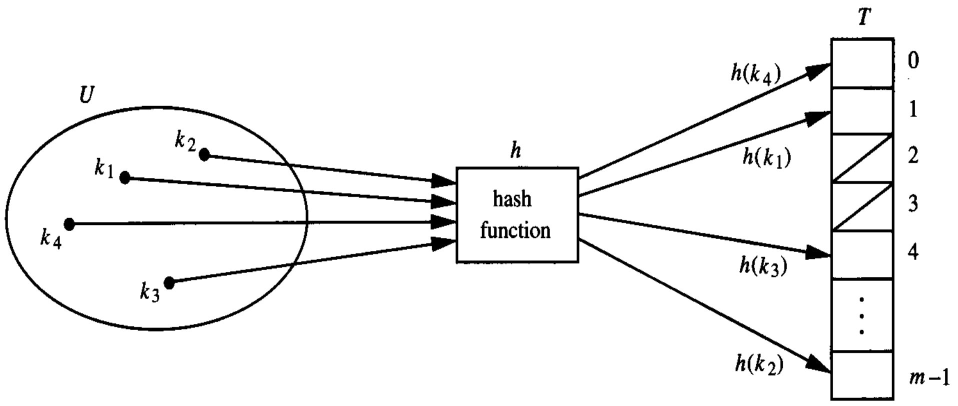


The obvious shortcoming associated with direct-access tables is that the set ***U*** rarely has such "nice" properties. In practice, $|\boldsymbol{U}|$ can be quite large. This will lead to wasted memory if the number of elements actually stored in the table is small relative to $|\boldsymbol{U}|$.

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 $\boldsymbol{h(k_i)}$ is the index, or hash value, computed by ***h*** when it is supplied with key $\boldsymbol{k_i}$ ***Î*** ***U***. We will say that $\boldsymbol{k_i}$ hashes to slot $\boldsymbol{T[h(k_i)]}$ in hash table ***T***. If we can ensure that all keys have unique hash values, then the DICTIONARY 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 ***Q(1)*** time, then each of the DICTIONARY ADT operations can be implemented in ***Q(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, …, m – 1]***:

h : U ® [0, 1, ..., 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:

for two keys k_i and k_j, where ***i ¹ 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 ***a*** 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 Θ(1) time, can be used to implement all DICTIONARY ADT operations in Θ(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:

$$\sum_{k \in s_j} P(k) = \frac{1}{m} \qquad \text{for } j = 0, 1, \ldots, m-1$$

If this condition holds, then the average running time of any DICTIONARY ADT operation is ***Q(l)***.

The difficulty in designing good hash functions is that we usually do not know the distribufion 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 mod m

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

0, 1, ..., 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) = ëm (k A - ëk Aû)û

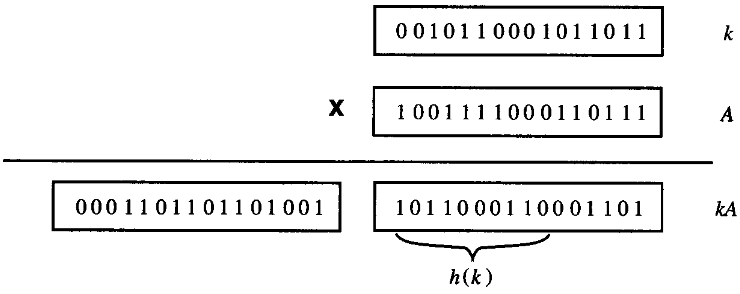
The hash values must be integers in the range $0, 1, . . . , m - 1$. One choice for ***A*** that often does a good job of distributing keys throughout the hash table is the inverse of the golden ratio

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

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 "ptrl" 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)***:



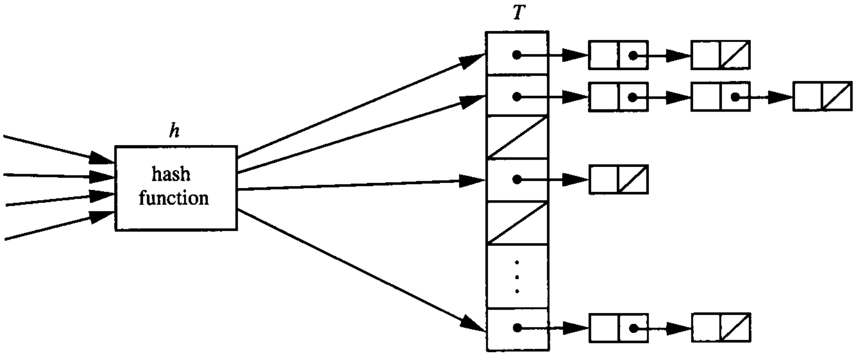
Collision Resolution Strategies

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 ***Q(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 ***Q(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

$$h : \mathrm{U} \times [0, 1, ..., \infty] \rightarrow [0, 1, ..., \mathrm{m} - 1]$$

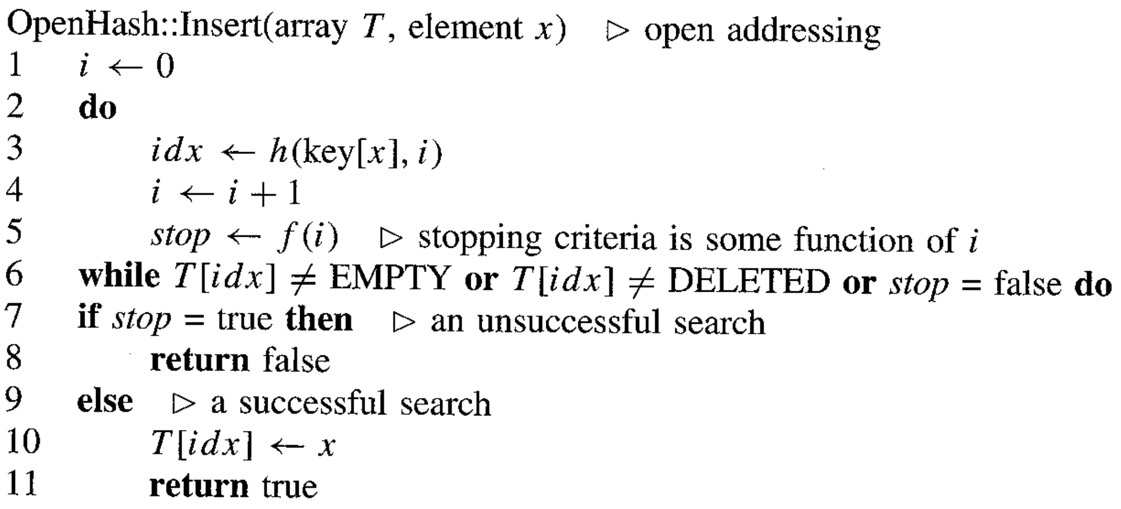
and produce the probe sequence

$$< h(k, 0), h(k, 1), h(k, 2).... >$$

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 ¹ j***. Tberefore 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
    $i \leftarrow 0$
    do
        $idx \leftarrow h(\text{key}[x], i)$
        $i \leftarrow i + 1$
        $stop \leftarrow f(i)$   ▷ stopping criteria is some function of $i$
    while $T[idx] \neq$ EMPTY or $T[idx] \neq$ DELETED or $stop$ = false do
    if $stop$ = true then   ▷ an unsuccessful search
        return false
    else   ▷ a successful search
        $T[idx] \leftarrow x$
        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

$$< j, (j + c_l - 1) \bmod m, (j + c_l - 2) \bmod m, \ldots >$$

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

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

where $i = 0, 1, \ldots, m - 1$ 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_l and m work better than others. For example, if we choose m arbitrarily and $c_l = 1$, then every slot in the hash table can be examined in m probes. However, if we choose m to be an even number and $c_l = 2$, then only half the slots can be examined by any given probe sequence.

In general, c_l 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 $i = 1, 2, . . ., m - 1$.

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

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

where c_l and c_2 are positive constants. Once again, the choices for c_l, 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

$$h\ (k, i) = (h'_1\ (k) + i\ h'_2\ (k))\ \mathrm{mod}\ m$$

Note that the initial probe $h\ (k, 0) = h'_1\ (k)\ \mathrm{mod}\ m$, and that successive probes are offset from previous probes by the amount $h'_2(k)\ \mathrm{mod}\ m$. 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.

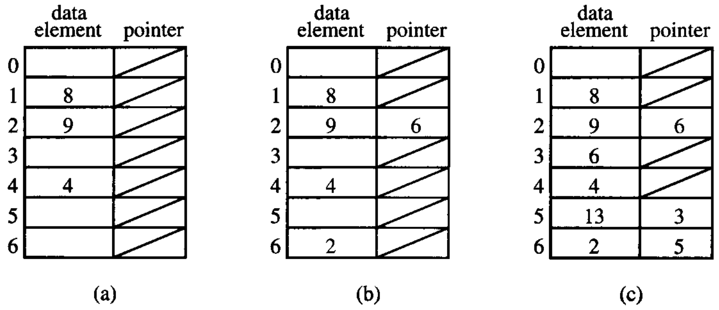
The probe sequences produced by this method have many of the characteristics associated with randomly chosen sequences, which makes the behavior of double hashing a good approximation to uniform hashing.

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.

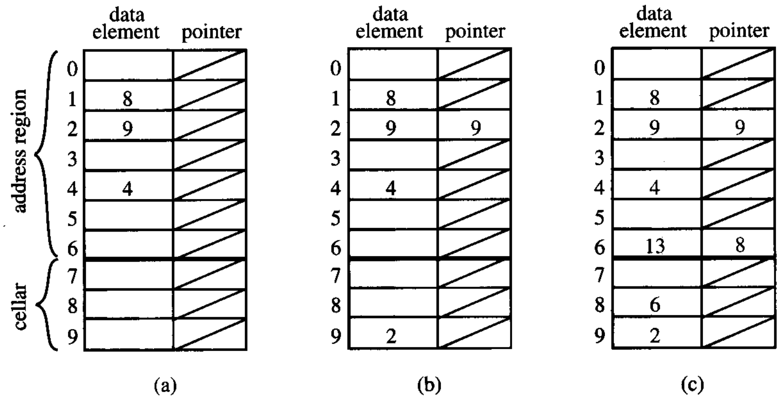
During an insertion, a collision is resolved by inserting the data element into the largest-numbered empty slot in the hash table, and then linking this element to the end of the chain that contains its hash address:



(a) (b) (c)

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:



(a) (b) (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 a exceeds a certain threshold which we denote by at.

Table Expansion

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

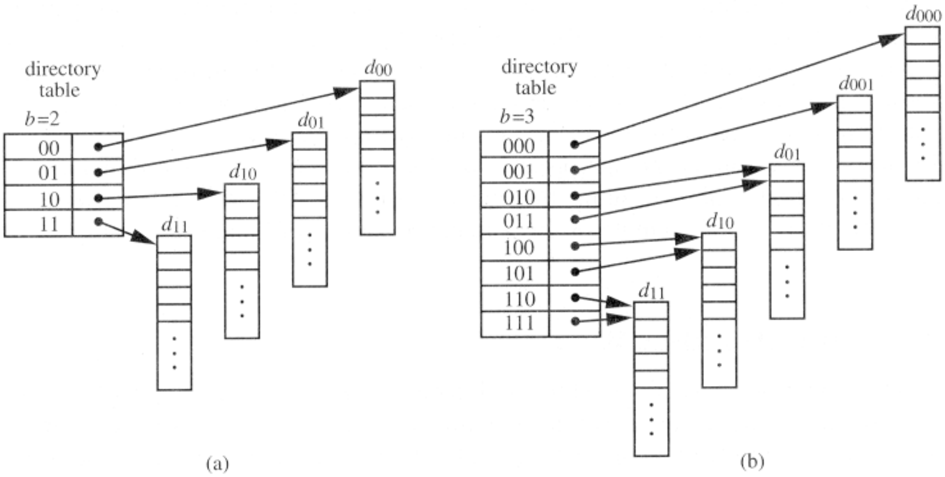
Using the technique of implementing lists by arrays for hash tables the method can be suggested but it is also complicated by the fact that the output of hash functions is dependent on the table size.

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.



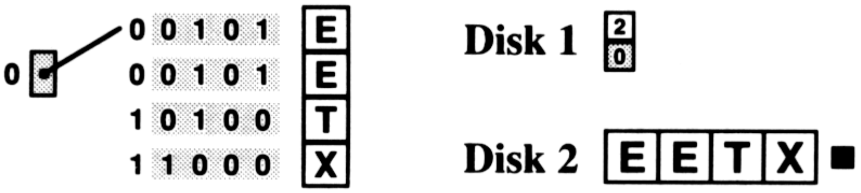
(a) (b)

Table overflow can now be handled as follows. Whenever the load factor ***a***, 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.

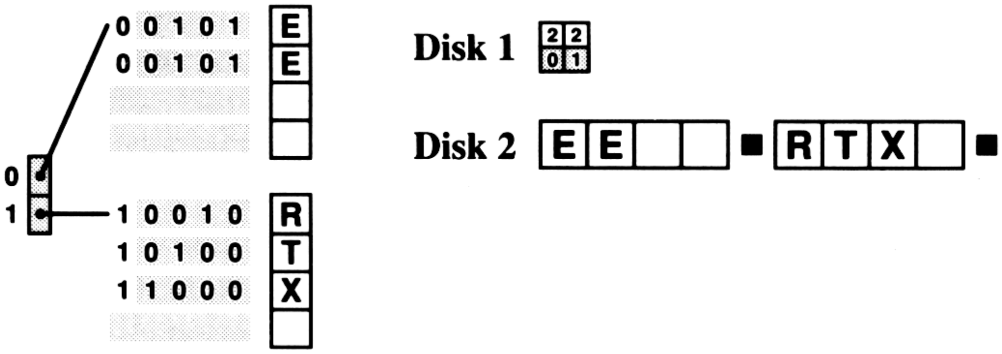
Extendible hashing, if applied to disk memory, gives an alternative to B-trees. This method involves two disk accesses for each search in typical applications while at the same time allowing efficient insertion. The records are stored on pages (clusters) which are split into two pieces when they fill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

The example, for the string of input data:

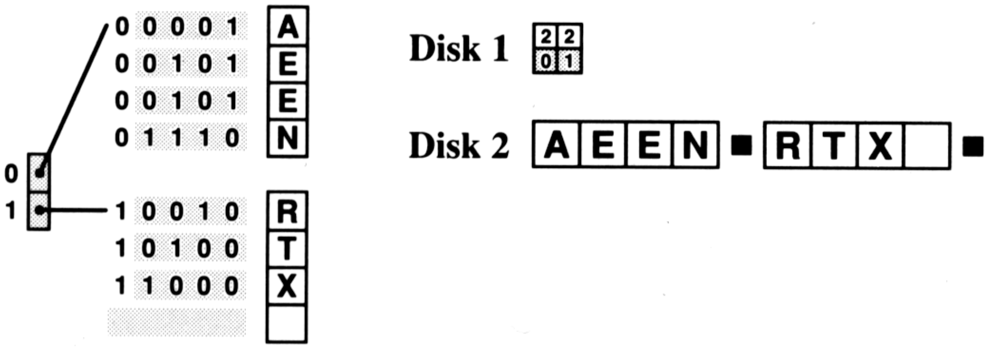
"E X T E R N A L S E A R C H I N G E X A M P L E"



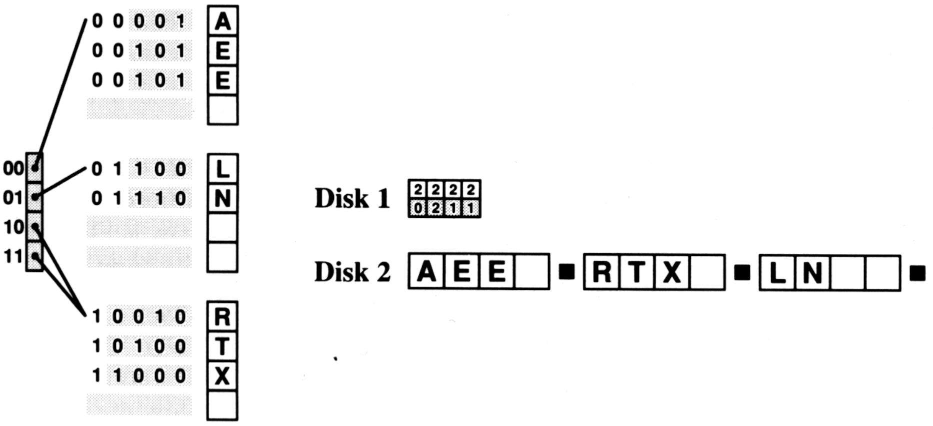
The first page



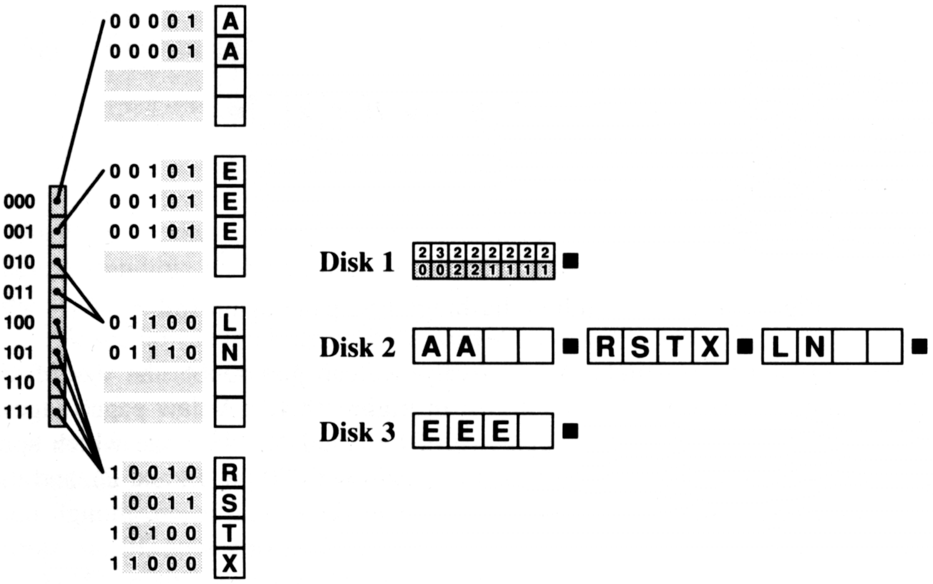
Directory split



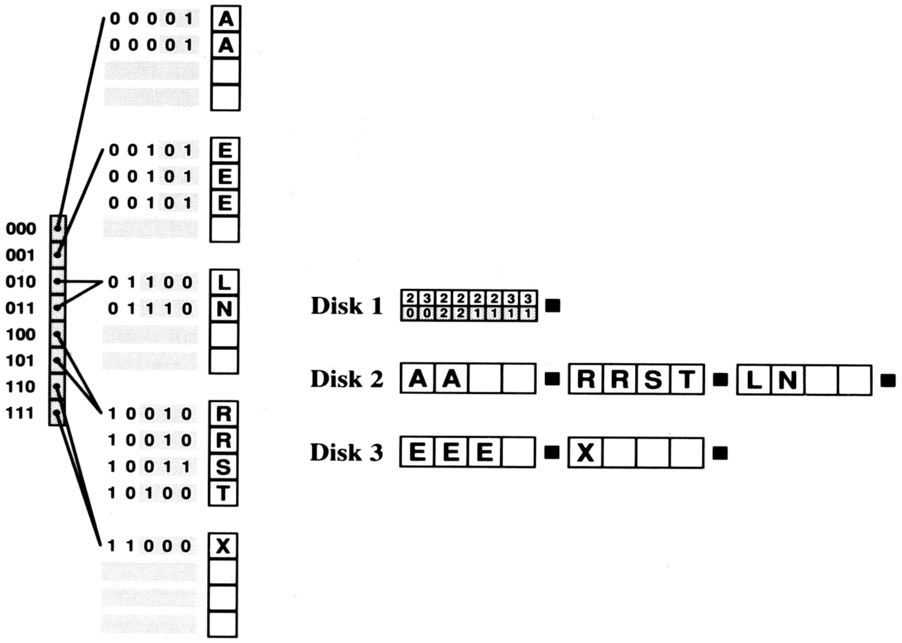
First page again full



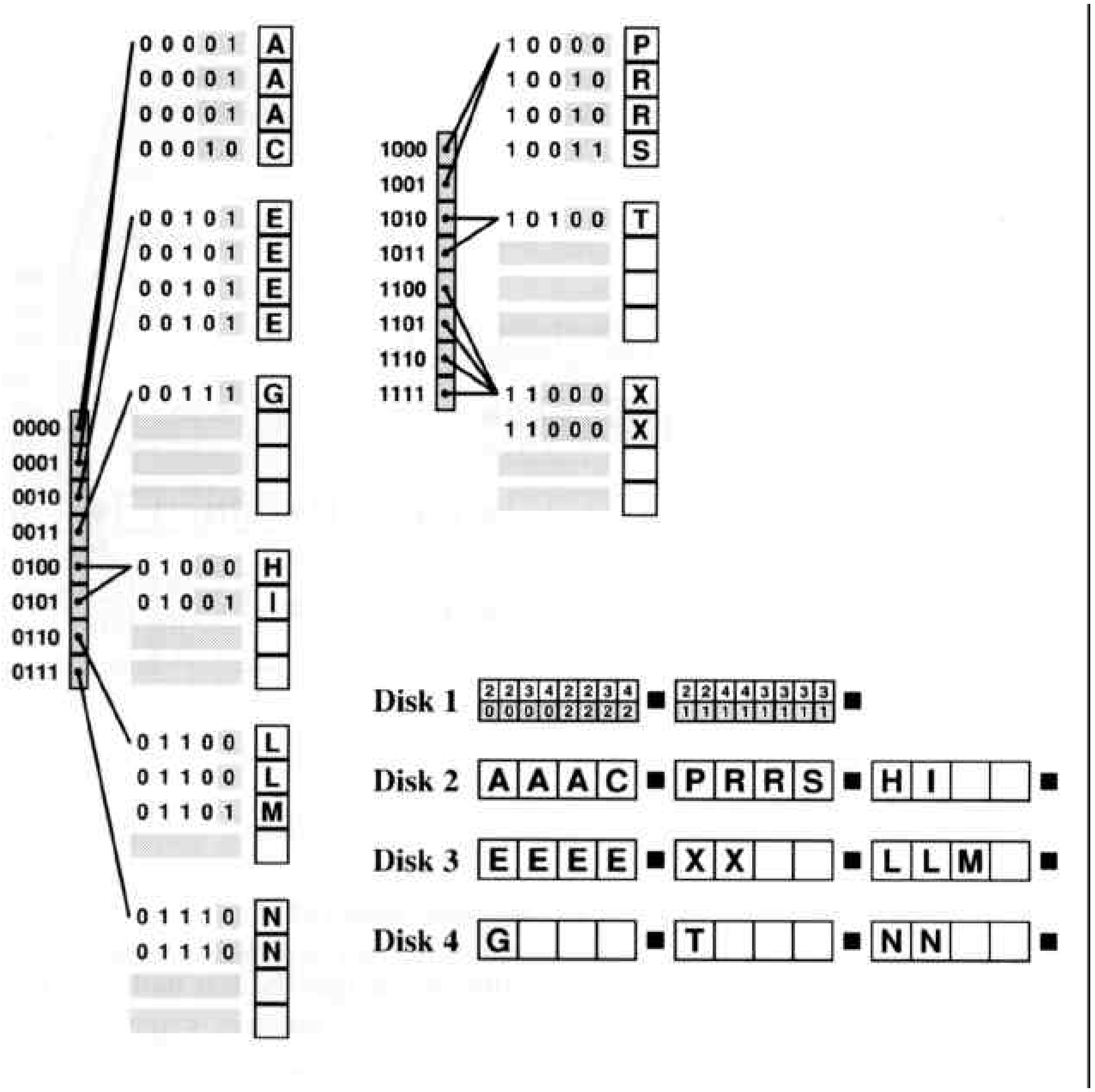
Second split



Third split



Fourth split



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.

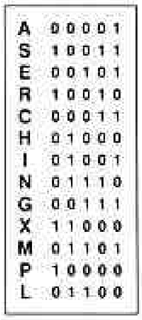
The analysis of algorithm is complicated and beyond the scope of material.

Radix Searching

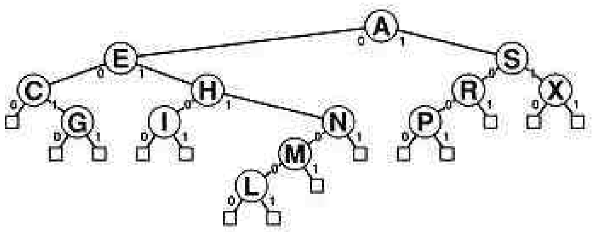
The most simple radix search method is digital tree searching - the binary search tree with the branch in the tree according to the bits of keys: at the first level the leading bit is used, at the second level the second leading bit, and so on until an external node is encountered. The code of the algorithm is similar to binary search tree. The data structures for the program. i.e. how to initialize the memory, are the same as those that for elementary binary search trees. The constant *maxb* is the number of bits in the keys to be sorted:

```
function digitalsearch(u: integer; x: link): link;
var b: integer;
begin
z|.key:=u; b:=maxb;
repeat
if bits (v, b, 1)=0 then x:=x|.l else x:=x|.r;
b:= b-1;
until v=x|.key;
digitalsearch:=x
end;
```

Equal keys are anathema in radix sorting, and the same is true in radix searching. Thus, all the keys to appear in the data structure are distinct: if necessary, a linked list could be maintained for each key value of the records whose keys have that value. It is assumed that the ith letter of the alphabet is represented by the five-bit binary representation of *i*.:



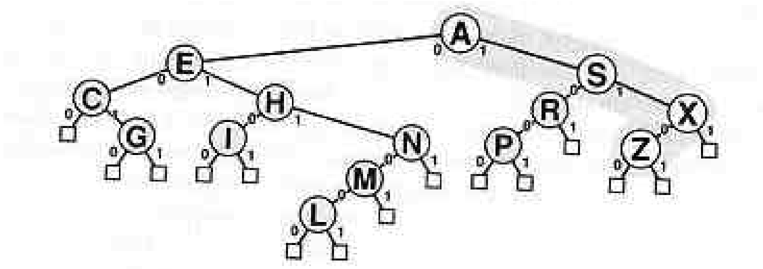
A	0 0 0 0 1
S	1 0 0 1 1
E	0 0 1 0 1
R	1 0 0 1 0
C	0 0 0 1 1
H	0 1 0 0 0
I	0 1 0 0 1
N	0 1 1 1 0
G	0 0 1 1 1
X	1 1 0 0 0
M	0 1 1 0 1
P	1 0 0 0 0
L	0 1 1 0 0



The insert procedure for digital search trees also derives directly from the corresponding procedure for binary search trees:

```
function digitalinsert(v: integer; x: link): link;
var p: link; b: integer;
begin
b:=maxb;
repeat
p:=x;
if bits (v., b. 1)=0 then x:=x|.1 else x:=x|.r;
b:=b-1;
until x=z;
new(x); x|.key:=u; x|.l:=z; x|.r:=z;
if bits(u.b+l,l)=0 then p|.l:=x else p|.r:=x;
digitalinsert:=x
end;
```

Figure below shows what happens when a new key *Z= 11010* is added to the tree:



The worst case for trees built with digital searching is much better than for binary search trees, if the number of keys is large and the keys are not long. The length of the longest path in a digital search tree is the length of the longest match in the leading bits between any two keys in the tree, and this is likely to be relatively small for many applications (for example, if the keys are comprised of random bits).

Property: A *search or insertion in a digital search tree requires about lgN comparisons on the average and b comparisons in the worst case in a tree built from N random b-bit keys.*

It is obvious that no path will ever be any longer than the number of bits in the keys: for example, a digital search tree built from eight-character keys with, say, six bits per character will have no path longer than 48, even if there are hundreds of thousands of keys.

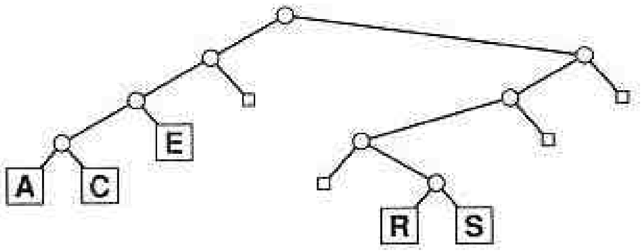
Radix Search Tries

It is quite often the case that search keys are very long, consisting of many characters. In such a situation, the cost of comparing a search key for equality with a key from the data structure can be a dominant cost which cannot be neglected. Digital tree searching uses such a comparison at each tree node; and it is possible in most cases to get by with only one comparison per search.

The idea is to not store keys in tree nodes at all, but rather to put all the keys in external nodes of the tree. Thus, we have two types of nodes:

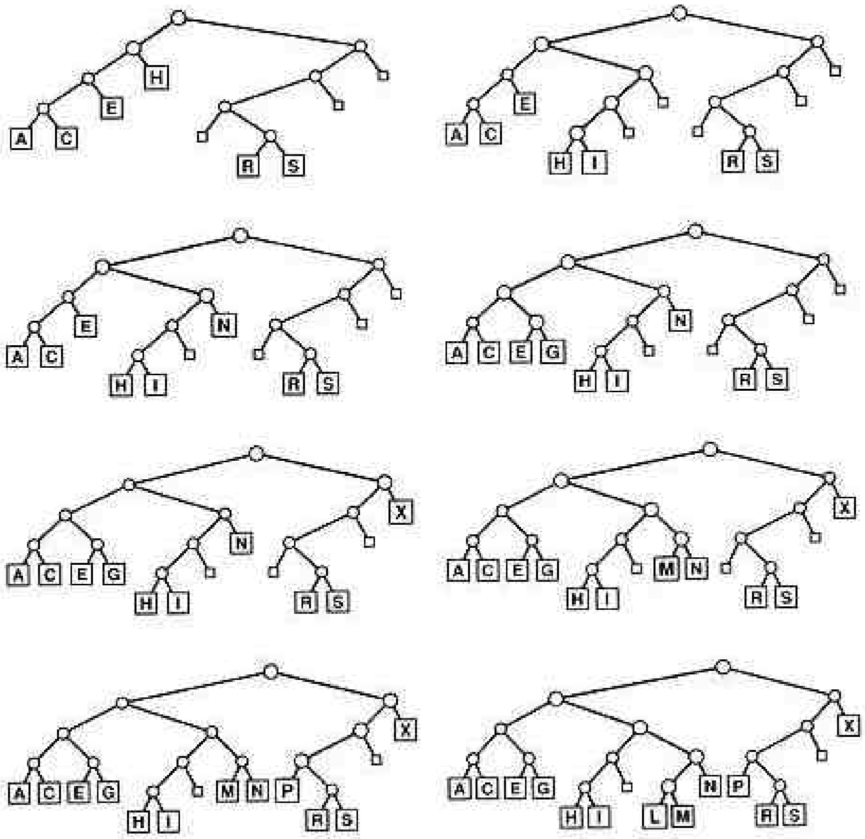
- internal nodes, which just contain links to other nodes,
- external nodes, which contain keys and no links.

To search for a key in such a structure, we just branch according to its bits, as above, but we don't compare it to anything until we get to an external node. Each key in the tree is stored in an external node on the path described by the leading bit pattern of the key and each search key winds up at one external node, so one full key comparison completes the search:



For example, to reach E in the figure, we go left, left, right from the root, since the first three bits of E are 001; but none of the keys in the trie begin with the bits 101, because an external node is encountered if one goes right, left, right. Before thinking about insertion, the reader should ponder the rather surprising property that the trie structure is independent of the order in which the keys are inserted: there is a unique trie for any given set of distinct keys.

As usual, after an unsuccessful search, we can insert the key sought by replacing the external node which terminated the search, *provided* it doesn't contain a key. If the external node which terminates the search does contain a key, then it must be replaced by an internal node which will have the key sought and the key which terminated the search in external nodes below it. Unfortunately, if these keys agree in more bit positions, it is necessary to add some external nodes which correspond to no keys in the tree (or put another way, some internal nodes with an empty external node as a child):



Implementing this method in Pascal is actually relatively complicated because of the necessity to maintain two types of nodes, both of which could be pointed to by links in internal nodes. This is an example of an algorithm for which a low-level implementation might be simpler than a high-level implementation. The left subtree of a binary radix search trie has all the keys which have 0 for the leading bit; the right subtree has all the keys which have I for the leading bit. This leads to an immediate correspondence with radix sorting: binary trie searching partitions the file in exactly the same way as radix exchange sorting.

Property: *A search or insertion in a radix search trie requires about **lgN** bit comparisons for an average search and **b** bit comparisons in the worst case in a tree built from **N** random **b**-bit keys.*

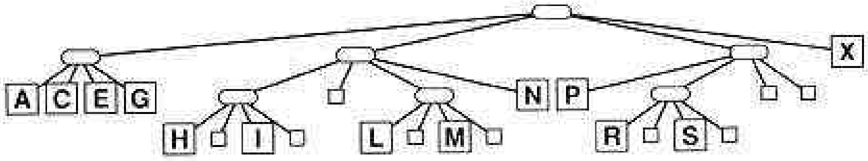
An annoying feature of radix tries, and one which distinguishes them from the other types of search trees we've seen, is the "one-way" branching required for keys with a large number of bits in common. For example, keys which differ only in the last bit require a path whose length is equal to the key length, no matter how many keys there are in the tree. The number of internal nodes can be somewhat larger than the number of keys.

Property: *A radix search trie built from **N** random **b** -bit keys has about N/ln2 **1.44N** nodes on the average.*

The height of tries is still limited by the number of bits in the keys, but we would like to consider the possibility of processing records with very long keys (say 1000 bits or more) which perhaps have some uniformity, as might arise in encoded character data. One way to shorten the paths in the trees is to use many more than two links per node (though this exacerbates the "space" problem of using too many nodes); another way is to "collapse" paths containing one-way branches into single links.

Multiway Radix Searching

For radix sorting, we could get a significant improvement in speed by considering more than one bit at a time. The same is true for radix searching: by examining m bits at a time, we can speed up the search by a factor of 2^m. However, the problem is that considering m bits at a time corresponds to using tree nodes with M = 2^m links, which can lead to a considerable amount of wasted space for unused links:

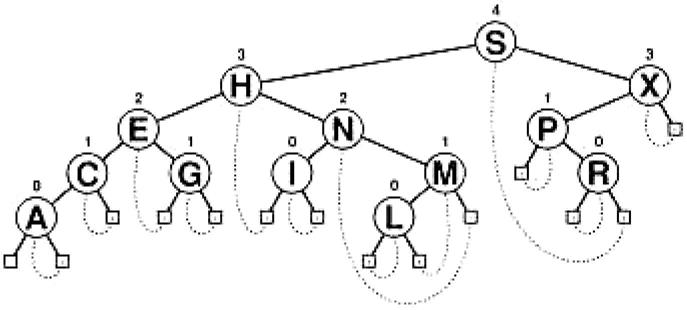


Note that there is some wasted space in this tree because of the large number of unused external links. As ***M*** gets larger, this effect gets worse: it turns out that the number of links used is about ***MN/lnM*** for random keys. On the other hand, this is a very efficient searching method: the running time is about logm *N.* A reasonable compromise can be struck between the time efficiency of multiway tries and the space efficiency of other methods by using a "hybrid" method with a large value of *M* at the top (say the first two levels) and a small value of *M* (or some elementary method) at the bottom. Again, efficient implementations of such methods can be quite complicated, however, because of multiple node types.

Patricia

The radix trie searching method as outlined above has two annoying flaws: the "one-way branching" leads to the creation of extra nodes in the tree, and there are two different types of nodes in the tree, which complicates the code somewhat (especially the insertion code). D. R. Morrison discovered a way to avoid both of these problems in a method which he named *Patricia* ("Practical Algorithm To Retrieve Information Coded In Alphanumeric"). In the present context, *Patricia* allows searching for ***N*** arbitrarily long keys in a tree with just ***N*** nodes, but requires only one full key comparison per search.

One-way branching is avoided by a simple device: each node contains the index of the bit to be tested to decide which path to take out of that node. External nodes are avoided by replacing links to external nodes with links that point upwards in the tree, back to our normal type of tree node with a key and two links. But in Patricia, the keys in the nodes are not used on the way down the tree to control the search; they are merely stored there for reference when the bottom of the tree is reached:



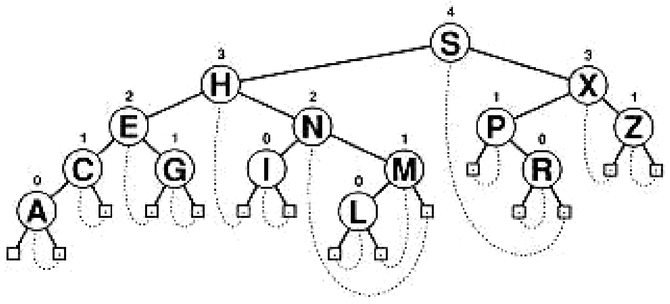
To search in this tree, we start at the root and proceed down the tree, using the bit index in each node to tell us which bit to examine in the search key-we go right if that bit is 1, left if it is 0. The keys in the nodes are not examined at all on the way down the tree. Eventually, an upwards link is encountered: each upward link points to the unique key in the tree that has the bits that would cause a search to take that link. For example, S is the only key in the tree that matches the bit pattern 10*11. Thus if the key at the node pointed to by the first upward link encountered is equal to the search key, then the search is successful; otherwise it is unsuccessful. For tries, all searches terminate at external nodes, whereupon one full key comparison is done to determine whether or not the search was successful; for Patricia all searches terminate at upwards links, whereupon one full key comparison is done to determine whether or not the search was successful. Furthermore, it's easy to test whether a link points up, because the bit indices in the nodes (by definition) decrease as we travel down the tree.

This leads to the following search code for Patricia, which is as simple as the code for radix tree or trie searching:

```
 type link= | node;
     node= record key, info. b: integer; l, r: link end;
     var head.z: link;
 function patriciasearch (v: integer; x: link): link;
   var p: link;
   begin
   repeat
      p:=x;
    if bits (v , x|.b, 1)=0 then x:=x|.l else x:=x|.r;
    until p|.b < =x|.b;
   patriciasearch:=x
   end;
```

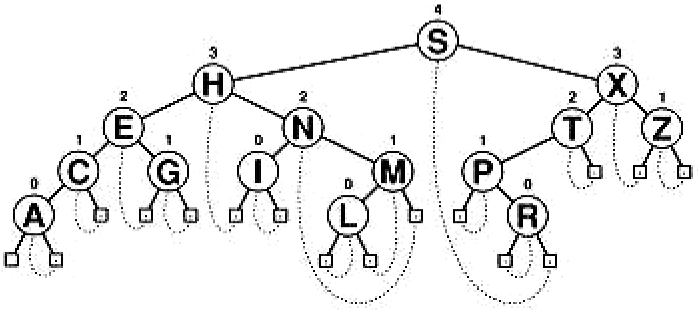
This function returns a link to the unique node which could contain the record with key *v*. The calling routine then can test whether the search was successful or not. Thus to search for *Z= 11010* in the above tree we go right and then up at the right link of *X*. The key there is not *Z*, so the search is unsuccessful.

Figure below shows the result of inserting *Z=11010* into the Patricia tree:



By the defining property of the tree, X is the only key in the tree for which a search would terminate at that node. If Z is inserted, there would be two such nodes, so the upward link that was followed into the node containing X must be made to point to a new node containing Z, with a bit index corresponding to the leftmost point where X and Z differ, and with two upward links: one pointing to X and the other pointing to Z. This corresponds precisely to replacing the external node containing X with a new internal node with X and Z as children in radix trie insertion, with one-way branching eliminated by including the bit index.

Inserting *T=10100* illustrates a more complicated case. . The search for T ends at *P=10000*, indicating that P is the only key in the tree with the pattern *10*0**. Now, T and P differ at bit 2, a position that was skipped during the search. The requirement that the bit indices decrease as we go down the tree dictates that T be inserted between X and P, with an upward self-pointer corresponding to its own bit 2. Note carefully that the fact that bit 2 was skipped before the insertion of T implies that P and R have the same bit-2 value:



Patricia is the quintessential radix searching method: it manages to identify the bits which distinguish the search keys and build them into a data structure (with no surplus nodes) that quickly leads from any search key to the only key in the data structure that could be equal. Clearly, the same technique as used in Patricia can be used in binary radix trie searching to eliminate one-way branching, but this only exacerbates the multiple-node-type problem.

Unlike standard binary tree search, the radix methods are insensitive to the order in which keys are inserted; they depend only upon the structure of the keys themselves. For Patricia the placement of the upwards links depend on the order of insertion, but the tree structure depends only on the bits in the keys, as in the other methods. Thus, even Patricia would have trouble with a set of keys like 001, 0001, 00001, 000001, etc., but for normal key sets, the tree should be relatively well-balanced so the number of bit inspections, even for very long keys, will be roughly proportional to *lgN* when there are N nodes in the tree.

Property: *A Patricia trie built from N random b-bit keys has N nodes and requires lgN bit comparisons for an average search.*

The most useful feature of radix trie searching is that it can be done efficiently with keys of varying length. In all of the other searching methods we have seen the length of the key is "built into" the searching procedure in some way, so that the running time is dependent on the length as well as the number of the keys. The specific savings available depends on the method of bit access used. For example, suppose we have a computer which can efficiently access 8-bit "bytes" of data, and we have to search among hundreds of 1000-bit keys. Then Patricia would require accessing only about 9 or 10 bytes of the search key for the search, plus one 125-byte equality comparison, while hashing would require accessing all 125 bytes of the search key to compute the hash function plus a few equality comparisons, and comparison-based methods require several long comparisons. This effect makes Patricia (or radix trie searching with one-way branching removed) the search method of choice when very long keys are involved.