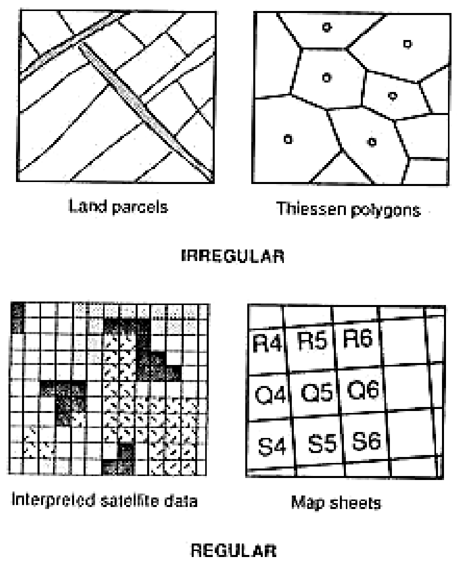
Discretization of spatial objects

Mosaics, Tessellations and Lattices

Discretization is the process of partitioning continuous space, either lines or areas, into pieces. This process is referred to as tiling in the case of two-dimensional spatial units, or segmenting in the case of lines.

Mosaics or tiles of zones of irregular and varied shape and size may be created via various processes. Originally conceived as polygons with four equal corners, as implied by the Greek word *tetara* or Latin word *tessella*, mosaics also can be made up of other regular geometric figures or a mixture of different types.

Tessellations (sometimes in the computer graphics field called meshes), are sets of connected discrete two-dimensional units:



Tessellations may be regular or irregular in geometry:

- a regular tessellation (for example, grid squares) is an (infinitely) repeatable pattern of a regular polygon (two-dimensional figure) or polyhedron (three-dimensional figure). This cellular decomposition implies every point in space is assigned to only one cell.
- an irregular tessellation is an (infinitely) extending configuration of polygons or polyhedra of varied shape and size. Irregular planar tessellations may also be regarded and represented as topological two-cells.

Irregular tessellations are used in areas such as:

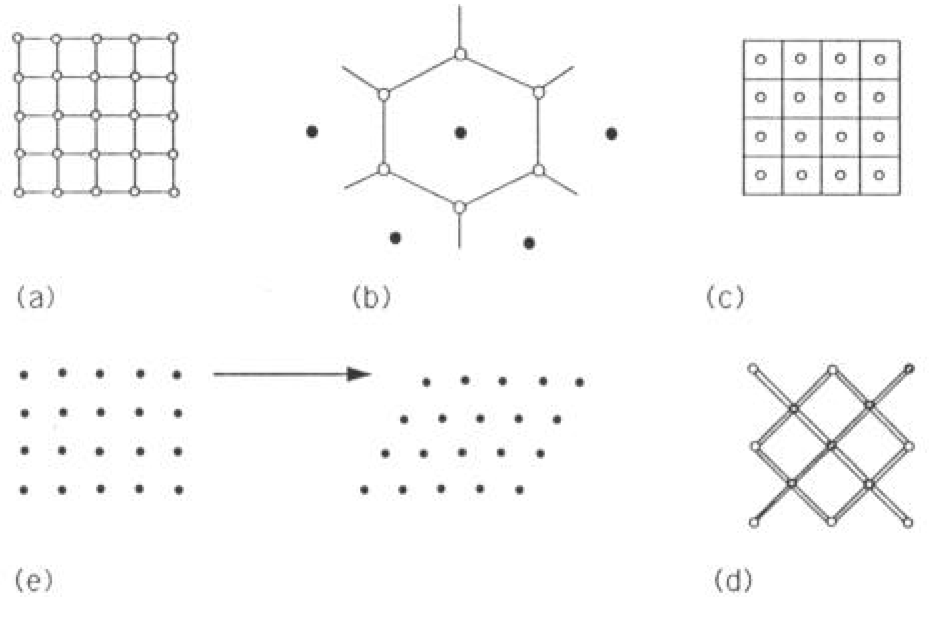
- zones for social, economic, and demographic data
- administrative or political units
- surface modelling using triangles

- partitions of a large geographic database
- man-made phenomena like land parcels
- irregular sampling of a continuous spatial distribution
- wireframe modelling of buildings

Regular tessellations are encountered in:
- image data from reniote sensing
- data compilations from maps by grid squares
- organization of map libraries
- data generated by photogrammetric systems as lattices of points
- uniform sampling of a continuous spatial distribution

The underlying semantic questions regarding what is in (this) space or at (this) point and what is the location of (this) object are not in a one-to-one relationship with geometric form. An object can be positioned in Cartesian coordinate space, or it could be referenced by location within a hexagon. Grid squares are often used to record the nature of space, but the square is just the extreme, perfectly symmetrical case for all tessellations.

Lattices are point entities arranged over space in a regular, perhaps square or hexagonal, pattern:

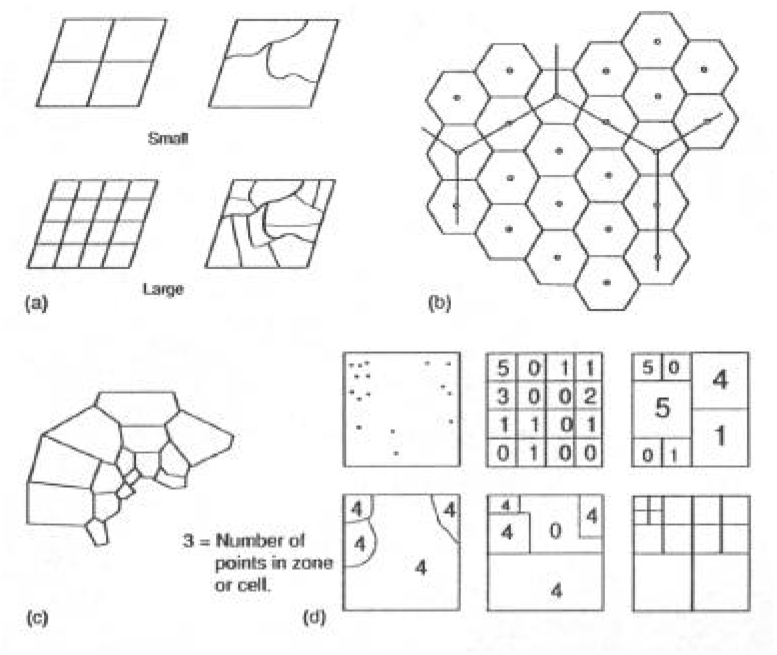


The lattices may be seen as the intersections of the grid lines or as the centres of a set of squares (c). Or, they may consist of a quinconcial pattern made up of the points and middle for a set of squares. Originally the word lattice referred to thin strips of wood or metal, crossing diagonally (d), and capable of forming different patterns or networks.

Moreover, the regular tessellations may be seen as different patterns of points. The patterns of the sets of points may change via application of a linear transformation, for example, tilting the *y*-axis, just as a lattice of wood pieces anchored by pins at the crossings can be changed into different forms by pushing or pulling in one or the other coordinate axis. Thus, sets of rectangles or parallelograms can be made from squares (e) by remapping the centroids of the squares. And a hexagonal arrangement of points can become a square grid of points or vice versa. There is not a one-to-one relationship between grid squares and a square lattice, for a square mosaic can be changed into a set of equidistant points arranged in a hexagonal pattern (or other forms), although there is a duality of zero (lattice points) and two cell (tessellation polygons) units.

Scale and resolution

The basic spatial units of a tessellation can vary in size, shape, orientation and spacing. Also, for a given theme, several tessellations can exist, representing different scales of observation, or aggregations or subdivisions from one initial scale of observation:



Variable resolution and scale:
(a) different scales, (b) different scales for hexagons, (c) variable resolution hexagons, (d) varying resolutions at one scale for point data.

For regularly shaped two-dimensional units the structuring is akin to a pyramid; for irregular shapes, the new larger units produced by combining smaller units, as in creating school districts from census tracts, will not have a shape consistent across the different scales. Varying scale is seen for lattices in different point densities per unit area.

The basic unit may vary in size for a given theme for a given scale. These concepts are met in theoretical formulations like the different sizes of hexagons for central place theory (b and c), a set of propositions which, among other things, addresses the question of the size and shape of market areas for services provided from towns. It also arises for practical applications like surface modelling triangulations. Irregular tessellations generally have implicitly **variable spatial resolution** (c), but regular tessellations with variable size units for one theme have to be produced from the smallest scale by subdivision.

Tessellations, mosaics and lattices are not only to be seen as spatial units for recording data, but also as devices for facilitating access to databases for continuous space. While any piece of the earth can conceivably be used for addressing and retrieval, regular tilings have some advantages for this purpose.

The Geometry of Regular Tessellations

Of the many regular geometric figures, from 3 to N sides, the square, triangle and hexagon are those currently most likely to be generally encountered in spatial data contexts. Some, like the pentagon, appear in special circumstances. The square has some practical advantages historically, being the unit used in graph paper and mechanical printing devices. The hexagon has utility in theory and practice where adjacency is an important property or where packing of geometric figures into space is important. The triangle has a property of being the most primitive polygon, that is, it cannot be further subdivided into a different figure, and it can readily have varying shape based on angles or length of sides conditions.

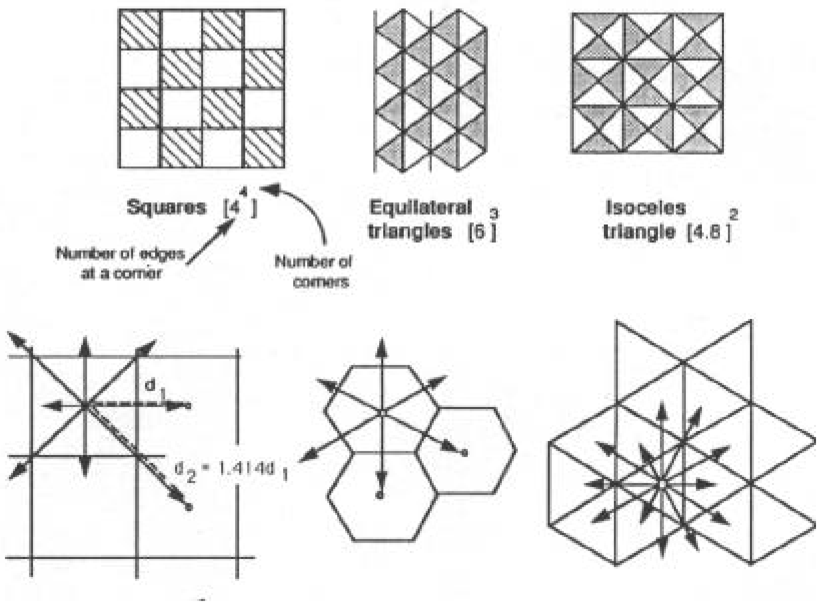
Comparisons of the value of different tiling units are best made on the basis of certain standard properties. In examining the geometry of regular tilings we will consider:

- shape
- adjacency
- connectivity
- orientation
- self-similarity
- decomposability
- packing properties

Shape is the geometry of the figure's different number of sides, albeit that particular instances of a particular type of figure may have different shapes, where there is no tight restriction on the length of sides or the angles, as for triangles. Shape for regular figures is identified by the number of edges at a vertex and the number of vertices.

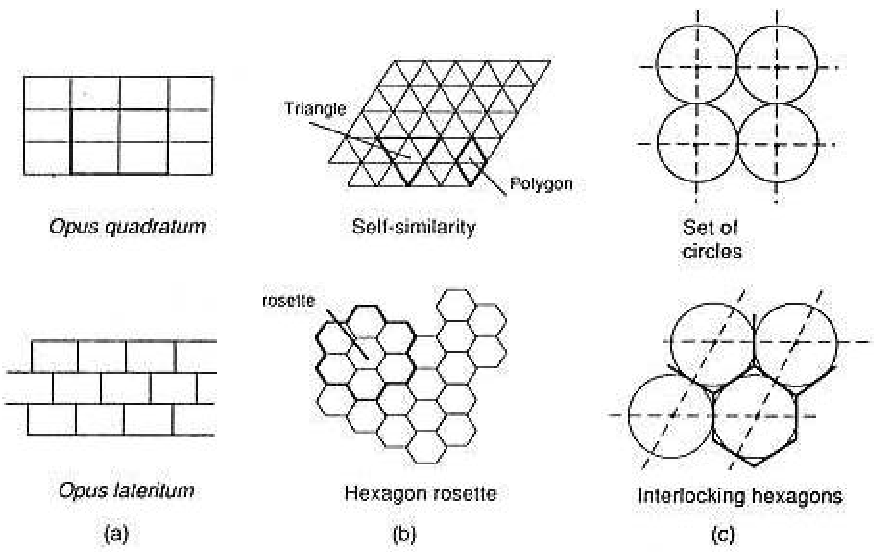
Adjacency refers to the conditions of touching for repeated instances of the basic figure. The contiguity may be at the sides (edges), or corners (vertices), or both. This situation varies from the sides only for hexagons, to both vertices and edges for triangles and squares or rectangles. Metric measurements can be used to distinguish different figures - the adjacency distance is the distance between the centroid of one tile and a neighbour, and the adjacency number is the number of different adjacency distances. The hexagon has one such distance, six in number (one for each equidistant neighbour). The square has two, and the equilateral triangle has three different distances.

Related to adjacency, but considering only the conditions at the intersections, for some applications, like transportation, the hexagon has the **connectivity** advantages of having junctions at only three edges, unlike the four for a regular grid, as in an American-style city block street system. For most purposes, though, the polygon space adjacency is of more interest that the intersection connectivity patterns.



Squares [4^4]
Number of edges at a corner
Number of corners
Equilateral triangles [6^3]
Isoceles triangle [4.8^2]
d_1
d_2 = 1.414d_1

The tile **orientation,** defined as the base relative to coordinate axes, is uniform for the hexagon and square, but not so for the triangle as not all instances of triangles in a tessellation can have the base parallel to either the *x* or *y* coordinate axes. 'T'iles with the same orientation can be transformed into each other by translations, that is, by shifting along **the** x- and y-axes, without necessitating rotation or reflection.



If the basic figure shape is similar to that produced by aggregations to a higher level, then the property of **self-similarity** exists. This is unlimited for regular figures which have each side on an infinitely straight line composed entirely of edges. This condition is seen clearly in the case of the tiles touching at corners *(opus quadratum)* as opposed to the bricklayer's scheme *(opus lateritum)* with only horizontal unbroken lines (Figure 6.5a). However, only two tessellation types, the square and equilatral triangle, with identical side lengths, and two tessellation figures with unequal side lengths, the isosceles and the 30-60 degrees right angle triangles, meet this condition.

It is also of interest to examine changing conditions if the figures are **decomposed** into smaller units or combined to larger units. The square and equilateral triangle can be decomposed into units of the same shape, but the hexagon cannot; it can be split into six equilateral triangles. Indeed, as (b) demonstrates, the aggregation of seven hexagons produces a rosette, while the aggregation of equilateral triangles produces polygons if pairs are taken, and higher-level triangles if four basic triangles are combined. While the hexagon is not decomposable with selfsimilarity, it can be used at different scales, rather as a lattice pattern, without nesting. Only the square can be decomposed into smaller or assembled into larger tiles of the same shape and orientation, and be compatible with the orthogonal axes of Cartesian coordinate systems.

The overall pattern of how **the** repeated regular figures **pack space** is an important condition in engineering for properties like stress, and is also important for living organisms in their effectiveness in the use of space. A structurally stable situation (c) is demonstrated by the relationships of a set of circles at right angles, at oblique angles representing rotation of one of the two coordinate axes, and then displacement to oblique axes where the vertical spacing is half the horizontal.

It is not surprising, though, on the basis of this review of properties, that the square figure dominates the world of regular tessellations for spatial information systems for two- or three-dimensional entities (except if a single sphere is to be partitioned). The square has simple and valuable conditions of equality of sides, decomposability, and stability for orientation and aggregation. So the square appears in the form of arrays of picture elements, grid cell data recording, lattices for elevation data, and electronic data capture and displays. The hexagon is more complex although it has nice properties of edge **touching** and economy of packing space. In theory it has had many uses, but not in practice. However, **recently** it **has** made an appearance as the data unit for one particular spatial database. The triangle has advantages of being the most primitive polygon, and does appear in several practical contexts, although in the form of irregular rather than regular triangles.

Fixed Spatial Resolutions: Regular Cell Grids

The commonly encountered organization of data into an array of regular cells, almost always, but not necessarily, **square**, is regarded as practically straightforward, but conceptually does not explicity deal with questions of the type: **Where do I find this particular entity?** The basic unit of the square handles both location, geographic or arbitrary, and the attributes of the location. Often known as raster encoding of earth features, this cell array approach contrasts with the vector object-oriented encodings, having some practical advantages, as well as some limitations.

Data encoding

Some fundamental considerations to be examined regarding grid-cell data organization are:

1. The existence of an *a priori* fixed resolution.
2. The method of determining what attribute is in a cell.
3. The level of measurement used for the attributes.
4. Limitations in recording precisely point or line features.
5. The absence of recording coordinates for features.
6. The lack of explicit topology.
7. The flexibility in undertaking many operations easily.
8. A match with a field view of the world.

Spatial properties

Spatial relationships are implicit in the data, but with only a few exceptions do the software systems for grid cell data allow direct handling of relationships between entities. Metrical distance relations along orthogonal axes are readily derived, although not without limitations, from the differences between *x* and *y* coordinates (row and column values), and other distances are computed as the hypotenuse of right angled triangles. While axial and diagonal lines can be measured by cell increments without loss of precision, this is not so for other hypotenuses. Thus, perimeter measures for areal objects or lengths for linear features may be severely erroneous. Shapes can be approximated by observing orientations of lines of cells relative to those of neighbours, assuming that the resolution is good enough to produce sufficient unambiguous data.

Metrical spatial properties can be readily dealt with by square cell tessellations, bearing in mind the resolution implied by the cell size. Distances are harder to obtain for hexagons, because the centres of the cells are arranged symmetrically along three coordinate axes, but properties based on neighbours are much easier because there are no corner adjacencies. Distances and adjacencies are more awkward for sets of equilateral triangles, because they have six triangles touching at corners and three different distance values for each triangle centre.

Graph properties of connectivity have to be explicitly recorded for cells containing nodes, and details of direction of edges leaving nodes (necessary in order to create a precise graph) must also be encoded. Consequently, the desirable situation is a combination of cells and vector representation of point and line elements within cells. Details for points and lines within a grid cell could be retained by particular coding techniques, or by linking the grid cell to other data by reference to the grid cell identifier. Some techniques have been devised to record information explicitly in a grid framework for line and polygon entities. Thus, the directional coding presented in section 4.5 is sometimes used for boundary line or linear feature representation.

Topological properties are only indirectly dealt with by the regular tessellation forms, because they do not treat phenomena as entities. Of course, there are properties of adjacency and connectivity among cells, but qualitative spatial relationships for entities must be derived from the data that are recorded. The topological properties of non-confiectivity may be violated, such as the apparent joining of highways or touching of inner and outer boundaries of polygons. The graph property of existence of holes can be determined by calculating the distance to the nearest edge of a polygon (a set of cells).

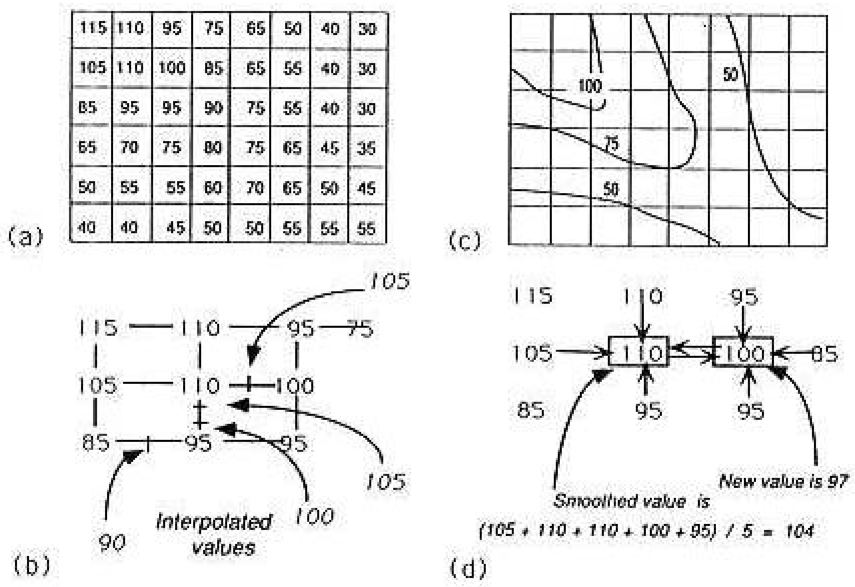
Cell encoding forces pattern to be inferred from the data values for adjacent cells, but it does facilitate the process as it directly establishes neighbours and neighbourhoods. If cells are square, then eight touching pairs exist in the immediate ring around a cell; other situations may take in other cells, such as the neighbours of a block of four cells. Yet, unlike for topological encoding, the identification of neighboujing regular cells provides little information unless the attributes of the cells are considered, such as in creating zones by finding cells of a particular attribute value.

However, many operations, logical or arithmetic, are quite straightforward when working with grid cell data. Area measures simply involve counting squares, not computational geometry; distances involve subtracting, and can be readily ascertained even on diagonals. Aggregations can be made within proximity zones, and map overlay modelling using attribute information essentially does nothing more than compare attribute values for identical grid cells. At the same time, vector geometry may have an advantage in that much of the space is void: for regular tessellations, the space is 'full'.

Surface modelling from lattices

The elevation dimension is readily manipulated and displayed by means of data for cells, although it is usual to record such data as a lattice. Such data, often referred to as digital elevation models because they are used for interpolating or extrapolating values for places in addition to the original data points, generally are themselves produced from other data by a process of gridding. Each cell or point carries data for an elevation or depth relative to a horizontal datum.

Much use has been made of grid cells or lattice-point data for representing fields of varying quantities across space, whether continuous or broken. The mesh form of the data, whether the values are for grid line intersections or cell centroids, lends itself to simple linear interpolations along the x- and y-axes.

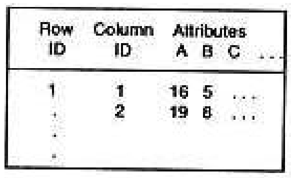


115	110	95	75	65	50	40	30
105	110	100	85	65	55	40	30
85	95	95	90	75	55	40	30
65	70	75	80	75	65	45	35
50	55	55	60	70	65	50	45
40	40	45	50	50	55	55	55

The outcome is often simply an isoline map. Proximity surfaces are easily created from the mesh data by accumulating distances in (circular) zones from specified points, or in bands out from line or area objects. Smoothing of attribute values for blocks of touching cells, five or nine, provides a means of generalizing the spatial attributes which are in scalar forms of measurement.

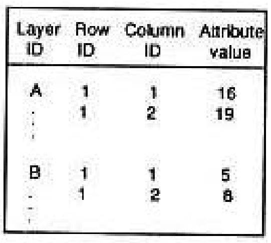
Structures for grid cell data

Just as there are numerous varieties of organizing geometric and topological properties in data tables, so there are different ways of structuring regular tessellation data:



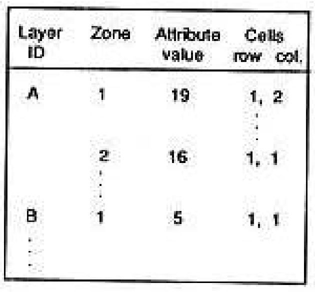
Row ID	Column ID	Attributes A	B	C ...
1	1	16	5	...
⋮	2	19	8	...
⋮				

(a)



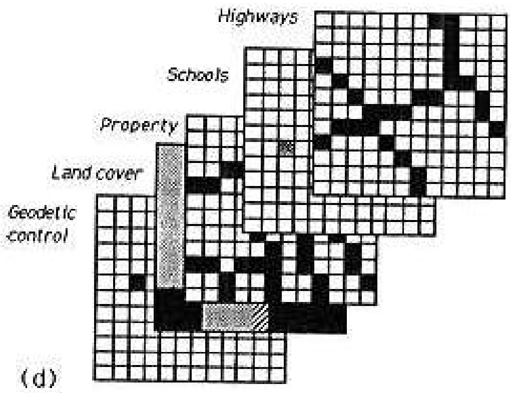
Layer ID	Row ID	Column ID	Attribute value
A	1	1	16
⋮	1	2	19
B	1	1	5
⋮	1	2	8

(b)



Layer ID	Zone	Attribute value	Cells row col.
A	1	19	1, 2
	⋮		⋮
	2	16	1, 1
	⋮		
B	1	5	1, 1
⋮			

(c)



(d)

The common form for a basic tabular record of data for grid cells consists of a row coordinate value, a column coordinate value, possibly a cell identifier, in addition to the row-column combination, a series of values for attributes found in a cell, possibly a code value pointing to another table of data, a reference to the identifier for the nearest cell on the sides or corners. If only one attribute is of interest, then the basic table consists of a row identifier, column identifier (jointly making a space locator) and one data item: an organization reflecting each grid cell as an individual entity.

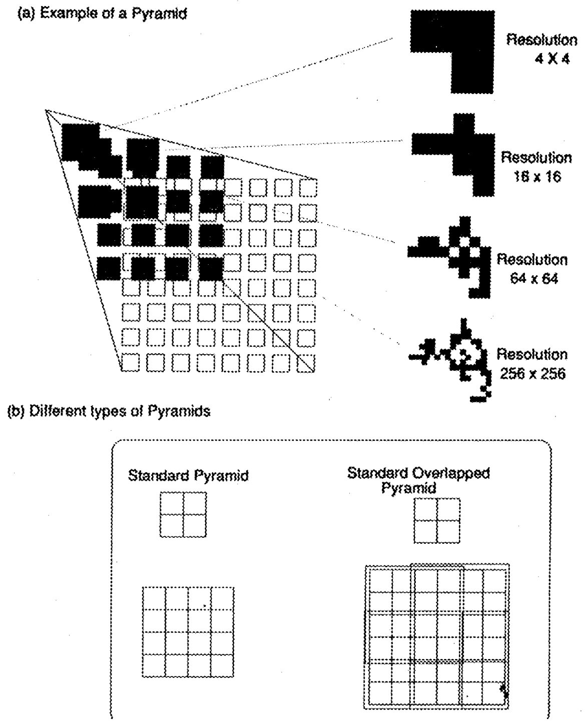
The records, that is one per grid cell, can be listed sequentially in row order or column order, or, taking advantage of computer storage formats known as arrays, the entire matrix of cells can be stored (b). The matrix format may also be used if there are several data items for each grid cell. The term **raster** (from the German *raster,* meaning screen), while often used interchangeably with grid cell data, implies, though, an ordering through the whole matrix in line scan form, that is, one row followed by the others, as in the set of parallel lines making up a television picture. Another variety **clusters** observations in groups of like character, and then provides coordinates for each pixel (c). Conceptually, a grid encoded database looks like (d).

Although often associated with a map layer concept, the overlay arrangement is not a requirement for regular tessellations as the grid-cell or pixel form suggests. However, the matrix and clustered forms are designed to work with thematic layers, meaning that the attribute data are not recorded in sequence for each and every cell. While the layering architecture is utilized for both entity-oriented and tessellation encoding of phenomena, the two types certainly do treat attribute data differently. For entity-oriented representations, attributes are separated from the spatial information in most cases, whereas for regular tessellations the positional and attribute data are associated.

Spatial **aggregation** can be achieved quite well for tessellations by combinations of adjacent cells; spatial disaggregation can be dealt with by splitting cells. Aggregations might be necessary to facilitate generalization from one resolution (geographic scale) to another. For example, the number of points in four cells can be added to get the incidence for a set of four cells, or, if the attributes are scalar, an average could be obtained. Combinations might be necessary to produce the grid cell equivalent of homogeneous regions or administrative districts.

However, awkwardness arises for shapes other than the square figure. Hexagons can be subdivided into only the triangle regular figure, although this is a straightforward process; equilateral triangles may be subdivided into other triangles, but they will not always be equilateral. Triangles can be combined into parallelograms, or polygons, or hexagons depending on how many are joined; and hexagons produce rosette shapes when seven are joined together.

It is not surprising, therefore, that the square cell is used for aggregating or disaggregating across scales. The resulting form, known as the pyramid model:



provides a multiple scale representation, with spatial units constant for a given scale. Used often in image processing, it provides a means to remove or hide detail in order to focus on structural components like general shape, or to reveal details of form that might be hidden by too much generalization.

It is designed for rapid detection of global (overall, not earthly) features in a complex image. An ordinary pyramid has blocks of four cells combining to larger cells at higher levels, without overlap. A standard overlapped pyramid has 50 per cent overlap for adjacent blocks (b).

A hierarchical representation has uses in image processing:

- for browsing at different scales,
- for simplifying mapping,
- for matching data collected at different resolutions,
- for access at different levels.

In this way, the grid cell becomes a handy spatial indexing tool, rather than a storage unit, for space filling curves.

The B -Tree: a Balanced Multiway Search Tree

It is customary to classify a multiway tree by the maximum number of ***branches*** at each node, rather than the maximum number of items which may be stored at each node. If we use a multiway search tree with ***M*** possible branches at each node, then we can store up to ***M - 1*** data items at each node.
Since multiway trees are primarily used in databases, the data that are stored are usually of a fairly ***complex type***.

In each node of a multiway search tree, we may have up to ***M - 1*** keys labeled

To get the greatest efficiency gain out of a multiway search tree, we need to ensure that most of the nodes contain as much data as possible, and that the tree is as balanced as possible. There are several algorithms which approach this problem from various angles, but the most popular method is the ***B-tree***:

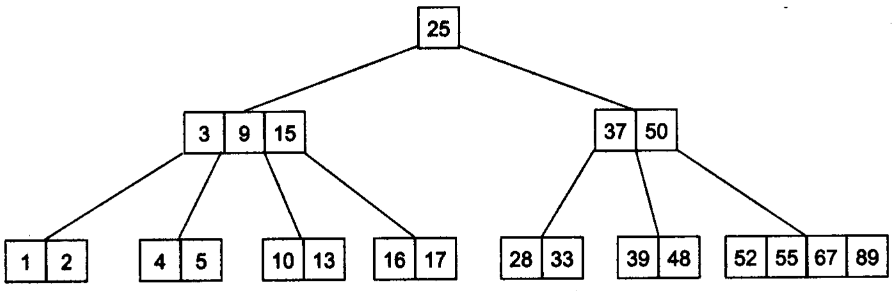
1 a ***B-tree*** is a multiway search tree with a maximum of ***M*** branches at each node. The number *M* is called the ***order*** of the tree.

2 there is ***a single root node*** which may have as few as two children, or none at all if the root is the only node in the tree.

3 at all nodes, except the root and leaf nodes, there must be ***at least half the maximum*** number of children.

4 all leaves are on the ***same level***.

A B-tree of order 5 is shown:



An example of how a ***B-tree*** (or any multiway tree) is searched for an item, which is presented in B-tree and is presented not. Calculating the efficiency of a B-tree.

Constructing a B-Tree

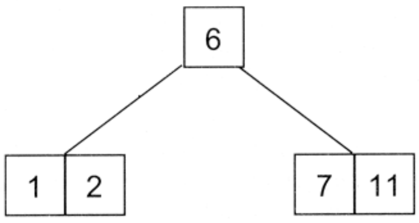
The insertion method for a B-tree is somewhat different to that for the other trees we have studied, since the condition that all leaves be on the same level forces insertion into the upper part of the tree. It is easiest to learn the insertion procedure by example, so we will construct an order-5 B-tree from the list of integers:

1 7 6 2 11 4 8 13 10 5 19 9 18 24 3 12 14 20 21 16

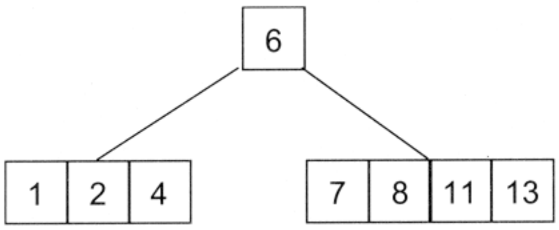
Since each node can have up to five branches, each node can store up to four keys. Therefore, the first four keys can be placed in the root, in sorted order, as shown:

1	2	6	7

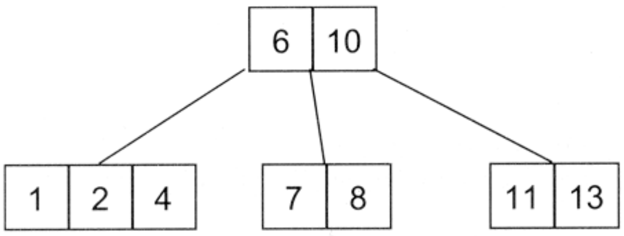
The fifth key, 11, will require the ***creation*** of a new node, since the root is full. In order ***not to violate*** one of the conditions on a B-tree: the root s not allowed to have only ***a single child***, we split the root at its midpoint and create two new nodes, leaving only the middle key in the root. This gives the tree shown:



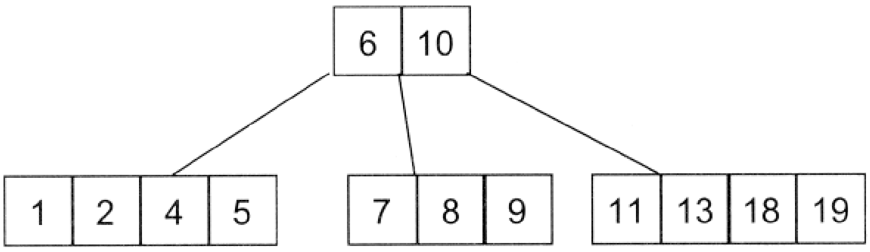
We can add the next three keys wothout having to create any more nodes:



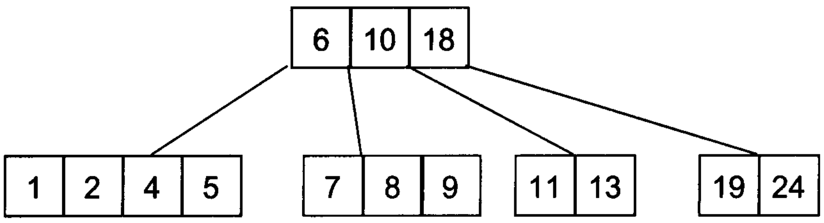
When we wish to add the next key, 10, it would fit into the right child of the root, but this node is full. We split the node, putting the middle key into the node's parent:



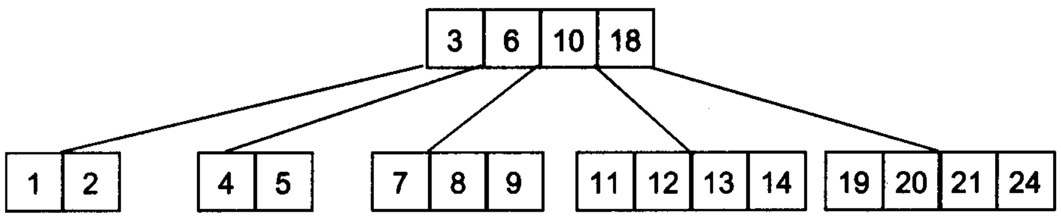
Now we can insert next four keys without any problems:



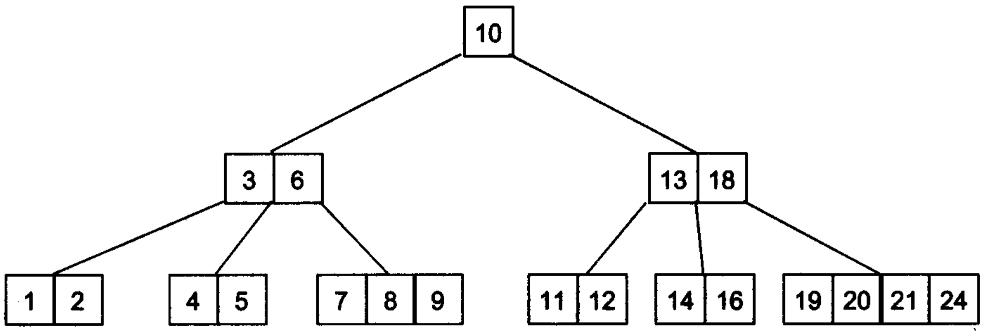
Inserting the key 24 causes another split and increases the number of keys in the root to three:



We can now insert another five keys:



Insertion of the final key, 16, causes the fourth leaf from the left to split, and pushes its middle key, 13, upwards. However, the parent node is also full, so it must split as well, following the same rules. This results in a new root node, and increases the height of the tree to three levels. The completed B-tree is shown:



The algorithm for insertion into a B-tree can be summarized as follows:

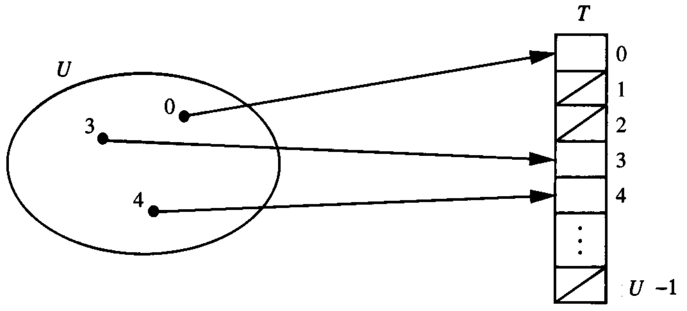
1. Find the node into which the new key should be inserted by searching the tree.
2. If the node is not full, insert the key into the node, using an appropriate sorting algorithm.
3. If the node is full, split the node into two and push the middle key upwards into the parent. If the parent is also full, follow the same procedure (splitting and pushing upwards) until either some space is found in a previously existing node, or a new root node is created.

Although the algorithm for insertion may look straightforward on paper, it contains quite a few subtleties which only come out when you try to program it. A program implementing this insertion routine is a nontrivial affair, and could be used as the basis for a programming project.

Hashing Procedures

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

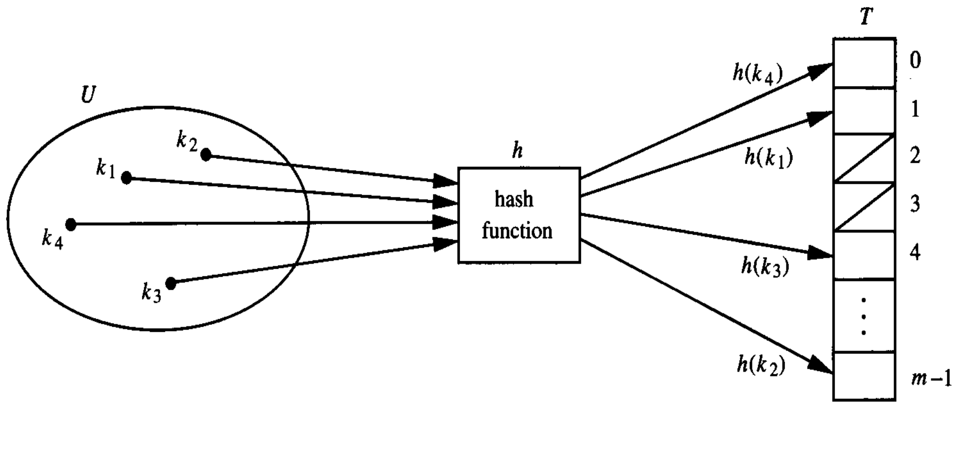


The obvious shortcoming associated with direct-access tables is that the set ***U*** rarely has such "nice" properties. In practice, ***|U|*** can be quite large. This will lead to wasted memory if the number of elements actually stored in the table is small relative to ***|U|***.

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 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 $\Theta(1)$ time, then each of the DICTIONARY 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 $T\ [0, \ldots, m - 1]$:

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

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 $i \neq j$, $h(k_i) = h(k_j)$.

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.

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.

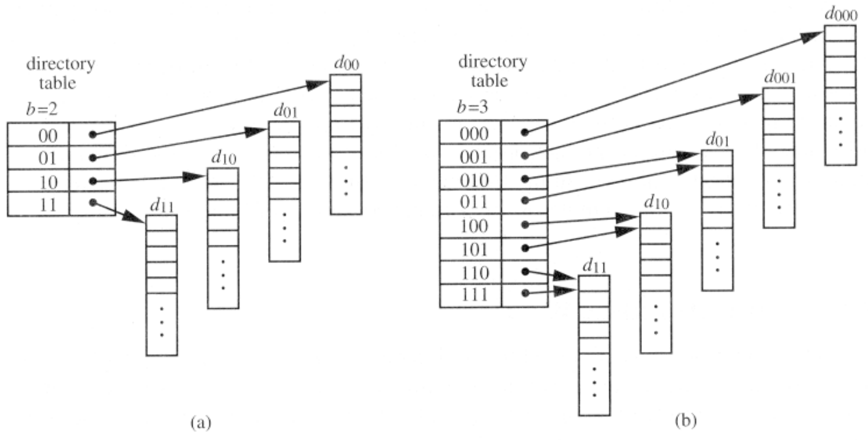
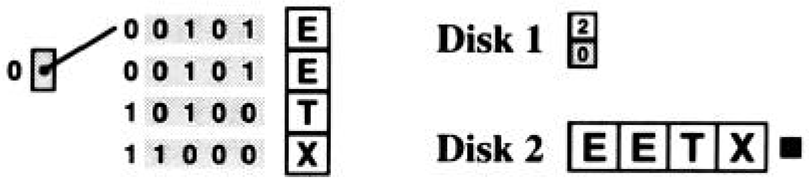


Table overflow can now be handled as follows. Whenever the load factor ***a***, of any one block ***d*** is exceeded, an additional block ***d'*** the same size as ***d*** is created, and the elements originally in ***d*** are rehashed into both ***d*** and ***d'*** using ***b + 1*** low-order bits in 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.

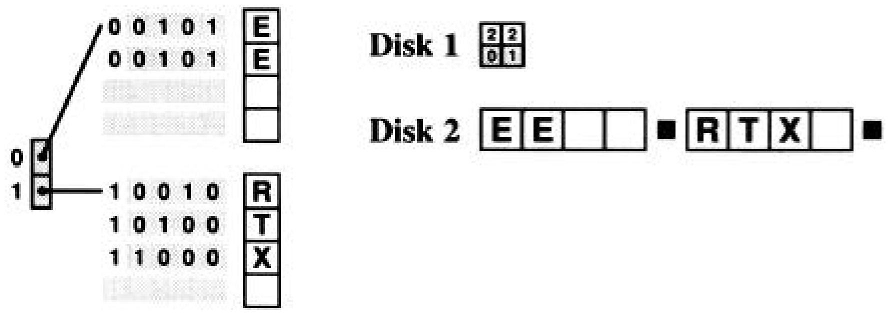
Extendible hashing, if applied to disk memory, gives an alternative to B-trees. This method involves two disk accesses for each search in typical applications while at the same time allowing efficient insertion. The records are stored on pages (clusters) which are split into two pieces when they fill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

The example, for the string of input data:

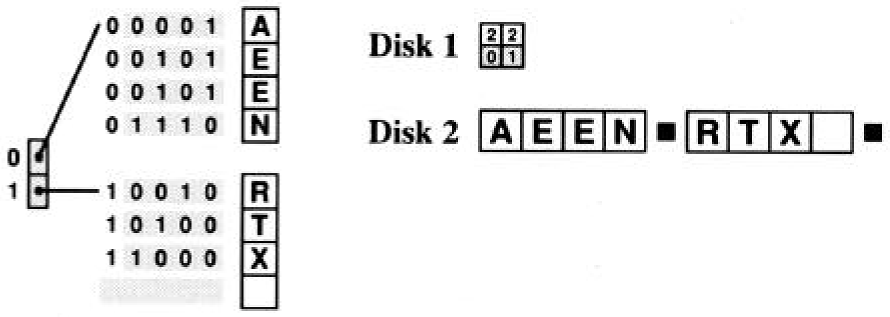
“E X T E R N A L S E A R C H I N G E X A M P L E”



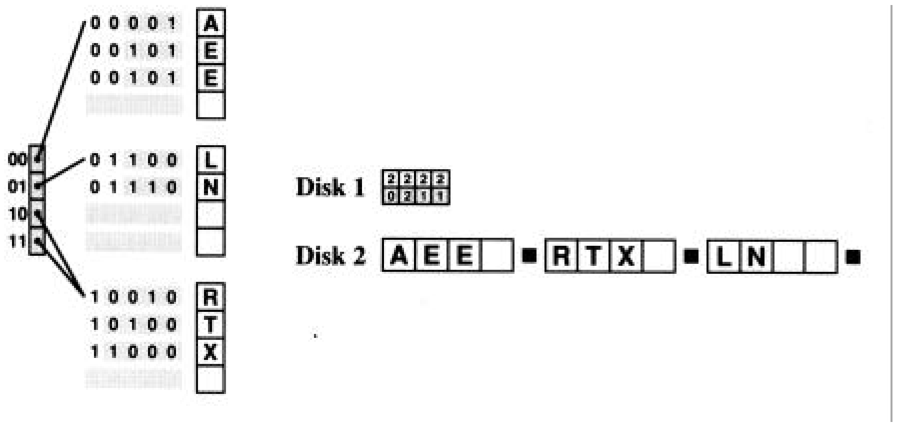
The first page



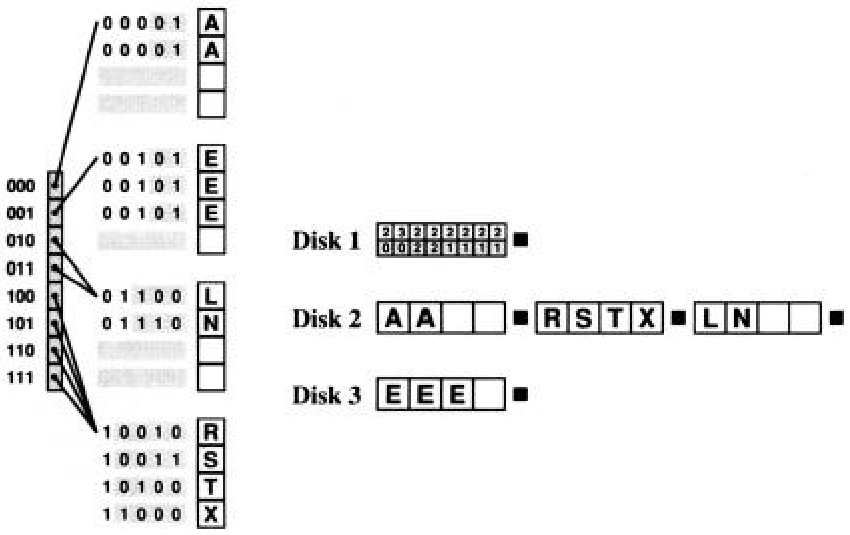
Directory split



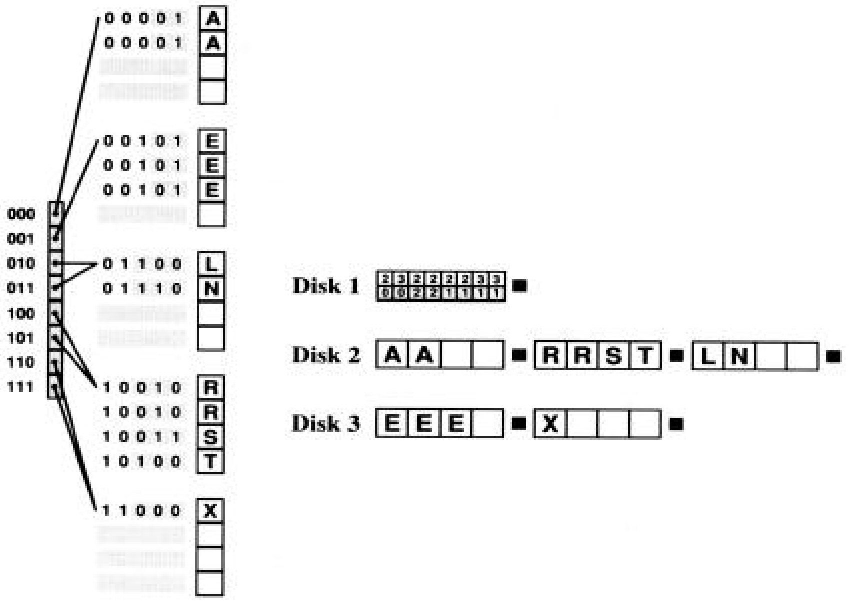
First page again full



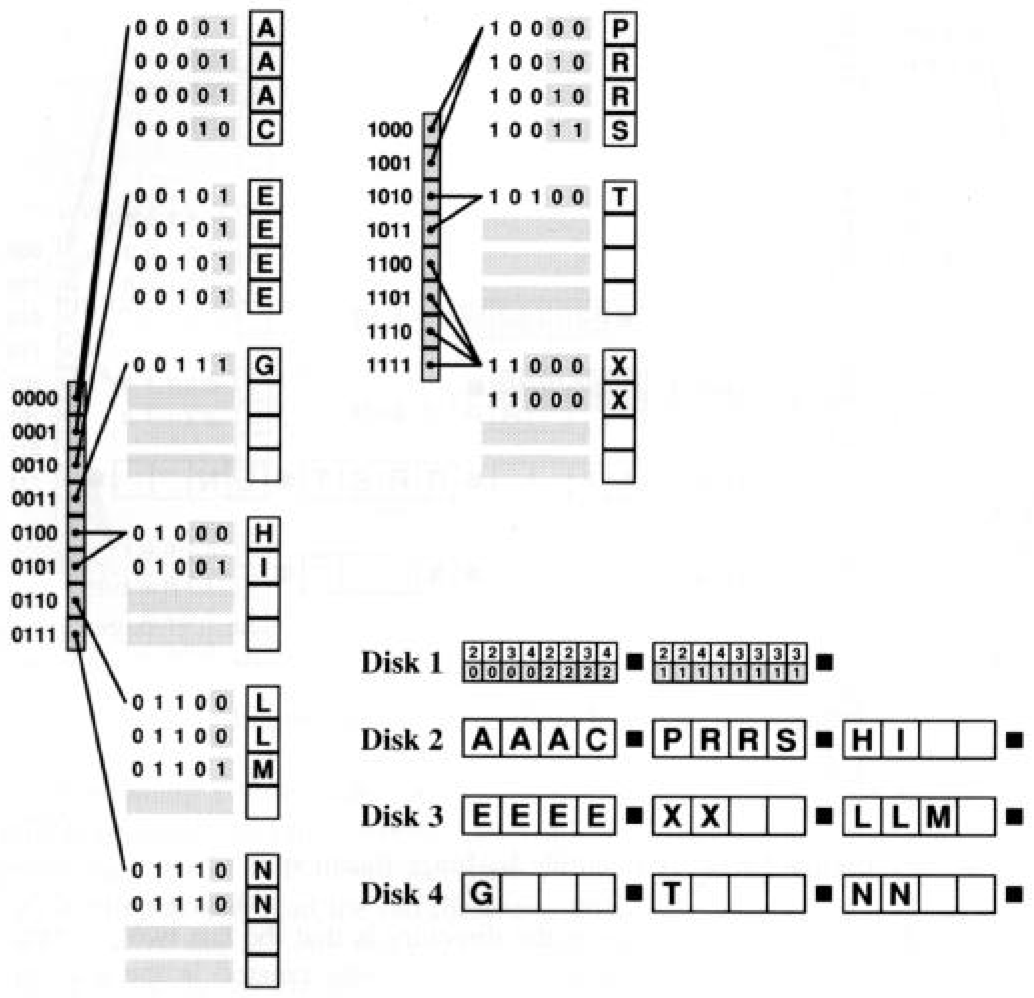
Second split



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.