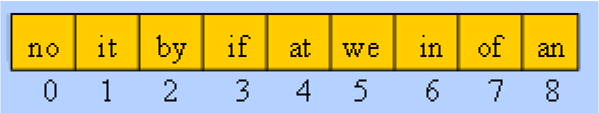
SCS1201 Advanced Data Structures

Unit V

Table Data Structures

Table is a data structure which plays a significant role in information retrieval. A set of n distinct records with keys K1, K2, …., Kn are stored in a file. If we want to find a record with a given key value, K, simply access the index given by its key k.



no	it	by	if	at	we	in	of	an
0	1	2	3	4	5	6	7	8

The table lookup has a running time of O(1). The searching time required is directly proportional to the number of number of records in the file. This searching time can be reduced, even can be made independent of the number of records, if we use a table called Access Table.

Some of possible kinds of tables are given below:

- Rectangular table
- Jagged table
- Inverted table.
- Hash tables

Rectangular Tables

Tables are very often in rectangular form with rows and columns. Most programming languages accommodate rectangular tables as 2-D arrays. Rectangular tables are also known as matrices. Almost all programming languages provide the implementation procedures for these tables as they are used in many applications.

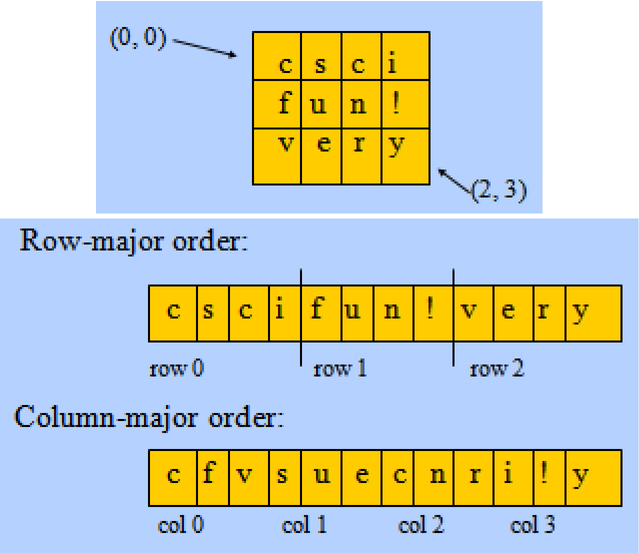


Fig. Rectangular tables in memory

Logically, a matrix appears as two-dimensional, but physically it is stored in a linear fashion. In order to map from the logical view to physical structure, an indexing formula is used.

The compiler must be able to convert the index (i, j) of a rectangular table to the correct position in the sequential array in memory.

For an m x n array (m rows, n columns):

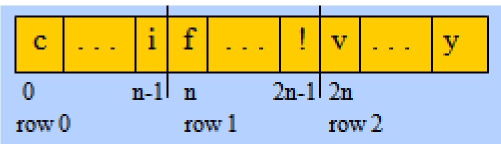
Each row is indexed from 0 to m-1
Each column is indexed from 0 to n – 1
Item at (i, j) is at sequential position i * n + j

Row major order:

Assume that the base address is the first location of the memory, so the
Address a_{ij} =storing all the elements in the first(i-1)th rows +
the number of elements in the ith row up to the jth coloumn
= (i-1)*n+j

Column major order:

Address of a_{ij} = storing all the elements in the first(j-1)th column +
The number of elements in the jth column up to the ith rows.
=(j-1)*m+i



Jagged table

Jagged tables are nothing but the special kind of sparse matrices such as triangular matrices, band matrices, etc… In the jagged tables, we put a restriction that in the row (or in a column) if elements are present then they are contiguous. Thus in fig (a) - (e), all are jagged tables except the table in fig (f).

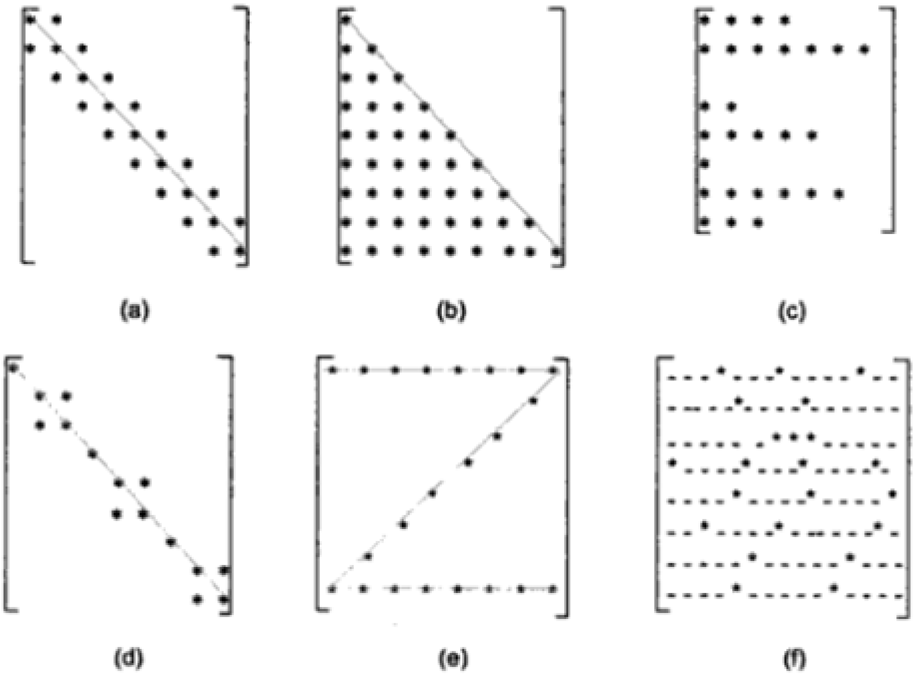


Fig. Jagged table and sparse matrices

Symmetric sparse matrices stored as one dimensional arrays can be retrieved using the indexing formula

$$\text{Address}\,(a_{ij}) = \frac{i \times (i-1)}{2} + j$$

This formula involves multiplication and division which are infact inefficient from the computational point of view. So, the alternative technique is by setting up an access table whose entries correspond to the row indices of the jagged table, such that the ith entry in the access table is

$$\frac{i \times (i-1)}{2}$$

The access table is calculated only at the time of initiation and a=can be stored in memory, it can referred each time the access of element in the jagged table occur. We can also calculate by pure addition rather than multiplication or division such as 0,1, (1+2), (1+2)+3,…

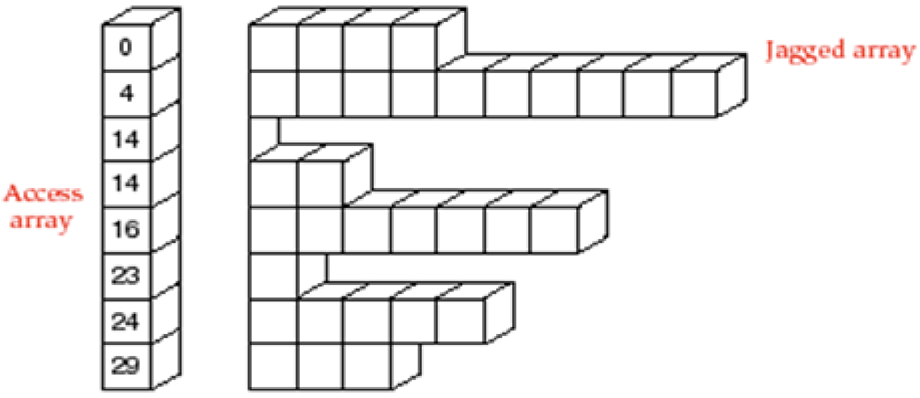
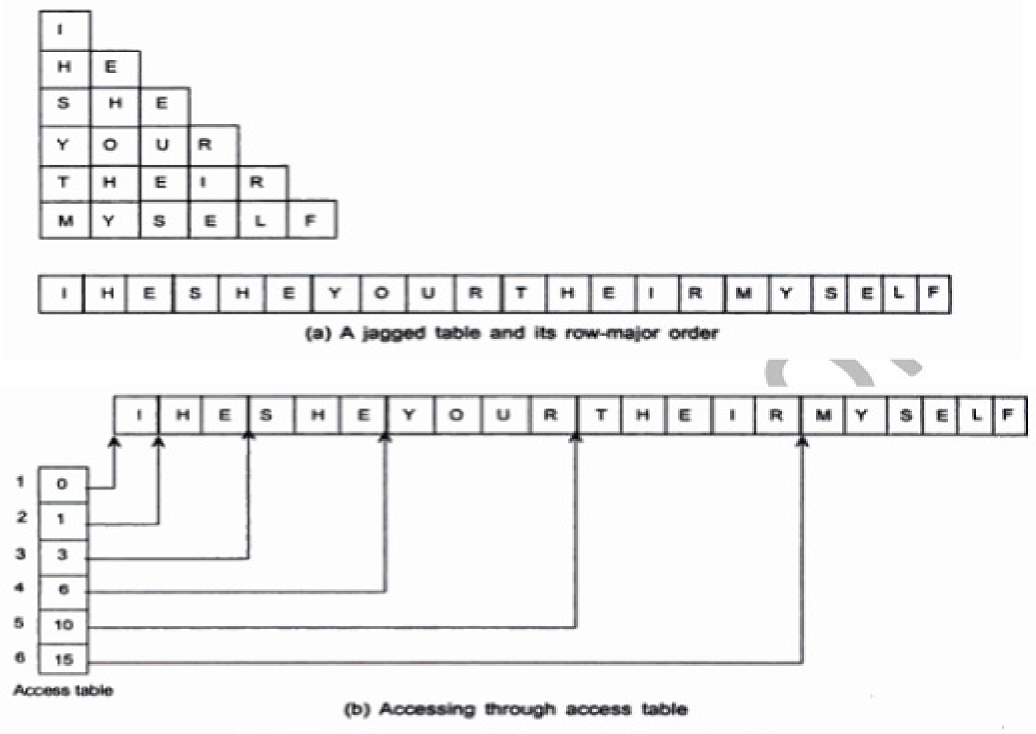


Fig. Access array for the jagged tables

For e.g., if we want to access a54 (element in 5th row and 4th column) then at 5th location of the access table, we see the entry is 10; hence desire element is at 14 (=10+4) location of the array which contains the elements. It is assumed that the 1st element of the table is located at the location of the array.

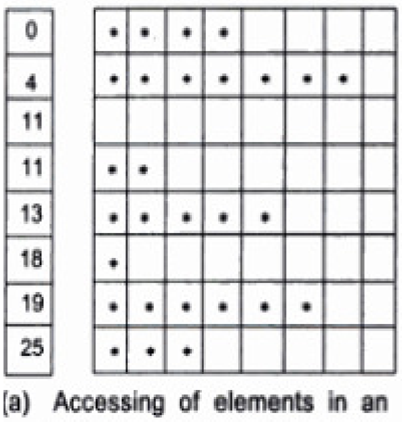


(a) A jagged table and its row-major order

(b) Accessing through access table

Access technique of a jagged table.

Above mentioned accessing technique has another advantage over the indexing formula. We can find the indexing formula even if the jagged tables are asymmetric with respect to the arrangement of elements in it. For e.g., in fig (d), it is difficult find index formula. In this case, we can easily maintain its storage in an array and can obtain faster access of elements from it. In the fig access of elements in the asymmetric matrix, its use is same as that of the symmetric sparse matrices. An entry in the ith location of the access table can be obtained by adding the number of elements in (i-1)th row of the jagged table and (i-1)th entry of the access table, assuming that entry 0 as the first entry in the access table, and as before, the starting location of the array storing the elements is 1.

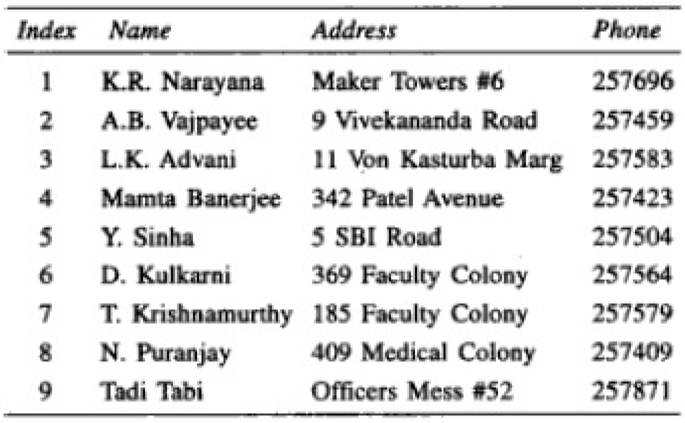


(a) Accessing of elements in an asymmetric matrix (jagged table).

Inverted tables

The concept of inverted tables can be explain with an example. Suppose, a telephone company maintains records of all the subscribers of a telephone exchange as shown in the fig given below. These records can be used to serve several purposes. One of them requires the alphabetical ordering of name of the subscriber. Second, it requires the lexicographical ordering of the address of subscriber. Third, it also requires the ascending order of the telephone numbers in order to estimate the cabling charge from the telephone exchange to the location of the telephone connection etc.... To serve these purposes, the telephone company should maintain 3 sets of records: one in alphabetical order of the NAME, second, the lexicographical ordering of the ADDRESS and third, the ascending order of the phone numbers.

(a) Records of a Telephone Exchange



Index	*Name*	*Address*	*Phone*
1	K.R. Narayana	Maker Towers #6	257696
2	A.B. Vajpayee	9 Vivekananda Road	257459
3	L.K. Advani	11 Von Kasturba Marg	257583
4	Mamta Banerjee	342 Patel Avenue	257423
5	Y. Sinha	5 SBI Road	257504
6	D. Kulkarni	369 Faculty Colony	257564
7	T. Krishnamurthy	185 Faculty Colony	257579
8	N. Puranjay	409 Medical Colony	257409
9	Tadi Tabi	Officers Mess #52	257871

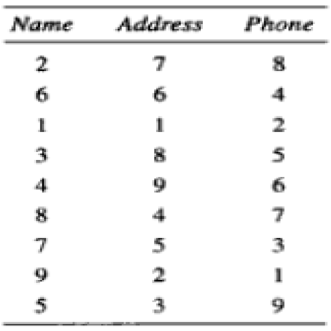
This way of maintaining records leads to the following drawbacks,

a. Requirement of extra storage: three times the actual memory

b. Difficulty in modification of records: if a subscriber changes his address, then we have to modify this in three storages to maintain consistency in information.

Using the concept of inverted tables, we can avoid the multiple set of records, and we can still retrieve the records by any of the three keys almost as quickly as if the records are fully sorted by that key. Therefore, we should maintain an3 inverted table. In this case, this table comprise of three columns: NAME, ADDRESS, and PHONE as shown in below figure. Each column contains the index numbers of records in the order based on the sorting of the corresponding key. This inverted table, therefore, can be consulted to retrieve information

(b) Inverted Table



Name	*Address*	*Phone*
2	7	8
6	6	4
1	1	2
3	8	5
4	9	6
8	4	7
7	5	3
9	2	1
5	3	9

Symbol Tables

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A **symbol table** is a major data structure used in a compiler:

- Associates **attributes** with identifiers used in a program
- For instance, a **type attribute** is usually associated with each identifier
- A symbol table is a necessary component
    - ✓ Definition (declaration) of identifiers appears once in a program
    - ✓ Use of identifiers may appear in many places of the program text
- Identifiers and attributes are entered by the analysis phases
    - When processing a definition (declaration) of an identifier
    - In simple languages with only global variables and implicit declarations:
        - ✓ The scanner can enter an identifier into a symbol table if it is not already there
    - In block-structured languages with scopes and explicit declarations:
        - ✓ The parser and/or semantic analyzer enter identifiers and corresponding attributes
- Symbol table information is used by the analysis and synthesis phases
    - To verify that used identifiers have been defined (declared)
    - To verify that expressions and assignments are semantically correct – **type checking**
    - To generate intermediate or target code

Symbol Table Interface

- The basic operations defined on a symbol table include:
    - **allocate** – to allocate a new empty symbol table
    - **free** – to remove all entries and free the storage of a symbol table
    - **insert** – to insert a name in a symbol table and return a pointer to its entry
    - **lookup** – to search for a name and return a pointer to its entry
    - **set_attribute** – to associate an attribute with a given entry
    - **get_attribute** – to get an attribute associated with a given entry
- Other operations can be added depending on requirement
    - For example, a **delete** operation removes a name previously inserted
        - ✓ Some identifiers become invisible (out of scope) after exiting a block
- This interface provides an abstract view of a symbol table
- Supports the simultaneous existence of multiple tables
- Implementation can vary without modifying the interface

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

```
<symbol name,  type,  attribute>
```

For example, if a symbol table has to store information about the following variable declaration:

```
static int interest;
```

then it should store the entry such as:

```
<interest, int, static>
```

The attribute clause contains the entries related to the name.

Implementation

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

- **Unordered List**
  - Simplest to implement
  - Implemented as an array or a linked list
  - Linked list can grow dynamically – alleviates problem of a fixed size array
  - Insertion is fast $O(1)$, but lookup is slow for large tables – $O(n)$ on average
- **Ordered List**
  - If an array is sorted, it can be searched using binary search – $O(\log_2 n)$
  - Insertion into a sorted array is expensive – $O(n)$ on average
  - Useful when set of names is known in advance – table of reserved words
- **Binary Search Tree**
  - Can grow dynamically
  - Insertion and lookup are $O(\log_2 n)$ on average

Operations

- First consideration is how to **insert** and **lookup** names Variety of implementation techniques
- A symbol table, either linear or hash, should provide the following operations.

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

```
int a;
```

should be processed by the compiler as:

```
insert(a, int);
```

lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

```
lookup(symbol)
```

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

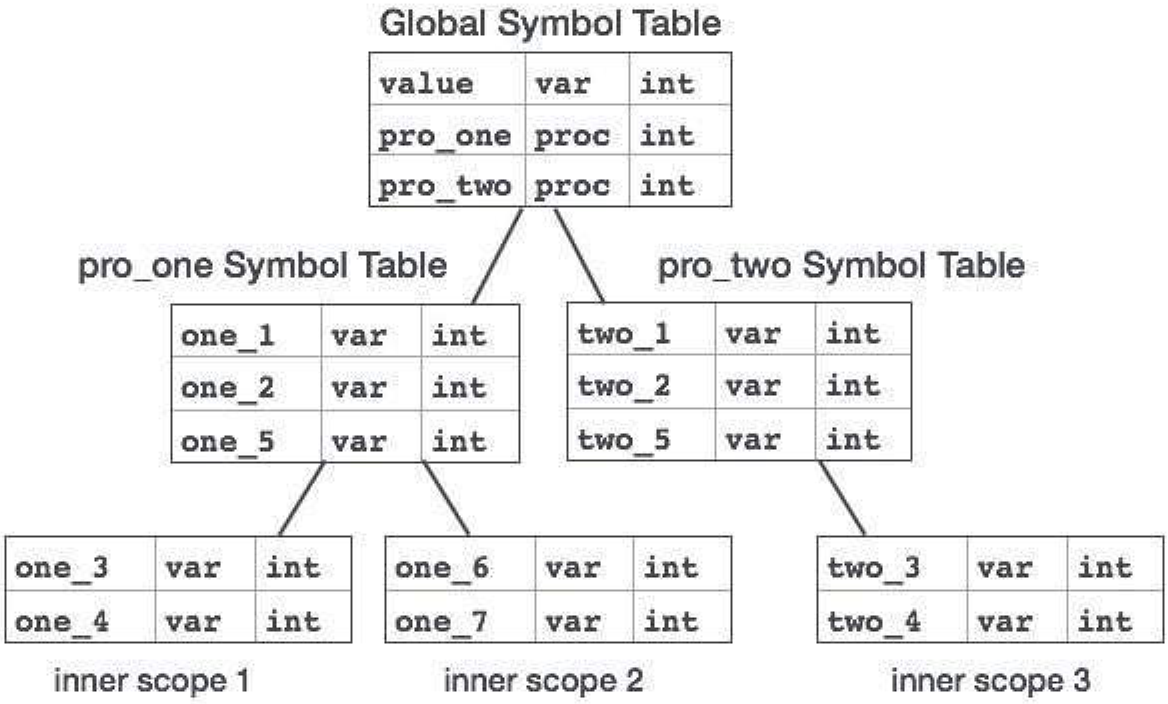
Scope Management

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
int value=10;
void pro_one()
  {
  int one_1;
  int one_2;
     {              \
     int one_3;      |_  inner scope 1
     int one_4;      |
     }              /
      int one_5;
      {              \
     int one_6;      |_  inner scope 2
     int one_7;      |
     }              /
  }
void pro_two()
  {
  int two_1;
  int two_2;
      {              \
     int two_3;      |_  inner scope 3
     int two_4;      |
     }              /
   int two_5;
  }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

Hash Table

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data value has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Hash table: *The memory area where the keys are stored is called hash table.*

Properties of Hash table:

- Hash table is partitioned into b buckets, HT(0),..HT(b-1).
- Each bucket is capable of holding ‘s’ records.
- A bucket is said to consist of s slots, each slot being large enough to hold 1 record.
- Usually s=1 and each bucket can hold exactly 1 record.

It becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

HASHING TECHNIQUES

Hashing Technique: *The hashing technique is the method in which the address or location of a key 'k' is obtained by computing some arithmetic function 'f' of k. f(k) gives the address of k in the table. The address will be referred to as hash address or home address.*

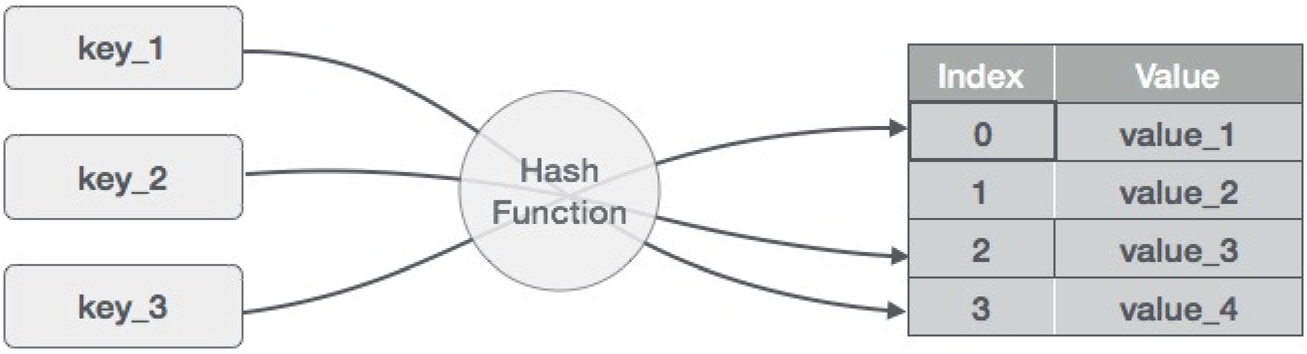
Hashing function f(k): *The hashing function used to perform an identifier transformation on k. f(k) maps the set of possible identifiers onto the integer 0 through b-1.*

Overflow: *An overflow is said to occur when a new identifier k is mapped or hashed by f(k) into a full bucket.*

Collision: *A collision occurs when two non-identical identifiers are hashed into the same bucket.*

When bucket size s=1, collisions and overflow occur simultaneously. Choose f which is easy to compute and results in very few collisions.

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in (key, value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)

- (37,98)

S.n.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Factors affecting Hash Table Design:

- Hash function
- Table size-usually fixed at the start
- Collision handling scheme

Hashing functions

A hash function is one which maps an element's key into a valid hash table index h(key) => hash table index.

Hash Function Properties:

- A hash function maps key to integer
  Constraint: Integer should be between [0, TableSize-1]
- A hash function can result in a many-to-one mappingν (causing collision)
  Collision occurs when hash function maps two or more keys to same array index.

- A “good” hash function should have the properties:
  1. Reduced chance of collision

     Different keys should ideally map to different indices

     Distribute keys uniformly over table

  2. Should be fast to compute

Different types:

(a) Mid square method.

The key k is squared then the required hash value is obtained by deleting some digits from both ends of k^2.

f (k) = specific digits(k^2)

For example, let us the compute the hash address for the following key value: 334, 567, 239.

(i) f(334) = 3rd , 4th digits (111556) = 11.
(ii) f(567) = 3rd , 4th digits (321489) = 21.
(iii) f(239) = 3rd , 4th digits (57121) = 57

(b) Division method

In the division method the hash address is the remainder after key k is divided by m, where m is the number of buckets.

f(k) = k mod m.

Let m = 10.

f(334) = 334 mod 10 = 4

f(567) = 567 mod 10 = 7

(c) Folding method

This method involves chopping the key k into two parts and adding them to get the hash address.

f(334) = 03 + 34 =37

f(567) = 05 + 67 = 72

Collision resolution

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, 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.

Techniques to Deal with Collisions:

Chaining
Open addressing
Double hashing, etc.

Chaining

In chaining, the buckets are implemented using linked lists. If there is a collision, then a new node is created and added at the end of the bucket. Hence all records in T with the same hash address h may be linked together to form a linked list. It is also known as open hashing.

In chaining, the entries are inserted as nodes in a linked list. The hash table itself is an array of head pointers.

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.

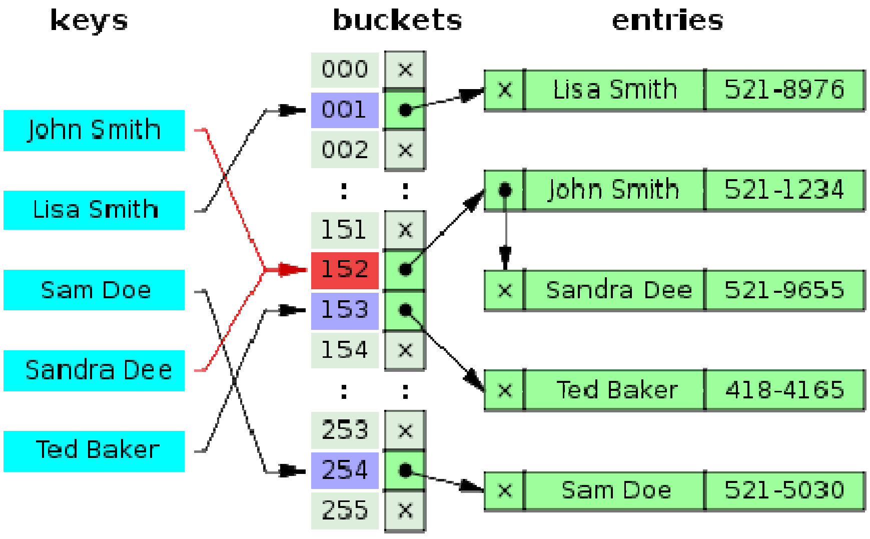


Fig. Hash collision resolved by separate chaining

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.

The advantages of using chaining are

- Insertion can be carried out at the head of the list at the index.
- The array size is not a limiting factor on the size of the table.

The prime disadvantage is the memory overhead incurred if the table size is small.

Potential disadvantages of Chaining

Linked lists could get long

□ Especially when N approaches M

□ Longer linked lists could negatively impact performance

More memory because of pointers

Absolute worst-case (even if N << M):

□ All N elements in one linked list!

□ Typically the result of a bad hash function

Open addressing

In open hashing, 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. 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.

Wellknown 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

Collision Resolution by Open Addressing

When a collision occurs, look elsewhere in the table for an empty slot

□ Advantages over chaining

□ No need for list structures

□ No need to allocate/deallocate memory during insertion/deletion (slow)

□ Disadvantages

□ Slower insertion – May need several attempts to find an empty slot

□ Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance, Load factor λ ≈ 0.5

Linear Probing

In linear probing, if there is a collision then we try to insert the new key value in the next available free space.

As we can see, it may happen that the hashing technique used to create already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

S.n.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13

9	37	37 % 20 = 17	17	18

Quadratic Probing:

Quadratic Probing is similar to Linear probing. The difference is that if you were to try to insert into a space that is filled you would first check 1^2 = 1 element away then 2^2 = 4 elements away, then 3^2 =9 elements away then 4^2=16 elements away and so on.

With linear probing we know that we will always find an open spot if one exists (It might be a long search but we will find it). However, this is not the case with quadratic probing unless you take care in the choosing of the table size. For example consider what would happen in the following situation:

Table size is 16. First 5 pieces of data that all hash to index 2

- First piece goes to index 2.
- Second piece goes to 3 ((2 + 1)%16
- Third piece goes to 6 ((2+4)%16
- Fourth piece goes to 11((2+9)%16
- Fifth piece dosen't get inserted because (2+16)%16==2 which is full so we end up back where we started and we haven't searched all empty spots.

Double Hashing:

Double Hashing is works on a similar idea to linear and quadratic probing. Use a big table and hash into it. Whenever a collision occurs, choose another spot in table to put the value. The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot. For example, given hash function H1 and H2 and key.

Steps:

- Check location hash1(key). If it is empty, put record in it.
- If it is not empty calculate hash2(key).
- check if hash1(key)+hash2(key) is open, if it is, put it in
- repeat with hash1(key)+2_hash2(key), hash1(key)+3_hash2(key) and so on, until an opening is found.

Basic Operations

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

- **Search –** search an element in a hashtable.
- **Insert –** insert an element in a hashtable.
- **Delete –** delete an element from a hashtable.

DataItem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem {
   int data;
   int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hashcode as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem *search(int key){
   //get the hash
   int hashIndex = hashCode(key);
   //move in array until an empty
   while(hashArray[hashIndex] != NULL){
      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }
   return NULL;
}
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the key passed and locate the index using that hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

```
void insert(int key,int data){
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;
   //get the hash
   int hashIndex = hashCode(key);
   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1){
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }
   hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, Compute the hash code of the key passed and locate the index using that hash code as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hash table intact.

```
struct DataItem* delete(struct DataItem* item){
   int key = item->key;
   //get the hash
   int hashIndex = hashCode(key);
   //move in array until an empty
   while(hashArray[hashIndex] !=NULL){
      if(hashArray[hashIndex]->key == key){
         struct DataItem* temp = hashArray[hashIndex];
         //assign a dummy item at deleted position
      hashArray[hashIndex] = dummyItem;
      return temp;
    }
    //go to next cell
    ++hashIndex;
   //wrap around the table
    hashIndex %= SIZE;
  }
  return NULL;
}
```

Sets

A Set is a collection of objects need not to be in any particular order. It is just applying the mathematical concept in concept. The set with no element is called null set or empty set.

Some set data structures are designed for **static** or **frozen sets** that do not change after they are constructed. Static sets allow only query operations on their elements — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and deletion of elements from the set.

Rules:

Elements should not be repeated.

Each element should have a reason to be in the set.

Example:

Assume we are going to store indian cricketers in a set. The names should not be repeated, as well the name who is not in the team can’t be inserted. This is the restriction which need to be followed in the set.

Representation of Sets:

List

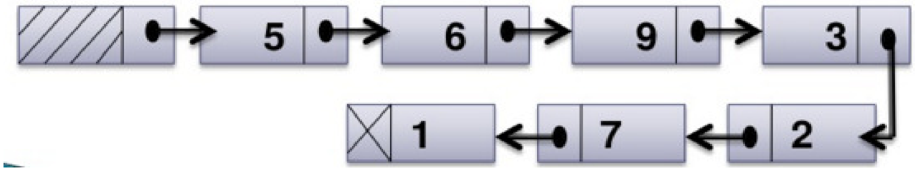
Tree

Hash Table

Bit Vectors

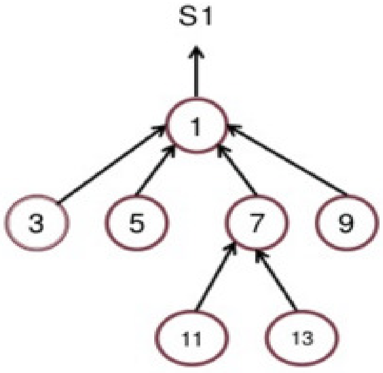
List Representation:

Its a simple and straight forward representation, which is best suited for dynamic storage facility. This representation allows multiplicity of elements i.e., Bag strutures. All the operations can be easily implemented and performance of these operations are as good as compared to other representatations. A set S={5,6,9,3,2,7,1}



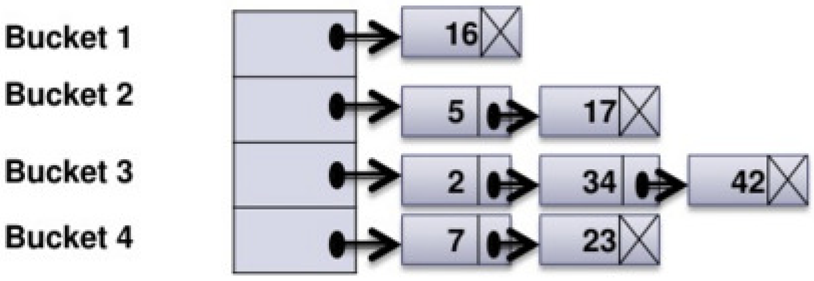
Tree Representation:

Tree is used to represent a set, and each element in the set has the same root. Each element in the set has a pointer to its parent. For example a set S1={1,3,5,7,9,11,13} can be represented as



Hash Table Representation:

In this representation the elements in collection are seperated into number of buckets. Each bucket can hold arbitrary number of elements. Consider the set S={2,5,7,16,17,23,34,42}, the hash table with 4 buckets and H(x) hash function can store wchi can place element from S to any of the four buckets.



Bit Vectors Representation:

Two variations are proposed which are maintaining the actual data values, maintaining the indication of presence or absence of data. A set, giving the records about the age of cricketer less than or equal to 35 is as given {0,0,0,0,1,1,1,1,0,1,1}. Here 1 indicates the presence of records having age less than or equal to 35. 0 indicates the absence of records having the age less than or euqla to 35. As we have to indicate presence or absence of an element only, so 0 or 1 can be used for indication of saving storage space.

Operations on Sets:

Basic operations:

- union(S,T): returns the union of sets S and T.
- intersection(S,T): returns the intersection of sets S and T.
- difference(S,T): returns the difference of sets S and T.
- subset(S,T): a predicate that tests whether the set S is a subset of set T.

Operations on a static set structure S:

- is_element_of(*x*,*S*): checks whether the value *x* is in the set *S*.
- is_empty(*S*): checks whether the set *S* is empty.
- size(*S*) or cardinality(*S*): returns the number of elements in *S*.
- iterate(*S*): returns a function that returns one more value of *S* at each call, in some arbitrary order.
- enumerate(*S*): returns a list containing the elements of *S* in some arbitrary order.
- build(x_1,x_2,…,x_n,): creates a set structure with values x_1,x_2,…,x_n.
- create_from(*collection*): creates a new set structure containing all the elements of the given collection or all the elements returned by the given iterator.

Operations on a Dynamic set structure:

- create(): creates a new, initially empty set structure.
  - create_with_capacity(*n*): creates a new set structure, initially empty but capable of holding up to *n* elements.
- add(*S*,*x*): adds the element *x* to *S*, if it is not present already.
- remove(*S*, *x*): removes the element *x* from *S*, if it is present.
- capacity(*S*): returns the maximum number of values that *S* can hold.

Other operations:

- pop(*S*): returns an arbitrary element of *S*, deleting it from *S*.
- pick(*S*): returns an arbitrary element of *S*. Functionally, the mutator pop can be interpreted as the pair of selectors (pick, rest), where rest returns the set consisting of all elements except for the arbitrary element. Can be interpreted in terms of iterate.
- map(*F*,*S*): returns the set of distinct values resulting from applying function *F* to each element of *S*.
- filter(*P*,*S*): returns the subset containing all elements of *S* that satisfy a given predicate *P*.
- fold(A_0,*F*,*S*): returns the value $A_{|S|}$ after applying $A_{i+1} := F(A_i, e)$ for each element *e* of *S*, for some binary operation *F*. *F* must be associative and commutative for this to be well-defined.
- clear(*S*): delete all elements of *S*.

- equal(S_1, S_2): checks whether the two given sets are equal (i.e. contain all and only the same elements).
- hash(S): returns a hash value for the static set S such that if equal(S_1, S_2) then hash(S_1) = hash(S_2)

Applications:

1. Hash function.
2. Spelling Checker.
3. Information storage and retrieval.
4. Filtering the records from the huge database.

Hash function:

Hash function is a function that generate an integernumber with a particular logic. This function is like a one way entry. We can't reverse this function. But by doing the process with same number we may get an answer. Sometimes the same valyue may be generated for the diffrent values called collision. In that case, we need to check for the data in that location. In some cases, same values will generate the different outout. Both are possible.

Spell Checker:

This concept applied by using the hash table data structure. The two files are provided to us for process. One is 'dictionary file', which is the collection of meaningful words, and another one is 'input file', that contains the word to be validated for spell. Initially the word from the dictionary file and also from input file be inserted into the hash tables seperately. Intersection operation will be done between the two hash tables. If the set with one element is the result of the intersection, then the word is in dictionary. Spell is perfect. If, null set returned from the intersection, then the spell is wrong.

Static tree tables:

- When symbols are known in advance and no insertion and deletion is allowed, it is called a static tree table.
- An example of this type of table is a reserved word table in a compiler.

Dynamic tree tables:

- A dynamic tree tables are used when symbols are not known in advance but are inserted as they come and deleted if not required.
- Dynamic keyed tables are those that are built on-the-fly.
- The keys have no history associated with their use.

Reference

Debasis Samatha, “Classic Datastructures”, 2nd edition, PHI learning pvt. ltd., 2009 ISBN, 812033731X, 9788120337312.