

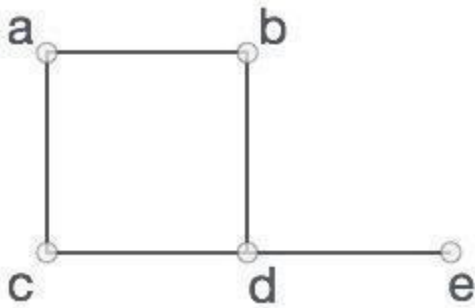
Module-5**Graph****Teaching Hours: 10**

CONTENTS	Pg no
<i>1. Graphs Introduction</i>	<i>109</i>
<i>2. Traversal methods</i>	<i>110</i>
2.1. Breadth First Search	<i>110</i>
2.2. Depth First Search	<i>113</i>
<i>3. Sorting and Searching</i>	<i>116</i>
3.1. Insertion Sort	<i>116</i>
3.2. Radix sort	<i>118</i>
3.3. Address Calculation Sort	<i>120</i>
<i>4. Hashing</i>	<i>121</i>
4.1. The Hash Table organizations	<i>121</i>
4.2. Hashing Functions	<i>122</i>
4.3. Static and Dynamic Hashing	<i>123</i>
4.4. Collision-Resolution Techniques	<i>125</i>
<i>5. File Structures</i>	<i>131</i>

1. Graphs Introduction

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

$$V = \{a, b, c, d, e\}$$

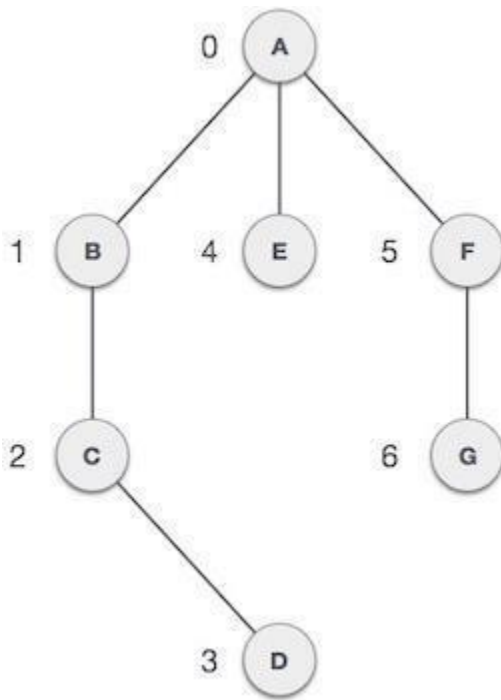
$$E = \{ab, ac, bd, cd, de\}$$

Graph Data Structure

Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.

- **Path** – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph which are following.

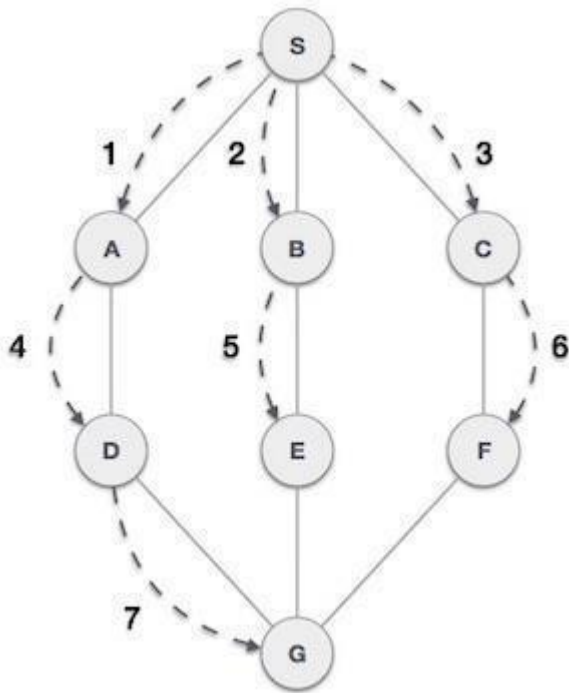
- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

To know more about Graph, please read Graph Theory Tutorial. We shall learn traversing a graph in coming chapters.

2. Traversal methods

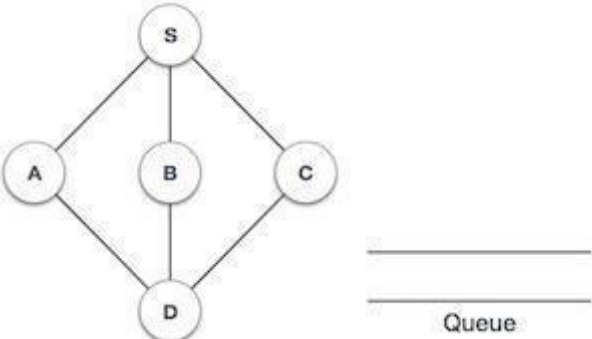
2.1. Breadth First Search

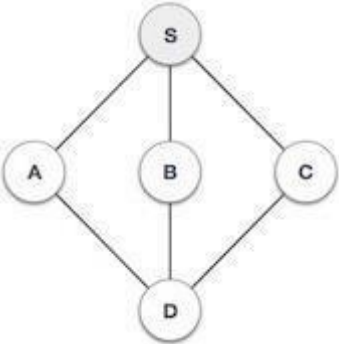
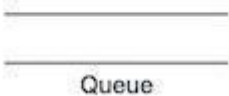
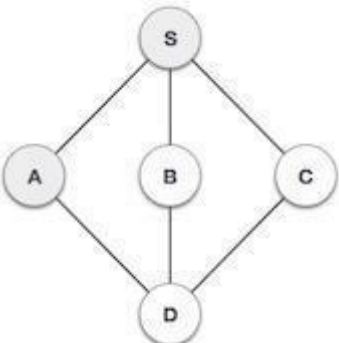
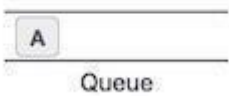
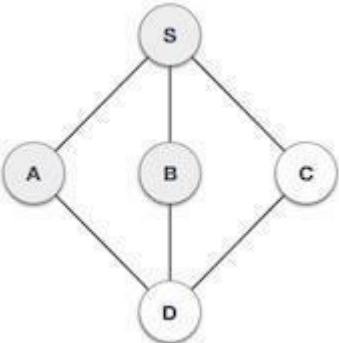
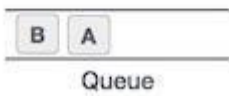
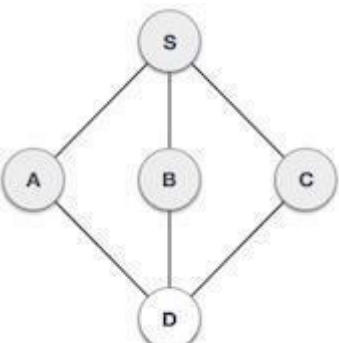
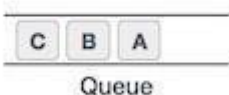
Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

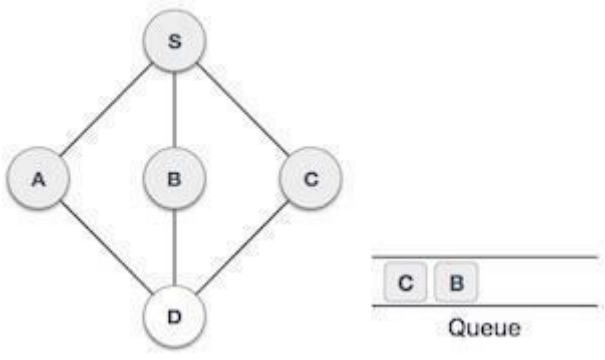
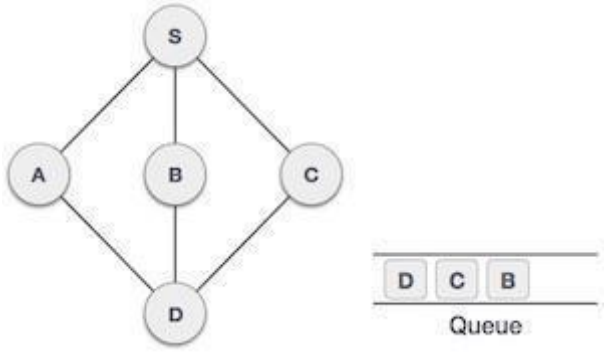


As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

Step	Traversal	Description
1.		Initialize the queue.

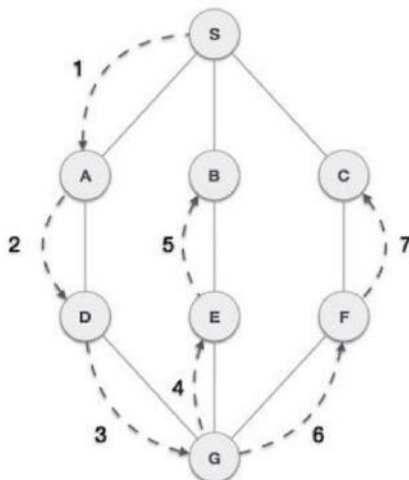
2.	 	We start from visiting S (starting node), and mark it visited.
3.	 	We then see unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A mark it visited and enqueue it.
4.	 	Next unvisited adjacent node from S is B . We mark it visited and enqueue it.
5.	 	Next unvisited adjacent node from S is C . We mark it visited and enqueue it.

6.		<p>Now S is left with no unvisited adjacent nodes. So we dequeue and find A.</p>
7.		<p>From A we have D as unvisited adjacent node. We mark it visited and enqueue it.</p>

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.

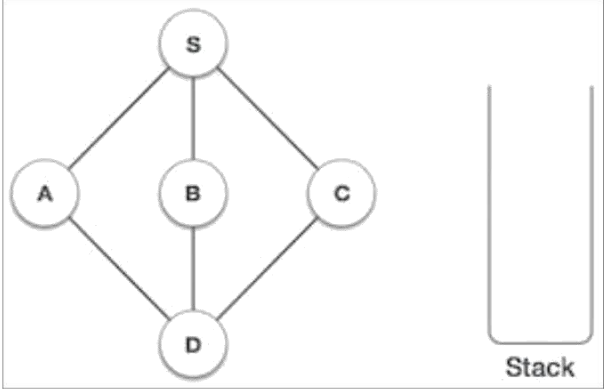
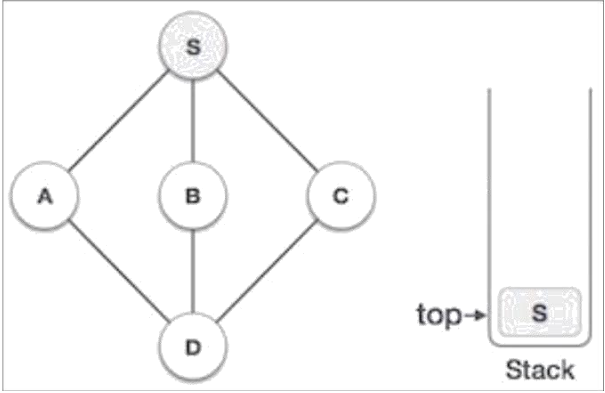
2.2. Depth First Search

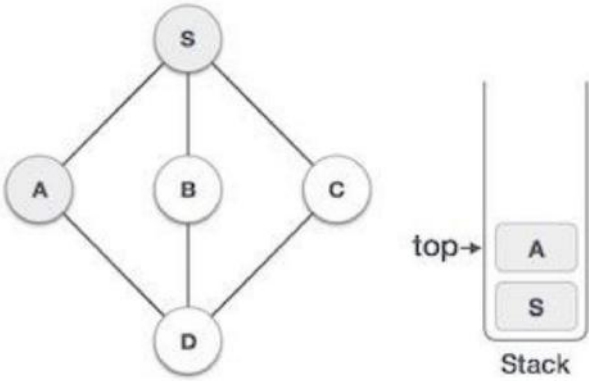
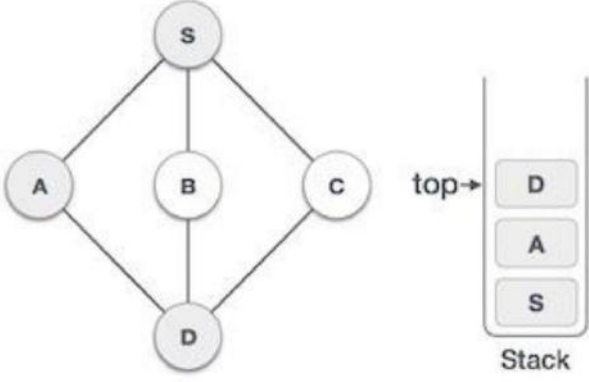
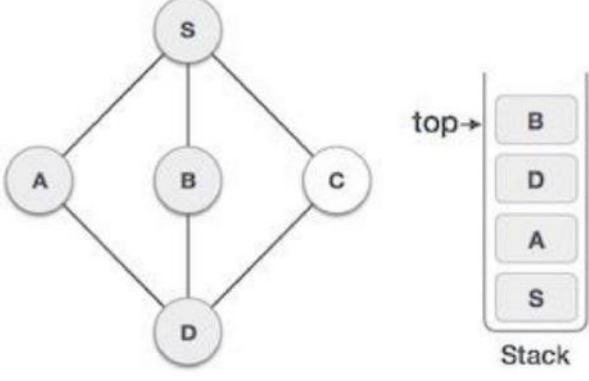
Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

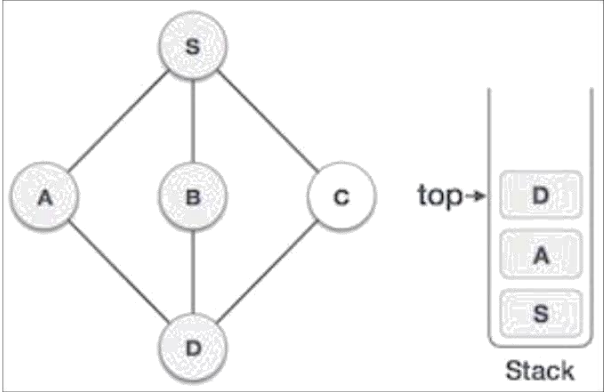
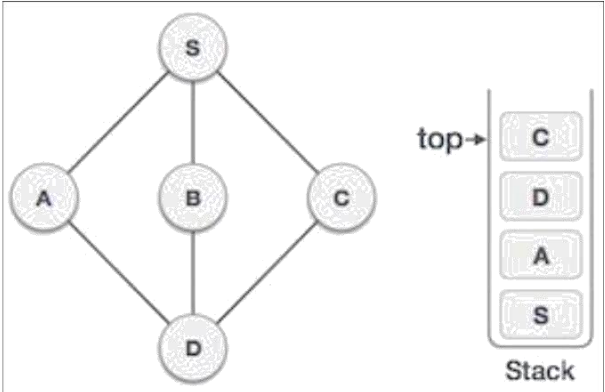


As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

Step	Traversal	Description
1.		Initialize the stack
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.

3.	 <p>The graph shows nodes S, A, B, C, and D. S is at the top, connected to A, B, and C. A, B, and C are in the middle row, and D is at the bottom, connected to A, B, and C. To the right, a stack is shown with 'A' at the top and 'S' below it. A 'top' pointer points to 'A'. The stack is labeled 'Stack' at the bottom.</p>	<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4.	 <p>The graph is the same as in step 3. The stack now has 'D' at the top, 'A' below it, and 'S' at the bottom. The 'top' pointer points to 'D'. The stack is labeled 'Stack' at the bottom.</p>	<p>Visit D and mark it visited and put onto the stack. Here we have B and C nodes which are adjacent to D and both are unvisited. But we shall again choose in alphabetical order.</p>
5.	 <p>The graph is the same as in step 3. The stack now has 'B' at the top, 'D' below it, 'A' below that, and 'S' at the bottom. The 'top' pointer points to 'B'. The stack is labeled 'Stack' at the bottom.</p>	<p>We choose B, mark it visited and put onto stack. Here B does not have any unvisited adjacent node. So we pop B from the stack.</p>

6.		<p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find D to be on the top of stack.</p>
7.		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it visited and put it onto the stack.</p>

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

3. Sorting and Searching

3.1. Insertion Sort

This is a in-place comparison based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. A element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and insert it there. Hence the name **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n are no. of items.

How insertion sort works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in correct position.



It swaps 33 with 27. Also it checks with all the elements of sorted sublist. Here we see that sorted sub-list has only one element 14 and 27 is greater than 14. Hence sorted sub-list remain sorted after swapping.



By now we have 14 and 27 in the sorted sublist. Next it compares 33 with 10,.



These values are not in sorted order.



So we swap them.



But swapping makes 27 and 10 unsorted.



So we swap them too.



Again we find 14 and 10 in unsorted order.



And we swap them. By the end of third iteration we have a sorted sublist of 4 items.



This process goes until all the unsorted values are covered in sorted sublist. And now we shall see some programming aspects of insertion sort.

3.2. Radix sort

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710
1	

	2	812 582
	3	493
	4	
	5	715 195 385
6		
	7	437
8		
9		

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	

7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

3.3. Address Calculation Sort

- In this method a function f is applied to each key.
- The result of this function determines into which of the several subfiles the record is to be placed.
- The function should have the property that: if $x \leq y$, $f(x) \leq f(y)$, Such a function is called order preserving.
- An item is placed into a subfile in correct sequence by placing sorting method – simple insertion is often used.

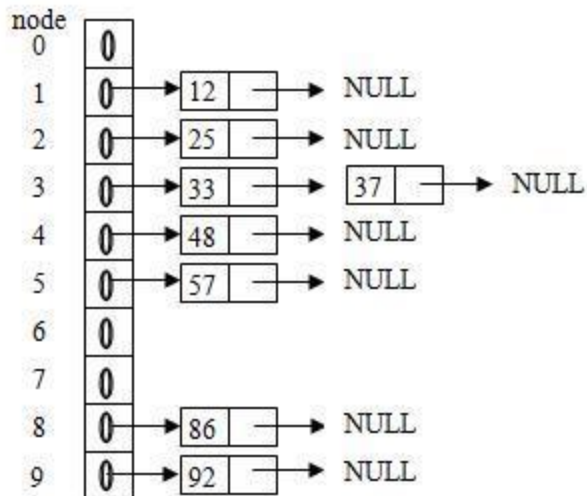
Example:

25 57 48 37 12 92 86 33

Let us create 10 subfiles. Initially each of these subfiles is empty. An array of pointer $f(10)$ is declared, where $f(i)$ refers to the first element in the file, whose first digit is i . The number is passed to hash function, which returns its last digit (ten's place digit), which is placed at that position only, in the array of pointers.

num=	25	—	$f(25)$	gives	2
57	—		$f(57)$	gives	5
48	—		$f(48)$	gives	4
37	—		$f(37)$	gives	3
12	—		$f(12)$	gives	1
92	—		$f(92)$	gives	9
86	—		$f(86)$	gives	8
33	—	$f(33)$ gives 3	which is repeated.		

Thus it is inserted in 3rd subfile (4th) only, but must be checked with the existing elements for its proper position in this subfile.



4. Hashing

4.1. The Hash Table organizations

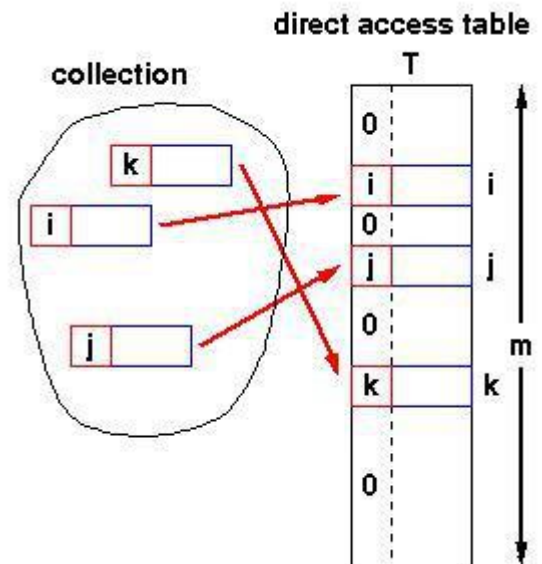
If we have a collection of n elements whose keys are unique integers in $(1, m)$, where $m \geq n$, then we can store the items in a *direct address table*, $T[m]$, where T_i is either empty or contains one of the elements of our collection.

Searching a direct address table is clearly an $O(1)$ operation: for a key, k , we access T_k ,

- if it contains an element, return it,
- if it doesn't then return a NULL.

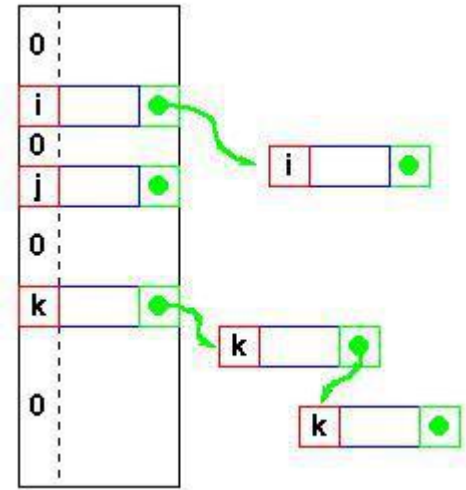
There are two constraints here:

1. the keys must be unique, and
2. the range of the key must be severely bounded.



If the keys are not unique, then we can simply construct a set of m lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be $O(1)$.

However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number of duplicates is $ndup^{max}$, then searching for a specific element is $O(ndup^{max})$. If duplicates are the exception rather than the rule, then $ndup^{max}$ is much smaller than n and a direct address table will provide good performance. But if $ndup^{max}$ approaches n , then the time to find a specific element is $O(n)$ and a tree structure will be more efficient.



The range of the key determines the size of the direct address table and may be too large to be practical. For instance it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32-bit integers as their keys for a few years yet!

Direct addressing is easily generalised to the case where there is a function,

$$h(k) \Rightarrow (1, m)$$

which maps each value of the key, k , to the range $(1, m)$. In this case, we place the element in $T[h(k)]$ rather than $T[k]$ and we can search in $O(1)$ time as before.

4.2. Hashing Functions

The following functions map a single integer key (k) to a small integer bucket value $h(k)$. m is the size of the hash table (number of buckets).

Division method (Cormen) Choose a prime that isn't close to a power of 2. $h(k) = k \bmod m$. Works badly for many types of patterns in the input data.

Knuth Variant on Division $h(k) = k(k+3) \bmod m$. Supposedly works much better than the raw division method.

Multiplication Method (Cormen). Choose m to be a power of 2. Let A be some random-looking real number. Knuth suggests $M = 0.5 * (\sqrt{5} - 1)$. Then do the following:

$$\begin{aligned} s &= k * A \\ x &= \text{fractional part of } s \\ h(k) &= \text{floor}(m * x) \end{aligned}$$

This seems to be the method that the theoreticians like.

To do this quickly with integer arithmetic, let w be the number of bits in a word (e.g. 32) and suppose m is 2^p . Then compute:

```
s = floor(A * 2^w)
x = k*s
h(k) = x >> (w-p)    // i.e. right shift x by (w-p) bits
                    // i.e. extract the p most significant
                    // bits from x
```

4.3. Static and Dynamic Hashing

The good functioning of a hash table depends on the fact that the table size is proportional to the number of entries. With a fixed size, and the common structures, it is similar to linear search, except with a better constant factor. In some cases, the number of entries may be definitely known in advance, for example keywords in a language. More commonly, this is not known for sure, if only due to later changes in code and data. It is one serious, although common, mistake to not provide *any* way for the table to resize. A general-purpose hash table "class" will almost always have some way to resize, and it is good practice even for simple "custom" tables. An implementation should check the load factor, and do something if it becomes too large (this needs to be done only on inserts, since that is the only thing that would increase it).

To keep the load factor under a certain limit, e.g., under $3/4$, many table implementations expand the table when items are inserted. For example, in Java's `HashMap` class the default load factor threshold for table expansion is $3/4$ and in Python's `dict`, table size is resized when load factor is greater than $2/3$.

Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays.

Resizing is accompanied by a full or incremental table *rehash* whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table—followed by a rehash—when items are deleted. From the point of space-time tradeoffs, this operation is similar to the deallocation in dynamic arrays.

Resizing by copying all entries

A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold r_{\max} . Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold r_{\min} , all entries are moved to a new smaller table.

For hash tables that shrink and grow frequently, the resizing downward can be skipped entirely. In this case, the table size is proportional to the maximum number of entries that ever were in the hash table at one time, rather

than the current number. The disadvantage is that memory usage will be higher, and thus cache behavior may be worse. For best control, a "shrink-to-fit" operation can be provided that does this only on request.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries n and of the number m of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If m elements are inserted into that table, the total number of extra re-insertions that occur in all dynamic resizings of the table is at most $m - 1$. In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

Incremental resizing

Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually:

- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move r elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least $(r + 1)/r$ during resizing.

Disk-based hash tables almost always use some scheme of incremental resizing, since the cost of rebuilding the entire table on disk would be too high.

Monotonic keys

If it is known that key values will always increase (or decrease) monotonically, then a variation of consistent hashing can be achieved by keeping a list of the single most recent key value at each hash table resize operation. Upon lookup, keys that fall in the ranges defined by these list entries are directed to the appropriate hash function—and indeed hash table—both of which can be different for each range. Since it is common to grow the overall number of entries by doubling, there will only be $O(\log(N))$ ranges to check, and binary search time for the redirection would be $O(\log(\log(N)))$. As with consistent hashing, this approach guarantees that any key's hash, once issued, will never change, even when the hash table is later grown.

Other solutions

Linear hashing is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible lookup functions.

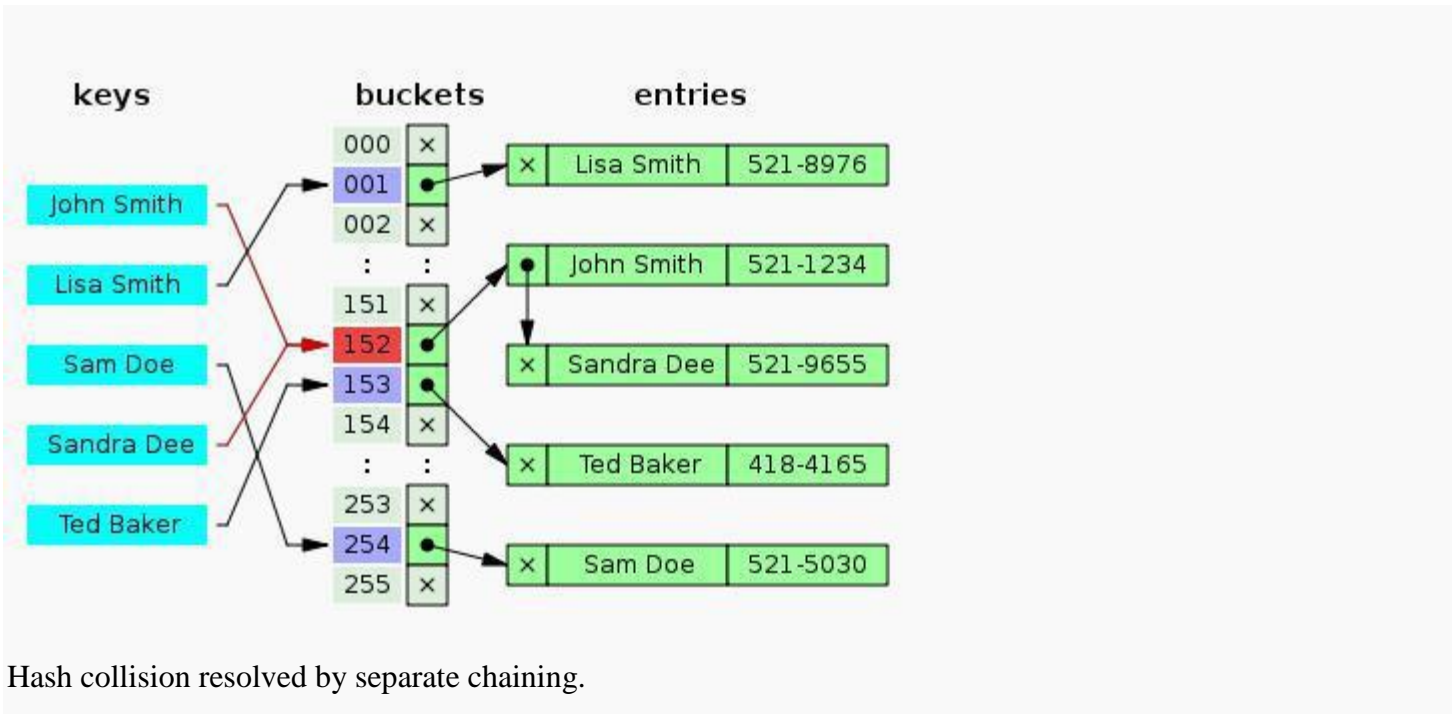
Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hash tables, where rehashing is prohibitively costly.

4.4. Collision-Resolution Techniques

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into a million buckets, even with a perfectly uniform random distribution, according to the birthday problem there is approximately a 95% chance of at least two of the keys being hashed to the same slot.

Therefore, almost all hash table implementations have some collision resolution strategy to handle such events. Some common strategies are described below. All these methods require that the keys (or pointers to them) be stored in the table, together with the associated values.

Separate chaining



In the method known as *separate chaining*, each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. Therefore, structures that are efficient in time and space for these cases are preferred. Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable. If these cases happen often, the hashing function needs to be fixed.

Separate chaining with linked lists

Chained hash tables with linked lists are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods.

The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the *average* cost of a lookup depends only on the average number of keys per bucket—that is, it is roughly proportional to the load factor.

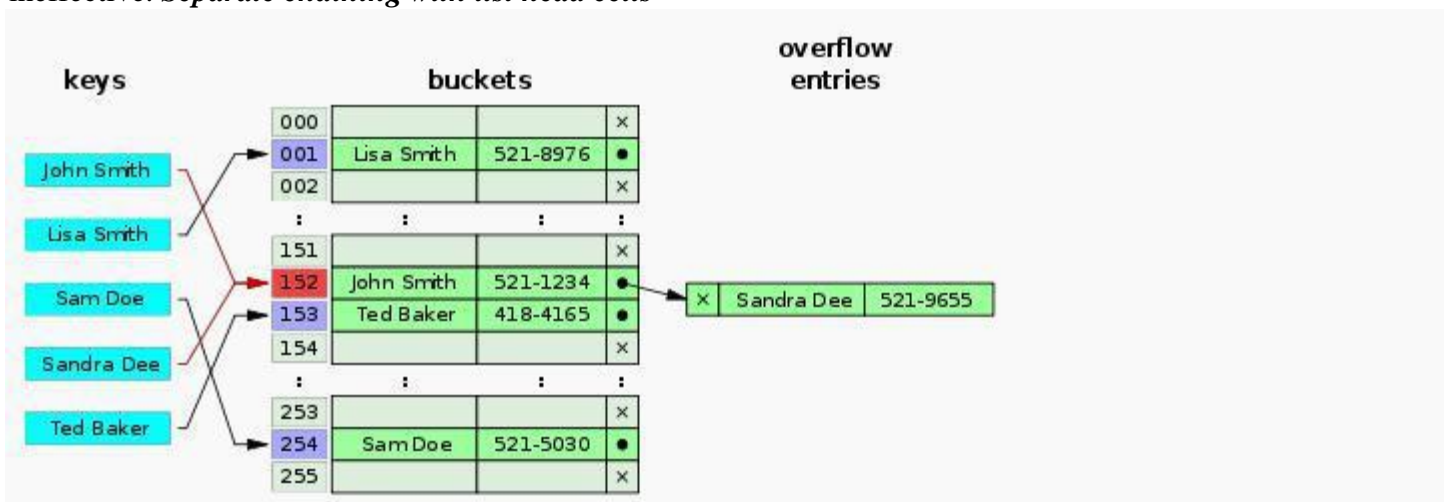
For this reason, chained hash tables remain effective even when the number of table entries n is much higher than the number of slots. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list.

For separate-chaining, the worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number n of entries in the table.

The bucket chains are often searched sequentially using the order the entries were added to the bucket. If the load factor is large and some keys are more likely to come up than others, then rearranging the chain with a move-to-front heuristic may be effective. More sophisticated data structures, such as balanced search trees, are worth considering only if the load factor is large (about 10 or more), or if the hash distribution is likely to be very non-uniform, or if one must guarantee good performance even in a worst-case scenario. However, using a larger table and/or a better hash function may be even more effective in those cases.

Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor cache performance, making the processor cache

ineffective. *Separate chaining with list head cells*



Hash collision by separate chaining with head records in the bucket array.

Some chaining implementations store the first record of each chain in the slot array itself. The number of pointer traversals is decreased by one for most cases. The purpose is to increase cache efficiency of hash table access.

The disadvantage is that an empty bucket takes the same space as a bucket with one entry. To save space, such hash tables often have about as many slots as stored entries, meaning that many slots have two or more entries.

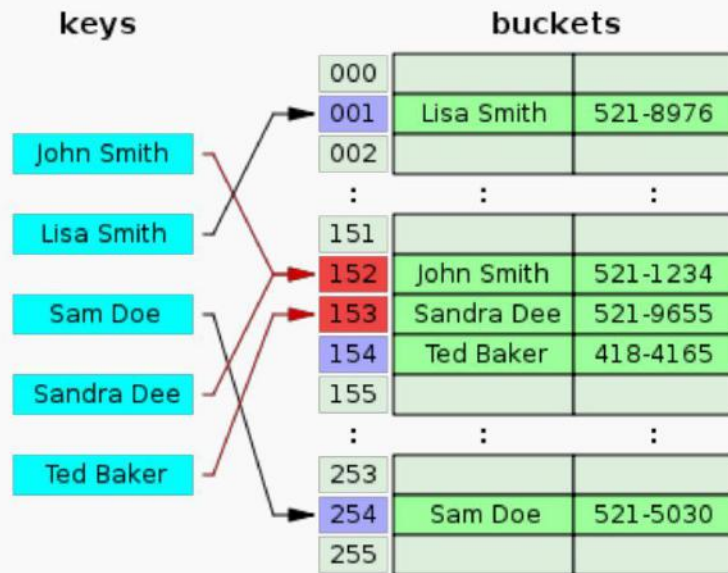
Separate chaining with other structures

Instead of a list, one can use any other data structure that supports the required operations. For example, by using a self-balancing tree, the theoretical worst-case time of common hash table operations (insertion, deletion, lookup) can be brought down to $O(\log n)$ rather than $O(n)$. However, this approach is only worth the trouble and extra memory cost if long delays must be avoided at all costs (e.g., in a real-time application), or if one must guard against many entries hashed to the same slot (e.g., if one expects extremely non-uniform distributions, or in the case of web sites or other publicly accessible services, which are vulnerable to malicious key distributions in requests).

The variant called array hash table uses a dynamic array to store all the entries that hash to the same slot. Each newly inserted entry gets appended to the end of the dynamic array that is assigned to the slot. The dynamic array is resized in an *exact-fit* manner, meaning it is grown only by as many bytes as needed. Alternative techniques such as growing the array by block sizes or *pages* were found to improve insertion performance, but at a cost in space. This variation makes more efficient use of CPU caching and the translation lookaside buffer (TLB), because slot entries are stored in sequential memory positions. It also dispenses with the next pointers that are required by linked lists, which saves space. Despite frequent array resizing, space overheads incurred by the operating system such as memory fragmentation were found to be small.

An elaboration on this approach is the so-called dynamic perfect hashing, where a bucket that contains k entries is organized as a perfect hash table with k^2 slots. While it uses more memory (n^2 slots for n entries, in the worst case and $n \times k$ slots in the average case), this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion. It is also possible to use a fusion tree for each bucket, achieving constant time for all operations with high probability.

Open addressing



Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith".

In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.^[13] The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called **closed hashing**; it should not be confused with "open hashing" or "closed addressing" that usually mean separate chaining.)

Well-known probe sequences include:

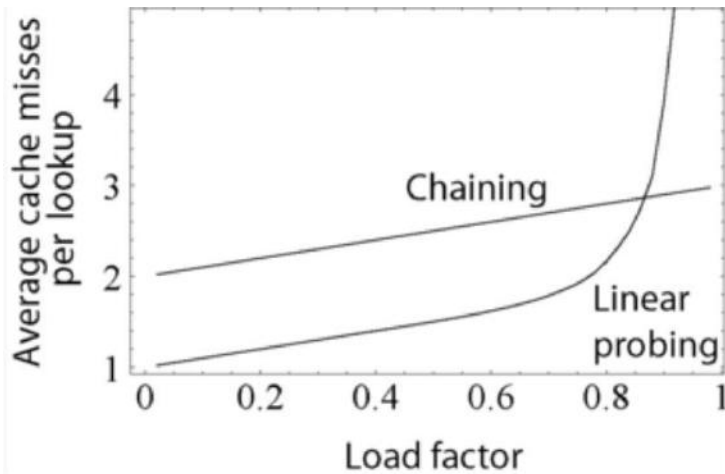
- ☐ Linear probing, in which the interval between probes is fixed (usually 1)
- ☐ Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- ☐ Double hashing, in which the interval between probes is computed by a second hash function

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are

consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the *same* hash value; whether they are adjacent or nearby is completely irrelevant.

Open addressing only saves memory if the entries are small (less than four times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.



This graph compares the average number of cache misses required to look up elements in tables with chaining and linear probing. As the table passes the 80%-full mark, linear probing's performance drastically degrades.

Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (that is, usually the only one). It also has better locality of reference, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups. Hash tables with open addressing are also easier to serialize, because they do not use pointers.

On the other hand, normal open addressing is a poor choice for large elements, because these elements fill entire CPU cache lines (negating the cache advantage), and a large amount of space is wasted on large empty table slots. If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records that can be stored within the table (internal storage) and fit in a cache line. They are particularly suitable for elements of oneword or less. If the table is expected to have a high load factor, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

Ultimately, used sensibly, any kind of hash table algorithm is usually fast *enough*; and the percentage of a calculation spent in hash table code is low. Memory usage is rarely considered excessive. Therefore, in most cases the differences between these algorithms are marginal, and other considerations typically come into play.

Coalesced hashing

A hybrid of chaining and open addressing, coalesced hashing links together chains of nodes within the table itself.^[13] Like open addressing, it achieves space usage and (somewhat diminished) cache advantages over chaining. Like chaining, it does not exhibit clustering effects; in fact, the table can be efficiently filled to a high density. Unlike chaining, it cannot have more elements than table slots.

Cuckoo hashing

Another alternative open-addressing solution is cuckoo hashing, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions. It uses two or more hash functions, which means any key/value pair could be in two or more locations. For lookup, the first hash function is used; if the key/value is not found, then the second hash function is used, and so on. If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets, at which point the table is resized. By combining multiple hash functions with multiple cells per bucket, very high space utilization can be achieved.

Hopscotch hashing

Another alternative open-addressing solution is hopscotch hashing, which combines the approaches of cuckoo hashing and linear probing, yet seems in general to avoid their limitations. In particular it works well even when the load factor grows beyond 0.9. The algorithm is well suited for implementing a resizable concurrent hash table.

The hopscotch hashing algorithm works by defining a neighborhood of buckets near the original hashed bucket, where a given entry is always found. Thus, search is limited to the number of entries in this neighborhood, which is logarithmic in the worst case, constant on average, and with proper alignment of the neighborhood typically requires one cache miss. When inserting an entry, one first attempts to add it to a bucket in the neighborhood. However, if all buckets in this neighborhood are occupied, the algorithm traverses buckets in sequence until an open slot (an unoccupied bucket) is found (as in linear probing). At that point, since the empty bucket is outside the neighborhood, items are repeatedly displaced in a sequence of hops. (This is similar to cuckoo hashing, but with the difference that in this case the empty slot is being moved into the neighborhood, instead of items being moved out with the hope of eventually finding an empty slot.) Each hop brings the open slot closer to the original neighborhood, without invalidating the neighborhood property of any of the buckets along the way. In the end, the open slot has been moved into the neighborhood, and the entry being inserted can be added to it.

Robin Hood hashing

One interesting variation on double-hashing collision resolution is Robin Hood hashing. The idea is that a new key may displace a key already inserted, if its probe count is larger than that of the key at the current position.

The net effect of this is that it reduces worst case search times in the table. This is similar to ordered hash tables except that the criterion for bumping a key does not depend on a direct relationship between the keys. Since both the worst case and the variation in the number of probes is reduced dramatically, an interesting variation is to probe the table starting at the expected successful probe value and then expand from that position in both directions. External Robin Hood hashing is an extension of this algorithm where the table is stored in an external file and each table position corresponds to a fixed-sized page or bucket with B records.

2-choice hashing

2-choice hashing employs two different hash functions, $h_1(x)$ and $h_2(x)$, for the hash table. Both hash functions are used to compute two table locations. When an object is inserted in the table, then it is placed in the table location that contains fewer objects (with the default being the $h_1(x)$ table location if there is equality in bucket size). 2-choice hashing employs the principle of the power of two choices.

5. File Structures

Introduction

This chapter is mainly concerned with the way in which file structures are used in document retrieval. Most surveys of file structures address themselves to applications in data management which is reflected in the terminology used to describe the basic concepts. I shall (on the whole) follow Hsiao and Harary whose terminology is perhaps slightly non-standard but emphasises the logical nature of file structures. A further advantage is that it enables me to bridge the gap between data management and document retrieval easily. A few other good references on file structures are Roberts, Bertziss, Dodd, and Climenson.

Logical or physical organisation and data independence

There is one important distinction that must be made at the outset when discussing file structures. And that is the difference between the *logical* and *physical* organisation of the data. On the whole a file structure will specify the logical structure of the data, that is the relationships that will exist between data items independently of the way in which these relationships may actually be realised within any computer. It is this logical aspect that we will concentrate on. The physical organisation is much more concerned with optimising the use of the storage medium when a particular logical structure is stored on, or in it. Typically for every unit of physical store there will be a number of units of the logical structure (probably records) to be stored in it. For example, if we were to store a tree structure on a magnetic disk, the physical organisation would be concerned with the best way of packing the nodes of the tree on the disk given the access characteristics of the disk.

The work on data bases has been very much concerned with a concept called *data independence*. The aim of this work is to enable programs to be written independently of the logical structure of the data they would interact with. The independence takes the following form, should the file structure overnight be changed from an inverted to a serial file the program should remain unaffected. This independence is achieved by interposing a *data model* between the user and the data base. The user sees the data model rather than the data base, and all his programs communicate with the model. The user therefore has no interest in the structure of the file.

There is a school of thought that says that applications in library automation and information retrieval should follow this path as well. And so it should. Unfortunately, there is still much debate about what a good data model should look like. Furthermore, operational implementations of some of the more advanced theoretical systems do not exist yet. So any suggestion that an IR system might be implemented through a data base package should still seem premature. Also, the scale of the problems in IR is such that efficient implementation of the application still demands close scrutiny of the file structure to be used.

Nevertheless, it is worth taking seriously the trend away from user knowledge of file structures, a trend that has been stimulated considerably by attempts to construct a theory of data. There are a number of proposals for dealing with data at an abstract level. The best known of these by now is the one put forward by Codd, which has become known as the relational model. In it data are described by n -tuples of attribute values. More formally if the data is described by *relations*, a relation on a set of *domains* D_1, \dots, D_n can be represented by a set of ordered n -tuples each of the form (d_1, \dots, d_n) where d_i \llbracket propersubset $\rrbracket D_i$. As it is rather difficult to cope with general relations, various levels (three in fact) of normalisation have been introduced restricting the kind of relations allowed.

A second approach is the *hierarchical* approach. It is used in many existing data base systems. This approach works as one might expect: data is represented in the form of hierarchies. Although it is more restrictive than the relational approach it often seems to be the natural way to proceed. It can be argued that in many applications a hierarchic structure is a good approximation to the natural structure in the data, and that the resulting loss in precision of representation is worth the gain in efficiency and simplicity of representation.

The third approach is the *network* approach associated with the proposals by the Data Base Task Group of CODASYL. Here data items are linked into a network in which any given link between two items exists because it satisfies some condition on the attributes of those items, for example, they share an attribute. It is more general than the hierarchic approach in the sense that a node can have any number of immediate superiors. It is also equivalent to the relational approach in descriptive power.

The whole field of data base structures is still very much in a state of flux. The advantages and disadvantages of each approach are discussed very thoroughly in Date, who also gives excellent annotated citations to the current literature. There is also a recent *Computing Survey*[11] which reviews the current state of the art. There have been some very early proponents of the relational approach in IR, as early as 1967 Maron and Levien discussed the design and implementation of an IR system via relations, be it binary ones. Also Prywes and Smith in their review chapter in the *Annual Review of Information Science and Technology* more recently recommended the DBTG proposals as ways of implementing IR systems.

Lurking in the background of any discussion of file structures nowadays is always the question whether data base technology will overtake all. Thus it may be that any application in the field of library automation and information retrieval will be implemented through the use of some appropriate data base package. This is certainly a possibility but not likely to happen in the near future. There are several reasons. One is that data base systems are *general* purpose systems whereas automated library and retrieval systems are *special* purpose. Normally one pays a price for generality and in this case it is still too great. Secondly, there now is a considerable investment in providing special purpose systems (for example, MARC) and this is not written off very easily. Nevertheless a trend towards increasing use of data-base technology exists and is well illustrated by the increased prominence given to it in the *Annual Review of Information Science and Technology*.

A language for describing file structures

Like all subjects in computer science the terminology of file structures has evolved higgledy-piggledy without much concern for consistency, ambiguity, or whether it was possible to make the kind of distinctions that were important. It was only much later that the need for a well-defined, unambiguous language to describe file structures became apparent. In particular, there arose a need to communicate ideas about file structures without getting bogged down by hardware considerations.

This section will present a formal description of file structures. The framework described is important for the understanding of any file structure. The terminology is based on that introduced by Hsiao and Harary (but also see Hsiao and Manola and Hsiao). Their terminology has been modified and extended by Severance, a summary of this can be found in van Rijsbergen. Jonkers has formalised a different framework which provides an interesting contrast to the one described here.

Basic terminology

Given a set of 'attributes' A and a set of 'values' V , then a *record* R is a subset of the cartesian product $A \times V$ in which each attribute has one and only one value. Thus R is a set of ordered pairs of the form (an attribute, its value). For example, the record for a document which has been processed by an automatic content analysis algorithm would be

$$R = \{(K1, x1), (K2, x2), \dots (Km, xm)\}$$

The K_i 's are keywords functioning as attributes and the value x_i can be thought of as a numerical weight. Frequently documents are simply characterised by the absence or presence of keywords, in which case we write

$$R = \{Kt1, Kt2, \dots, Kti\}$$

where Kti is present if $x_{ti} = 1$ and is absent otherwise.

Records are collected into logical units called files. They enable one to refer to a set of records by name, the file name. The records within a file are often organised according to relationships between the records. This logical organisation has become known as a file structure (or data structure).

It is difficult in describing file structures to keep the logical features separate from the physical ones. The latter are characteristics forced upon us by the recording media (e.g. tape, disk). Some features can be defined abstractly (with little gain) but are more easily understood when illustrated concretely. One such feature is a *field*. In any implementation of a record, the attribute values are usually positional, that is the identity of an attribute is given by the position of its attribute value within the record. Therefore the data within a record is registered sequentially and has a definite beginning and end. The record is said to be divided into *fields* and the n th field carries the n th attribute value. Pictorially we have an example of a record with associated fields in *Figure 4.1*.

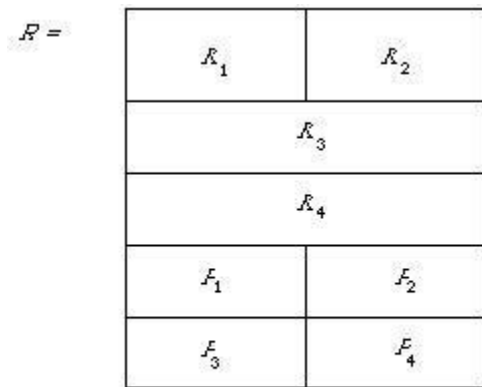


Figure 4.1. An example of a record with associated fields.

The fields are not necessarily constant in length. To find the value of the attribute K_4 , we first find the address of the record R (which is actually the address of the start of the record) and read the data in the 4th field.

In the same picture I have also shown some fields labelled P_i . They are addresses of other records, and are commonly called pointers. Now we have extended the definition of a record to a set of attribute-value pairs and pointers. Each pointer is usually associated with a particular attribute-value pair. For example, (see Figure 4.2) pointers could be used to link all records for which the value x_1 (of attribute K_1) is a , similarly for x_2 equal to b , etc.

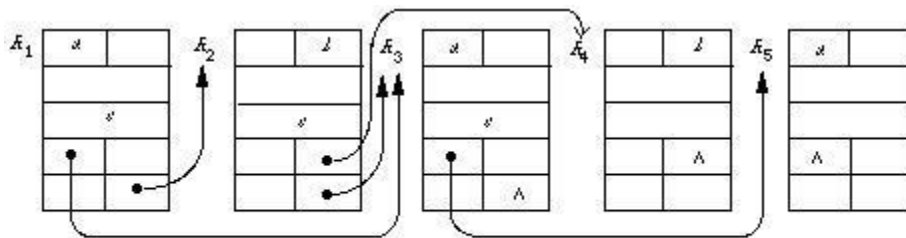


Figure 4.2. A demonstration of the use of pointers to link records.

To indicate that a record is the last record pointed to in a list of records we use the *null pointer* [[logicaland]]. The pointer associated with attribute K in record R will be called a *K-pointer*. An attribute (keyword) that is used in this way to organise a file is called a *key*.

To unify the discussion of file structures we need some further concepts. Following Hsiao and Harary again, we define a list L of records with respect to a keyword K , or more briefly a *K-list* as a set of records containing K such that:

- (1) the *K*-pointers are distinct;
- (2) each non-null *K*-pointer in L gives the address of a record within L ;
- (3) there is a unique record in L not pointed to by any record containing K ; it is called the *beginning* of the list; and

(4) there is a unique record in L containing the null K -pointer; it is the *end* of the list.

(Hsiao and Harary state condition (2) slightly differently so that no two K -lists have a record in common; this only appears to complicate things.)

From our previous example:

$K1$ -list : $R1, R2, R5$

$K2$ -list : $R2, R4$

$K4$ -list : $R1, R2, R3$

Finally, we need the definition of a *directory* of a file. Let F be a file whose records contain just m different keywords $K1, K2, \dots, Km$. Let n_i be the number of records containing the keyword K_i , and h_i be the number of K_i -lists in F . Furthermore, we denote by aij the beginning address of the j th K_i -list. Then the *directory* is the set of sequences

$(K_i, n_i, h_i, ai1, ai2, \dots, aih_i) \ i = 1, 2, \dots, m$

We are now in a position to give a unified treatment of sequential files, inverted files, index-sequential files and multi-list files.

Sequential files

A sequential file is the most primitive of all file structures. It has no directory and no linking pointers. The records are generally organised in lexicographic order on the value of some key. In other words, a particular attribute is chosen whose value will determine the order of the records. Sometimes when the attribute value is constant for a large number of records a second key is chosen to give an order when the first key fails to discriminate.

The implementation of this file structure requires the use of a sorting routine.

Its main advantages are:

- (1) it is easy to implement;
- (2) it provides fast access to the next record using lexicographic order.

Its disadvantages:

- (1) it is difficult to update - inserting a new record may require moving a large proportion of the file;
- (2) random access is extremely slow.

Sometimes a file is considered to be sequentially organised despite the fact that it is not ordered according to any key. Perhaps the date of acquisition is considered to be the key value, the newest entries are added to the end of the file and therefore pose no difficulty to updating.

Inverted files

The importance of this file structure will become more apparent when Boolean Searches are discussed in the next chapter. For the moment we limit ourselves to describing its structure.

An *inverted file* is a file structure in which every list contains only one record. Remember that a list is defined with respect to a keyword K , so every K -list contains only one record. This implies that the directory will be such that $ni = hi$ for all i , that is, the number of records containing K_i will equal the number of K_i -lists. So the directory will have an address for each record containing K_i . For document retrieval this means that given a keyword we can immediately locate the addresses of all the documents containing that keyword. For the previous example let us assume that a non-black entry in the field corresponding to an attribute indicates the presence of a keyword and a black entry its absence. Then the directory will point to the file in the way shown in Figure 4.3. The definition of an inverted file does *not* require that the addresses in the directory are in any order. However, to facilitate operations such as conjunction ('and') and disjunction ('or') on any two inverted lists, the addresses are normally kept in record number order. This means that 'and' and 'or' operations can be performed with one pass through both lists. The penalty we pay is of course that the inverted file becomes slower to update.

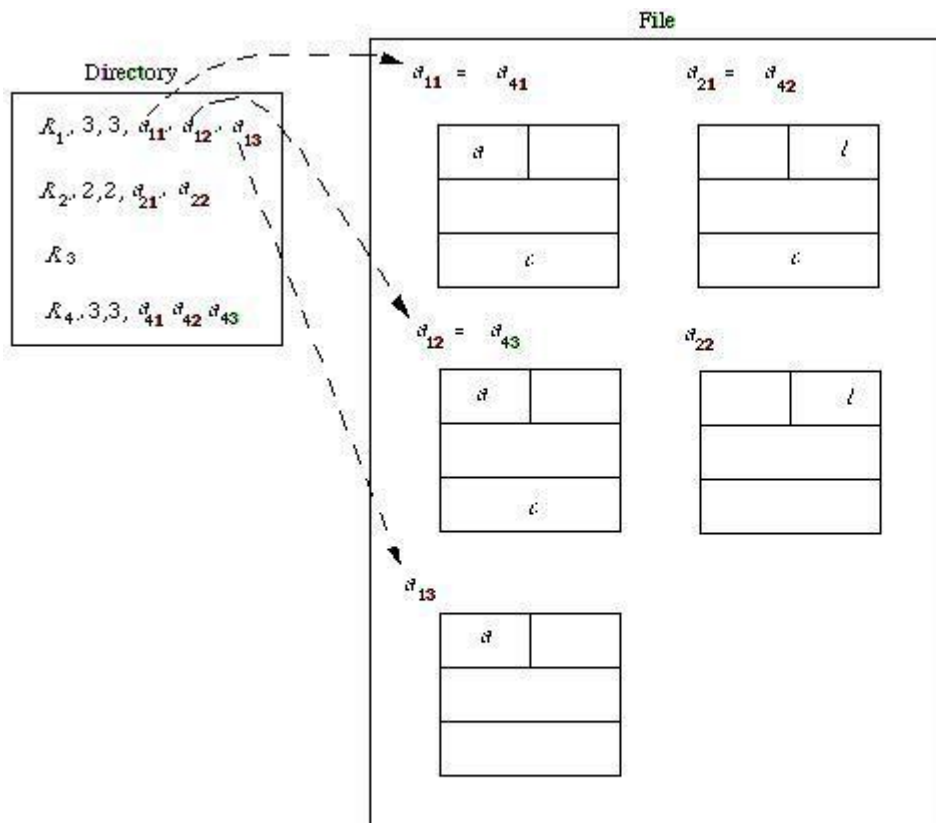


Figure 4.3. An inverted file

Index-sequential files

An index-sequential file is an inverted file in which for every keyword K_i , we have $n_i = h_i = 1$ and $a_{i1} < a_{21} \dots < a_{m1}$. This situation can only arise if each record has just one unique keyword, or one unique attribute-value. In practice therefore, this set of records may be ordered sequentially by a key. Each key value appears in the directory with the associated address of its record. An obvious interpretation of a key of this kind would be the record number. In our example none of the attributes would do the job except the record number. Diagrammatically the index-sequential file would therefore appear as shown in Figure 4.4. I have deliberately written R_i instead of K_i to emphasise the nature of the key.

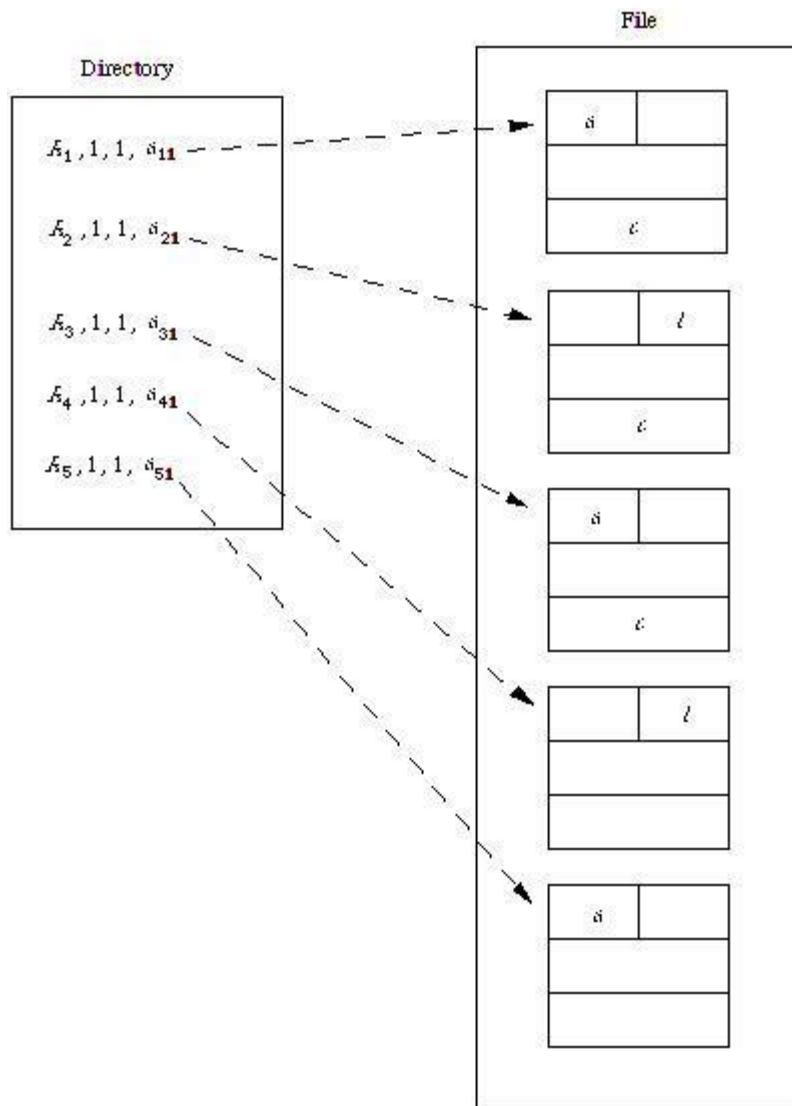


Figure 4.4. An index-sequential file

In the literature an index-sequential file is usually thought of as a sequential file with a hierarchy of indices. This does not contradict the previous definition, it merely describes the way in which the directory is implemented. It is not surprising therefore that the indexes ('index' = 'directory' here) are often oriented to the characteristics of the storage medium. For example (see *Figure 4.5*) there might be three levels of indexing:

track, cylinder and master. Each entry in the track index will contain enough information to locate the start of the track, and the key of the last record in the track which is also normally the highest value on that track. There is a track index for each cylinder. Each entry in the cylinder index gives the last record on each cylinder and the address of the track index for that cylinder. If the cylinder index itself is stored on tracks, then the master index will give the highest key referenced for each track of the cylinder index and the starting address of that track.

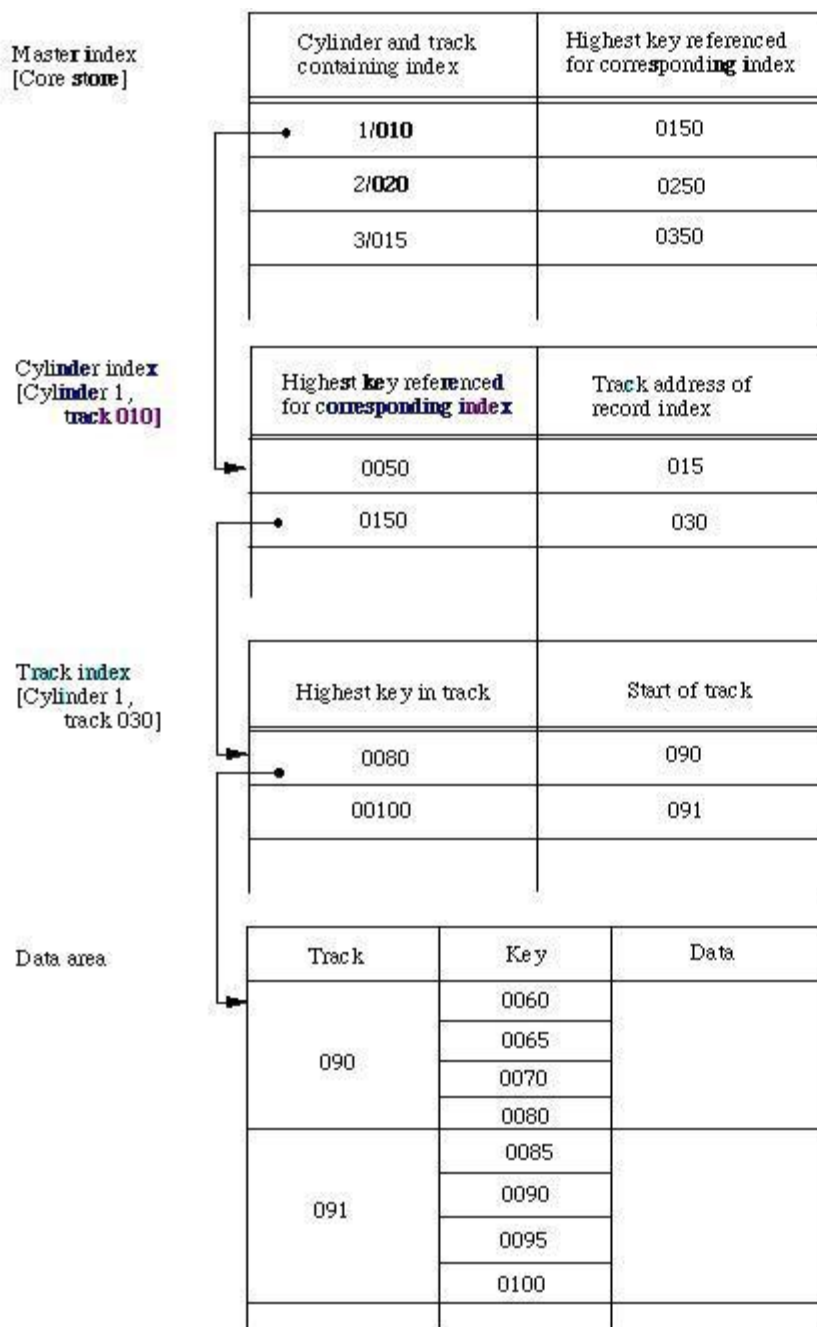


Figure 4.5: An example of an implementation of an index-sequential file (Adapted from D. R. Judd, *The Use of Files*, Macdonald and Elsevier, London and New York 1973, page 46.)

No mention has been made of the possibility of overflow during an updating process. Normally provision is made in the directory to administer an overflow area. This of course increases the number of book-keeping entries in each entry of the index.

Multi-lists

A multi-list is really only a slightly modified inverted file. There is one list per keyword, i.e. $hi = 1$. The records containing a particular keyword K_i are chained together to form the K_i -list and the start of the K_i -list is given in the directory, as illustrated in *Figure 4.6*. Since there is no K_3 -list, the field reserved for its pointer could well have been omitted. So could any blank pointer field, so long as no ambiguity arises as to which pointer belongs to which keyword. One way of ensuring this, particularly if the data values (attribute-values) are fixed format, is to have the pointer not pointing to the beginning of the record but pointing to the location of the next pointer in the chain.

The multi-list is designed to overcome the difficulties of updating an inverted file. The addresses in the directory of an inverted file are normally kept in record-number order. But, when the time comes to add a new record to the file, this sequence must be maintained, and inserting the new address can be expensive. No such problem arises with the multi-list, we update the appropriate K -lists by simply chaining in the new record. The penalty we pay for this is of course the increase in search time. This is in fact typical of many of the file structures. Inherent in their design is a trade-off between search time and update time.

Cellular multi-lists

A further modification of the multi-list is inspired by the fact that many storage media are divided into *pages*, which can be retrieved one at a time. A K -list may cross several page boundaries which means that several pages may have to be accessed to retrieve one record. A modified multi-list structure which avoids this is called a *cellular multi-list*. The K -lists are limited so that they will not cross the page (cell) boundaries.

At this point the full power of the notation introduced before comes into play. The directory for a cellular multi-list will be the set of sequences

$$(K_i, n_i, h_i, a_{i1}, \dots, a_{ih_i}) \quad i = 1, 2, \dots, m$$

where the h_i have been picked to ensure that a K_i -list does not cross a page boundary. In an implementation, just as in the implementation of an index-sequential file, further information will be stored with each address to enable the right page to be located for each key value.

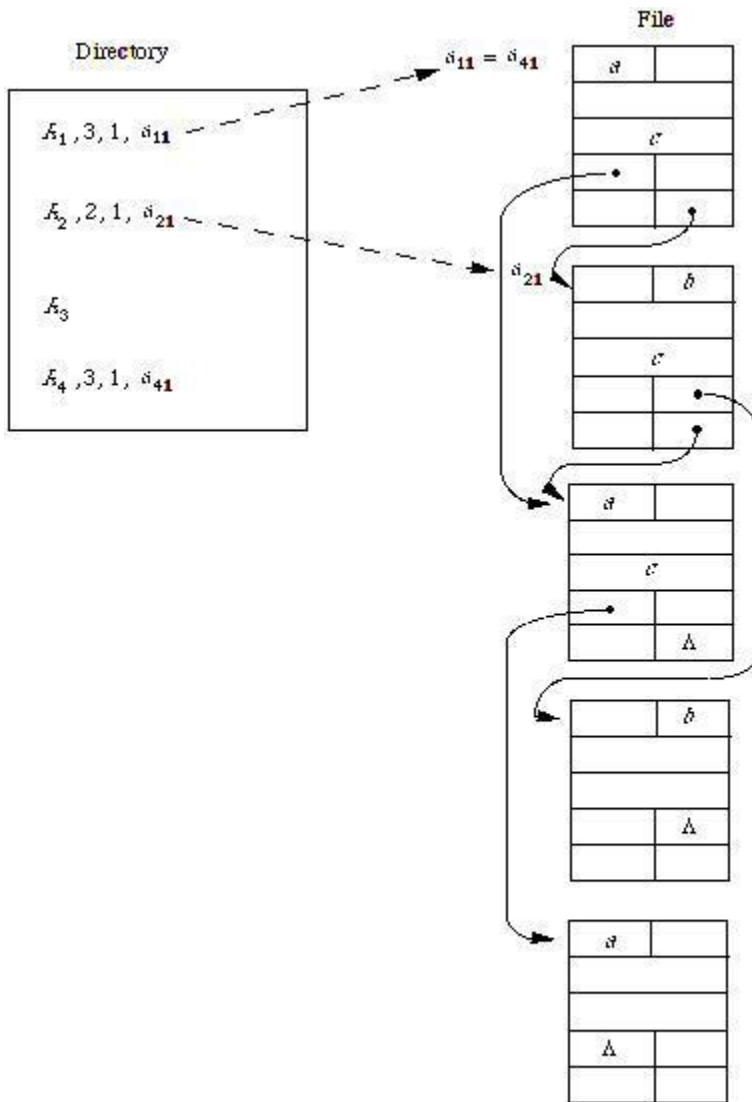


Figure 4.6. A multi-list

Ring structures

A *ring* is simply a linear list that closes upon itself. In terms of the definition of a *K*-list, the beginning and end of the list are the same record. This data-structure is particularly useful to show classification of data.

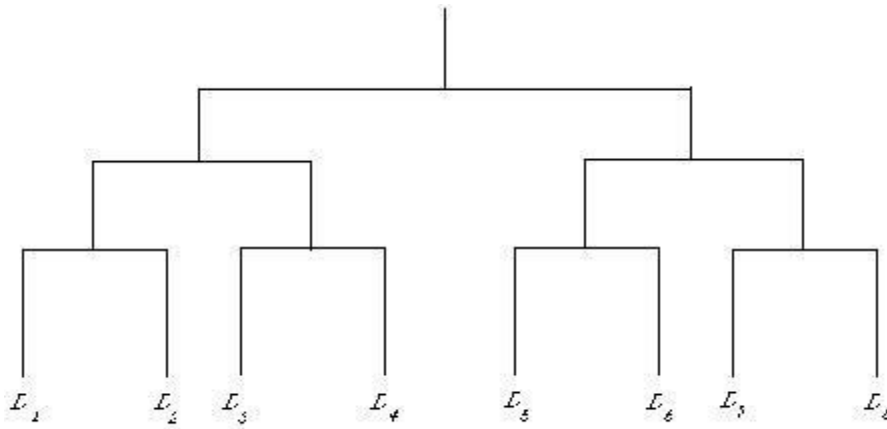


Figure 4.7. A dendrogram.

Let us suppose that a set of documents

$\{D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8\}$

has been classified into four groups, that is

$\{(D_1, D_2), (D_3, D_4), (D_5, D_6), (D_7, D_8)\}$

Furthermore these have themselves been classified into two groups,

$\{((D_1, D_2), (D_3, D_4)), ((D_5, D_6), (D_7, D_8))\}$

The dendrogram for this structure would be that given in Figure 4.7. To represent this in storage by means of ring structures is now a simple matter (see Figure 4.8).

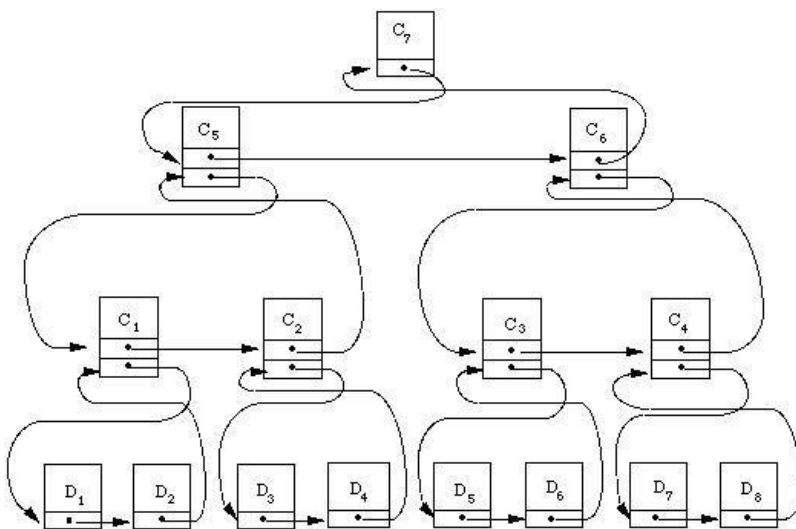


Figure 4.8. An implementation of a dendrogram via ring structures.

The D_i indicates a description (representation) of a document. Notice how the rings at a lower level are contained in those at a higher level. The field marked C_i normally contains some identifying information with respect to the ring it subsumes. For example, C_1 in some way identifies the class of documents $\{D_1, D_2\}$.

Were we to group documents according to the keywords they shared, then for each keyword we would have a group of documents, namely, those which had that keyword in common. C_i would then be the field containing the keyword uniting that particular group. The rings would of course overlap (*Figure 4.9*), as in this example:

$$D_1 = \{K_1, K_2\}$$

$$D_2 = \{K_2, K_3\}$$

$$D_3 = \{K_1, K_4\}$$

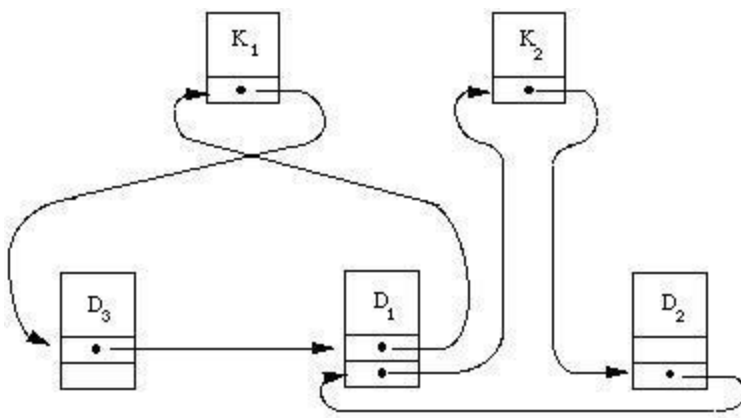


Figure 4.9. Two overlapping rings

The usefulness of this kind of structure will become more apparent when we discuss searching of classifications. If each ring has associated with it a record which contains identifying information for its members, then, a search strategy searching a structure such as this will first look at C_i (or K_i in the second example) to determine whether to proceed or abandon the search.

Threaded lists

In this section an elementary knowledge of list processing will be assumed. Readers who are unfamiliar with this topic should consult the little book by Foster.

A simple list representation of the classification

$$((D_1, D_2), (D_3, D_4)), ((D_5, D_6), (D_7, D_8))$$

is given in *Figure 4.10*. Each sublist in this structure has associated with it a record containing *only* two pointers. (We can assume that D_i is really a pointer to document D_i .) The function of the pointers should be clear from the diagram. The main thing to note, however, is that the record associated with a list does *not* contain any identifying information.

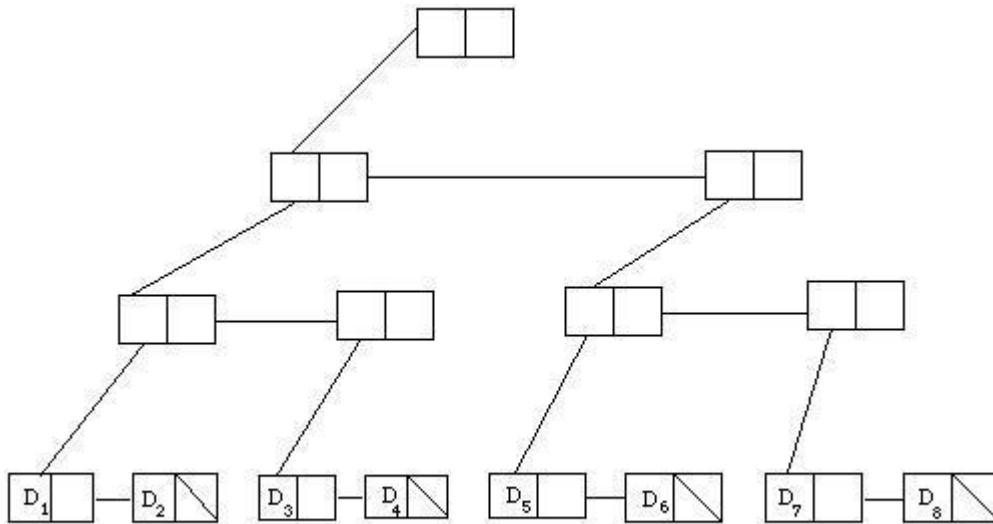


Figure 4.10. A list structure implementation of a hierarchical classification.

A modification of the implementation of a list structure like this which makes it resemble a set of ring structures is to make the right hand pointer of the *last* element of a sublist point back to the head of the sublist. Each sublist has become effectively a ring structure. We now have what is commonly called a *threaded list* (see Figure 4.11). The representation I have given is a slight oversimplification in that we need to flag which elements are data elements (giving access to the documents D_i) and which elements are just pointer elements. The major advantage associated with a threaded list is that it can be traversed without the aid of a stack. Normally when traversing a conventional list structure the return addresses are stacked, whereas in the threaded list they have been incorporated in the data structure.

One disadvantage associated with the use of list and ring structures for representing classifications is that they can only be entered at the 'top'. An additional index giving entry to the structure at each of the data elements increases the update speed considerably.

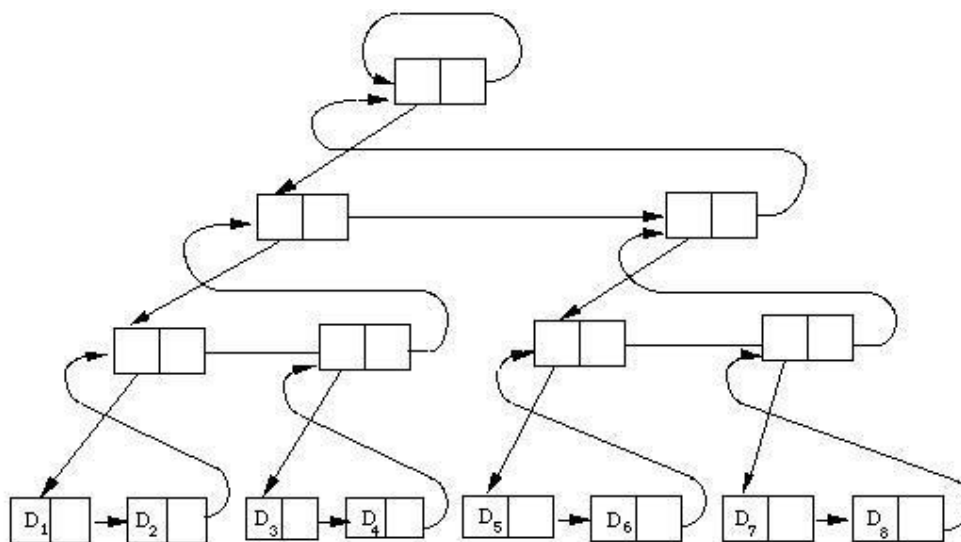
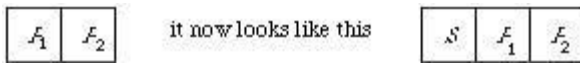


Figure 4.11. A threaded list implementation of a hierarchical classification

Another modification of the simple list representation has been studied extensively by Stanfel[21,22] and Patt[23]. The individual elements (or cells) of the list structure are modified to incorporate one extra field, so that instead of each element looking like this



where the P_i s are pointers and S is a symbol. Otherwise no essential change has been made to the simple representation. This structure has become known as the *Doubly Chained Tree*. Its properties have mainly been investigated for storing variable length keys, where each key is made up by selecting symbols from a finite (usually small) alphabet. For example, let $\{A, B, C\}$ be the set of key symbols and let R_1, R_2, R_3, R_4, R_5 be five records to be stored. Let us assign keys made of the 3 symbols, to the record as follows:

AAA R1

AB R2

AC R3

BB R4

BC R5

An example of a doubly chained tree containing the keys and giving access to the records is given in *Figure 4.12*. The topmost element contains no symbol, it merely functions as the start of the structure. Given an arbitrary key its presence or absence is detected by matching it against keys in the structure. Matching proceeds level by level, once a matching symbol has been found at one level, the P_1 pointer is followed to the set of *alternative* symbols at the next level down. The matching will terminate either:

- (1) when the key is exhausted, that is, no more key symbols are left to match; or
- (2) when no matching symbol is found at the current level.

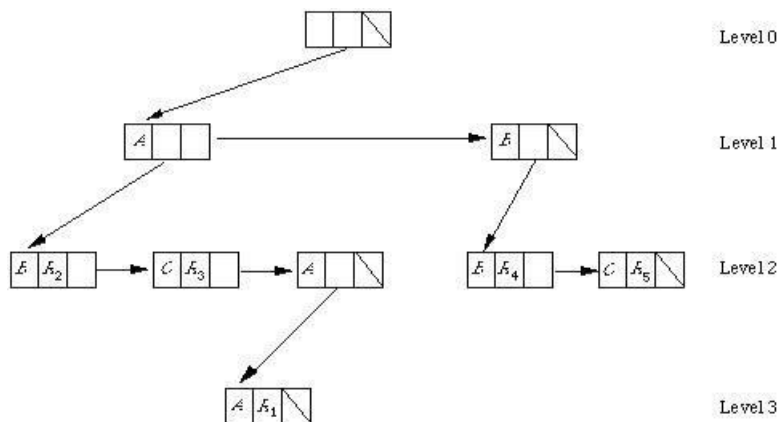


Figure 4.12. An example of a doubly chained tree.

For case (1) we have:

(a) the key is present if the P1 pointer in the same cell as the last matching symbol

now points to a record;

(b) P1 points to a further symbol, that is, the key 'falls short' and is therefore not in the structure.

For case (2), we also have that the key is not in the structure, but now there is a mismatch.

Stanfel and Patt have concentrated on generating search trees with minimum expected search time, and preserving this property despite updating. For the detailed mathematics demonstrating that this is possible the reader is referred to their cited work.