Dynamic Hashing

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Introduction

- Traditional hashing schemes as described in the previous section are not ideal. This follows from the fact that one must statically allocate a portion of memory to hold the hash table.
- This hash table is used to point to the pages used to hold identifiers, or it may actually. hold the identifiers themselves. In either case, if the table is allocated to be as large as possible. then space can be wasted. If it is allocated to be too small, then when the data exceed the capacity of the hash table, the entire file must be restructured. a time-consuming process.
- The purpose of **dynamic hashing** (also referred to as **extendible hashing**) is to retain the fast retrieval time of conventional hashing while extending the technique so that it can accommodate dynamically increasing and decreasing tile size without penalty.

Introduction

- The purpose of dynamic hashing is to retain the fast retrieval time of conventional hashing while extending the technique so that it can accommodate dynamically increasing and decreasing file size without penalty.
- Assume a file F is a collection of records R. Each record has a key field K by which it is identified. Records are stored in pages or buckets whose capacity is p.
- The goal of dynamic hashing is to minimize access to pages.
- The measure of space utilization is the ratio of the number of records, n, divided by the total space, **mp**, where **m** is the number of pages.

Difference

Dynamic Hashing	Static Hashing
It allows the number of buckets to vary dynamically.	A fixed number of buckets is allocated to a file to store the records.
It uses a second stage of mapping to determine the bucket associated with some search-key value.	It uses a fixed hash function to partition the set of all possible search-key values into subsets, and then maps each subset to a bucket.
Performance does not degrade as the file grows.	Performance degrades due to the bucket overflow.
The index entries are randomized in a way that the number of index entries in each bucket is roughly the same.	It uses a dynamically changing function that allows the addressed space to grow and shrink with varying database files.

Dynamic Hashing

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

How to search a key

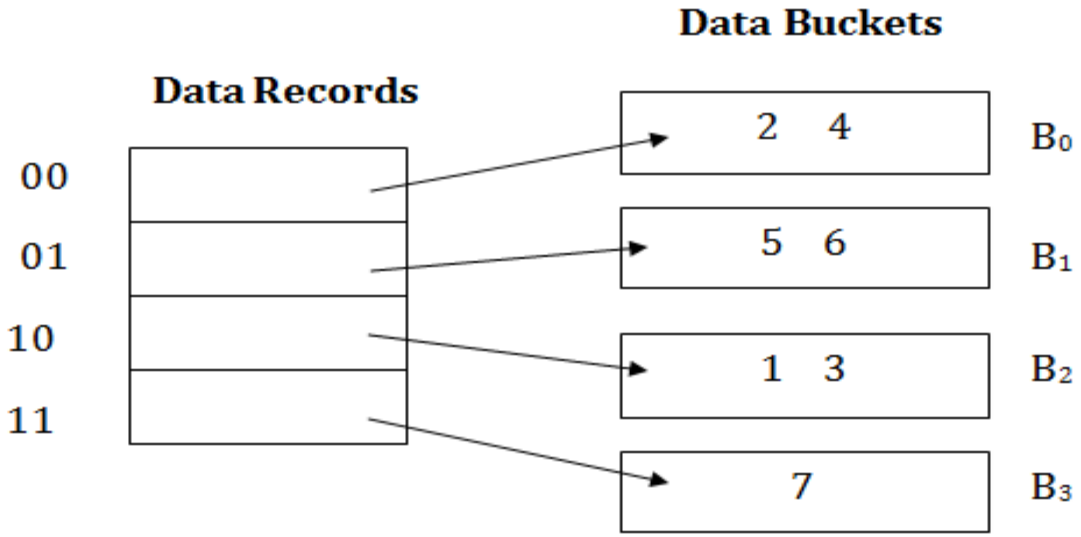
- First, calculate the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i.
- Take the least significant i bits of the hash address. This gives an index of the directory.
- Now using the index, go to the directory and find bucket address where the record might be.

How to insert a new record

- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.
- If the bucket is full, then we will split the bucket and redistribute the records.

Dynamic Hashing

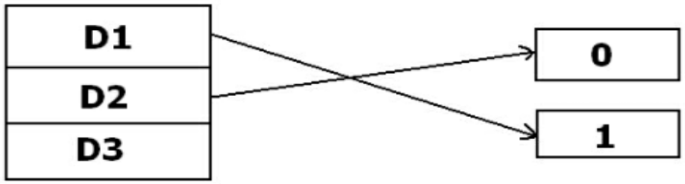
- The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. **Dynamic hashing** is also known as **extended hashing**.
- Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



Dynamic hashing

- For Example, there are three data records D1, D2 and D3 .
- The hash function generates three addresses 1001, 0101 and 1010 respectively.
- This method of storing considers only part of this address – especially only first one bit to store the data. So it tries to load three of them at address 0 and 1.

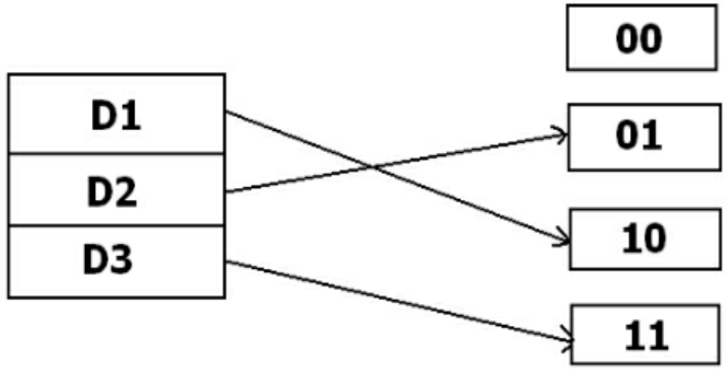
h(D1) -> 1001
h(D2) -> 0101
h(D3) -> 1010



Dynamic hashing

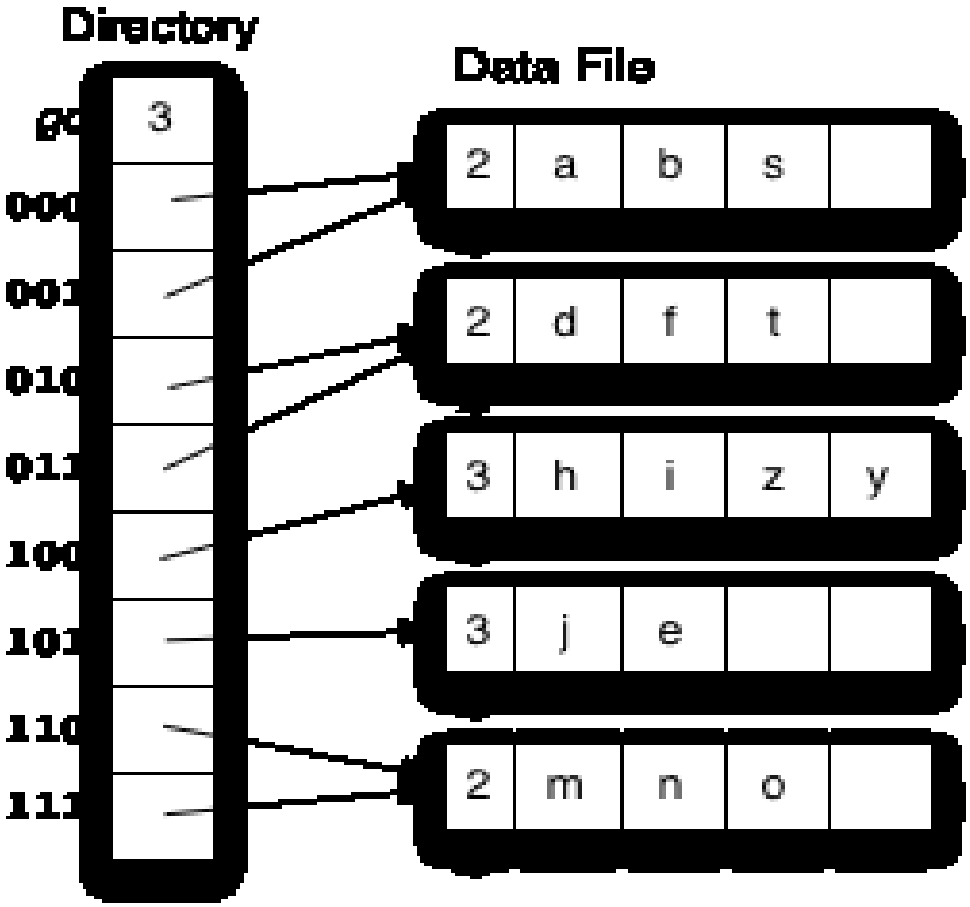
- But the problem is that No bucket address is remaining for D3.
- The bucket has to grow dynamically to accommodate D3. So it changes the address have 2 bits rather than 1 bit, and then it updates the existing data to have 2 bit address. Then it tries to accommodate D3.

h(D1) -> 1001
h(D2) -> 0101
h(D3) -> 1010



Dynamic Hashing

- In Dynamic hashing, data buckets grows or shrinks (added or removed dynamically) as the records increases or decreases



Dynamic Hashing

- We assume that a file **F** is a collection of records **R**. Each record has a key field **K** by which it is identified.
- Records are stored in pages or buckets whose capacity is **p**.
- The goal of dynamic hashing is to minimize access to pages. since they are typically stored on disk and their retrieval into memory dominates any operation.
- The measure of space utilization is the ratio of the number of records. **n**, divided by the total space **mp**, where **m** is the number of pages.

Dynamic Hashing: Schemes

(1) Expandable Hashing

- Knott, G. D. Expandable Open Addressing Hash Table Storage and Retrieval. Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, 186-206, 1971.

(2) Dynamic Hashing

- Larson, P. A. Dynamic Hashing. BIT 18(1978) ,184-201.
- Scholl, M. New File Organization Based on Dynamic Hashing. ACM Trans. on Database Systems, 6, 1(March 1981), 194-211.

(3) Virtual Hashing

- Litwin, W. Virtual Hashing: A Dynamically Changing Hashing. Proc. 4th Conf. on Very Large Data Bases, West Berlin, Sept. 1978, 517-523.

Dynamic Hashing: Schemes (cont.)

(4) Linear Hashing

- Litwin, W. Linear Hashing: A New Tool for File and Table Addressing. Proc. 6th Conf. on Very Large Data Bases, 212-223, Montreal, Oct. 1980.
- Larson, P. A. Linear Hashing with Partial Expansions. Proc. 6th Conf. on Very Large Data Bases, Montreal, Oct. 1980, 224-232
- Larson, P. A. Performance Analysis of Linear Hashing with Partial Expansions. ACM Trans. on Database Systems, 7, 4(Dec. 1982), 566-587.

(5) Trie Hashing

- Litwin, W. Trie Hashing. Res. Rep. MAP-I-014, I.R.I.A. Le Chesnay, France, 1981. (also in Proc. 1981 ACM SIGMOD International Conference on Management of Data)

Dynamic Hashing: Schemes (cont.)

(6) Extendible Hashing

- Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible Hashing - A Fast Access Method for Dynamic Files. ACM Trans. Database System 4, 3(Sept. 1979), 315-344.
- Tamminen, M. Extendible Hashing with Overflow. Information Processing Lett. 15, 5(Dec. 1982), 227-232.
- Mendelson, H. Analysis of Extendible Hashing. IEEE Trans. on Software Engineering, SE-8, 6(Nov. 1982), 611-619.
- Yao, A. C. A Note on the Analysis of Extendible Hashing. Information Processing Letter 11, 2(1980), 84-86.

Dynamic Hashing: Schemes (cont.)

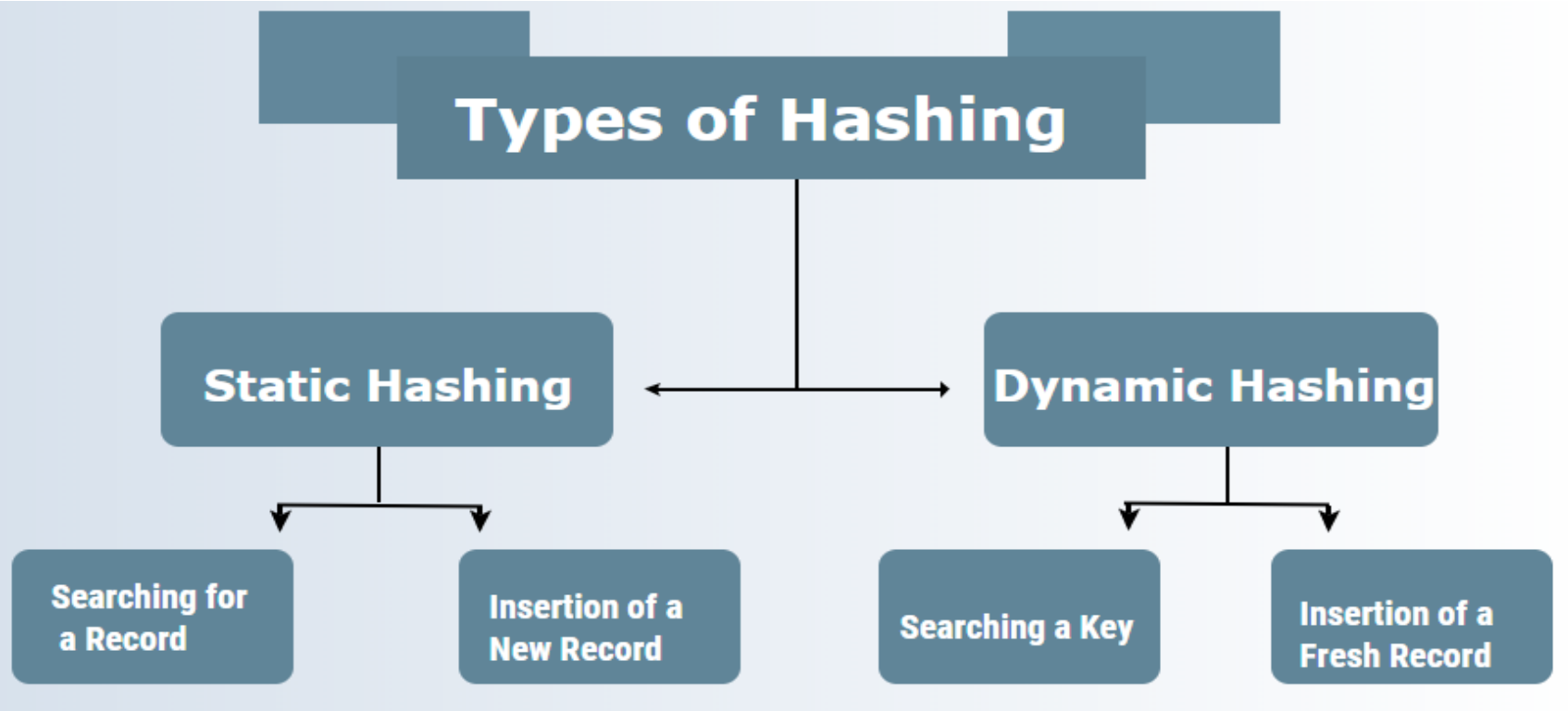
(7) HIT (Hash Indicator Table) Method

- Du, M. W., Hsieh, T. M., Jea, K. F., and Shieh, D. W. The Study of a New Perfect Hash Scheme. IEEE Trans. On Software Engineering, SE-9, 3(May 1983), 305-313.
- Yang, W. P., and Du, M. W. Expandable Single-Pass Perfect Hashing. Proc. of National Computer Symposium, Taiwan, Dec. 1983, 210-217.
- Yang, W. P., and Du, M. W. A Dynamic Perfect Hash Function Defined by an Extended Hash Indicator Table. Proc. 10th Conf. on Very Large Data Bases, Singapore, Aug. 1984.
- Yang, W. P. Methods for Constructing Perfect Hash Functions and its Application to the Design of Dynamic Hash Files. Doctor Thesis, National Chiao Tung University, Hsinchu, Taiwan, ROC, June 1984.

(8) …

Type of Dynamic Hashing

1. Directory-Based Dynamic Hashing
2. Directoryless Dynamic Hashing



Dynamic Hashing Using Directories

Directory-Based Dynamic Hashing

Dynamic Hashing Using Directories

Given a list of identifiers in the following

Identifiers	Binary representation
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C0	110 000
C1	110 001
C2	110 010
C3	110 011
C5	110 101

- Now put these identifiers into a table of four pages. Each page can hold at most two identifiers, and each page is indexed by two-bit sequence 00, 01, 10, 11.
- Now place **A0, B0, C2, A1, B1, and C3** in a binary tree, called **Trie**, which is branching based on the last significant bit at root. If the bit is 0, the **upper branch** is taken; otherwise, the **lower branch** is taken. Repeat this for next least significant bit for the next level.

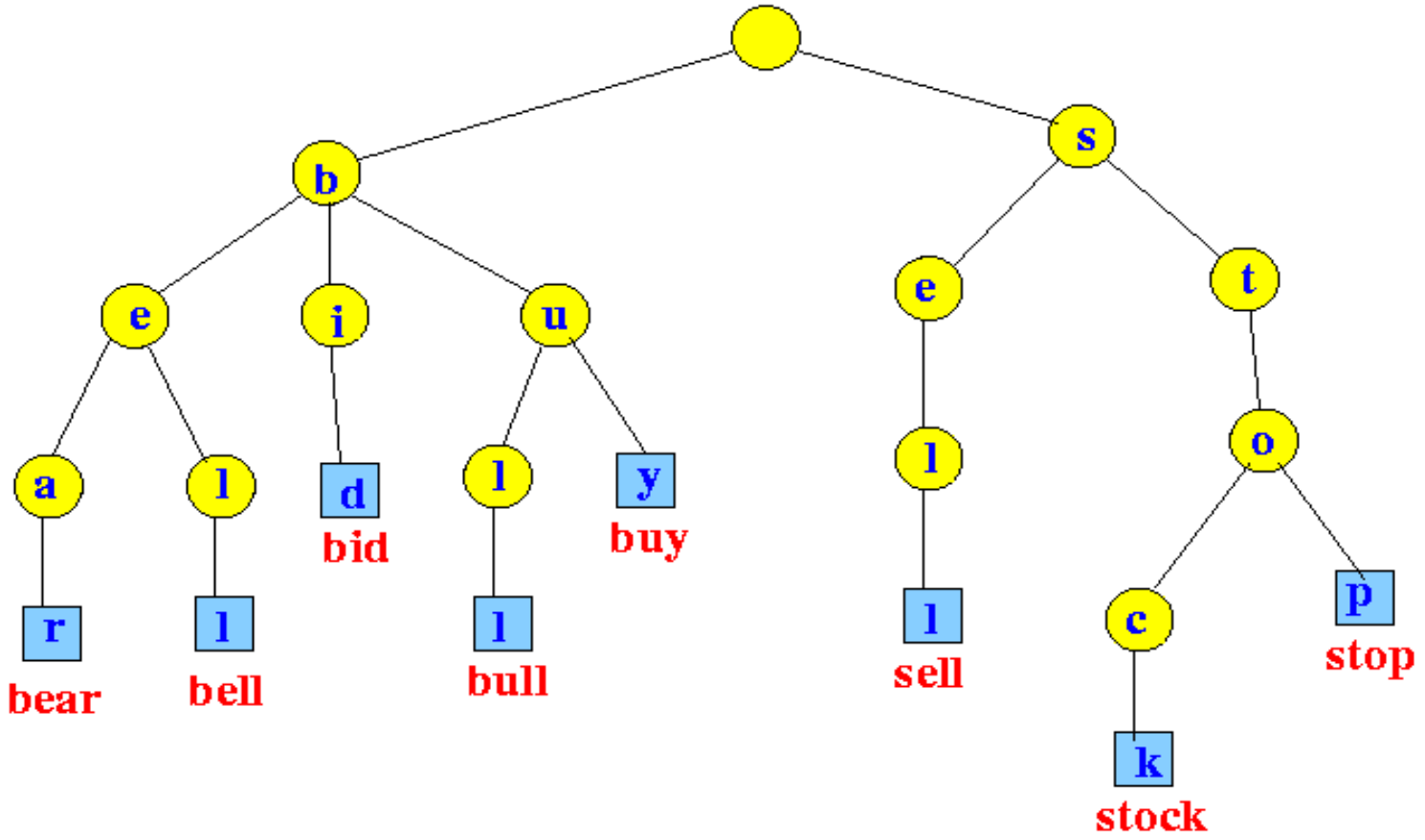
The Trie Data Structure

Trie: a binary tree in which an identifier is located by its bit sequence

Key lookup is faster. Looking up a key of length m takes worst case O(m) time.

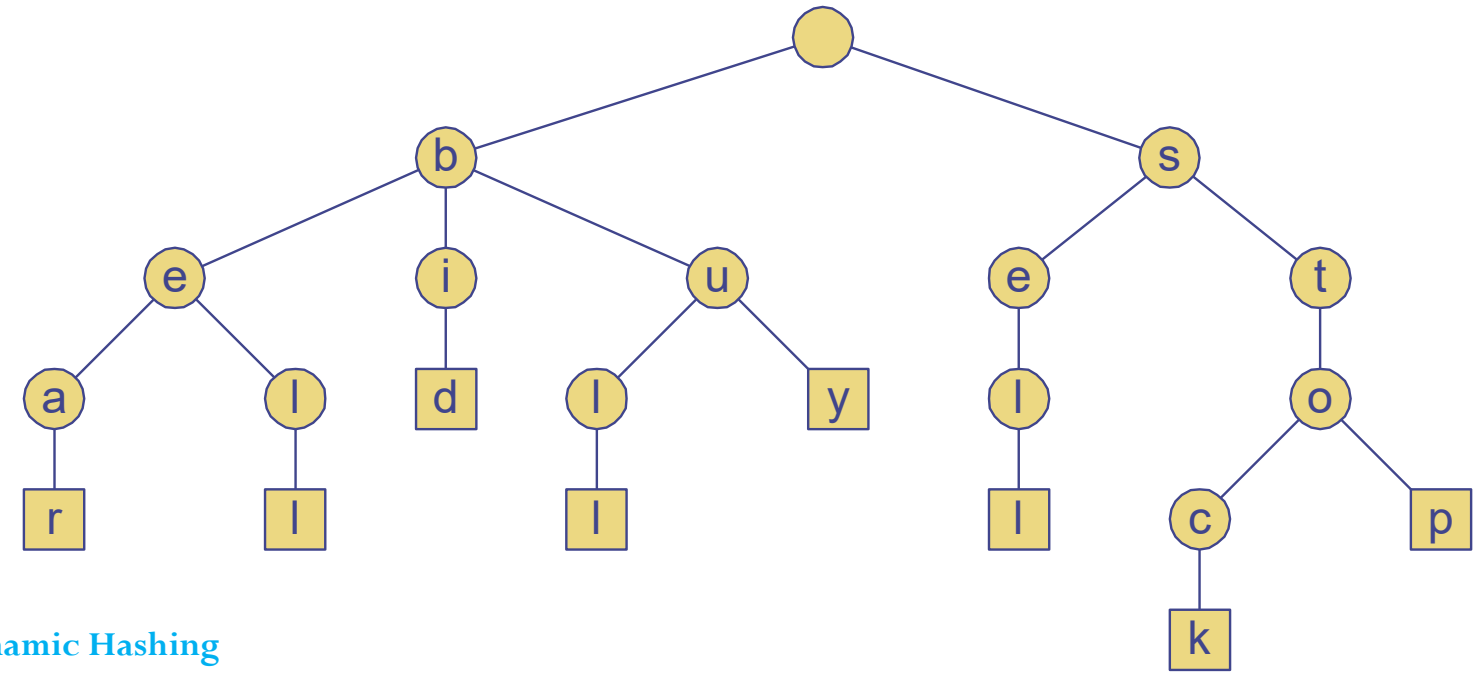
The Trie Data Structure

- Basic definition: a recursive tree structure that uses the digital decomposition of strings to represent a set of strings for searching



Standard Tries

- The standard **trie** for a set of strings S is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of S
- Example: standard **trie** for the set of strings
  S = { bear, bell, bid, bull, buy, sell, stock, stop }



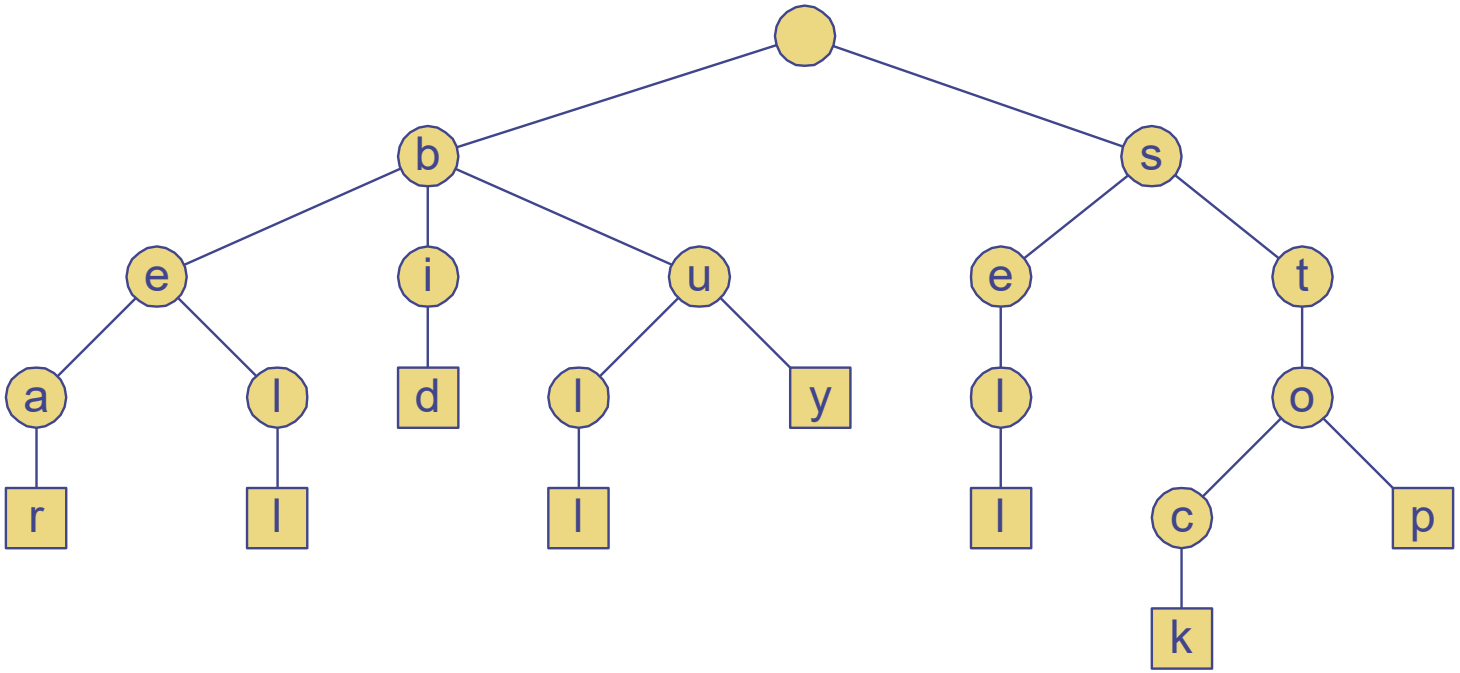
Analysis of Standard Tries

- A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:

❑ n total size of the strings in S

❑ m size of the string parameter of the operation

❑ d size of the alphabet



Dynamic Hashing Using Directories

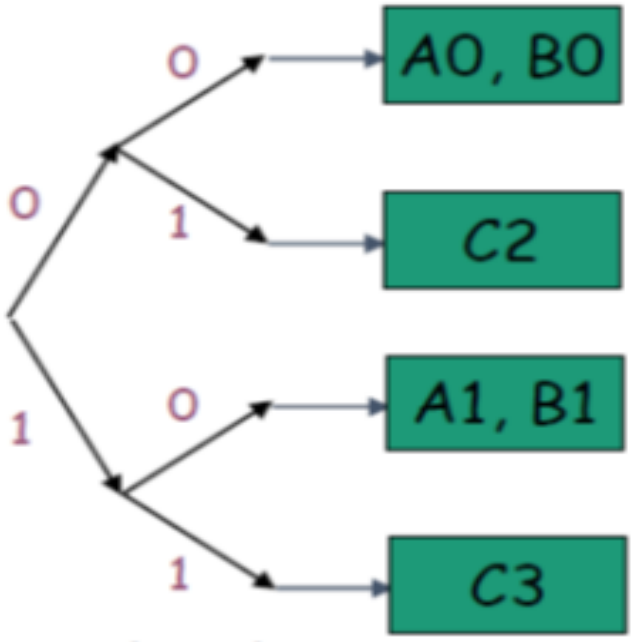
Given a list of identifiers in the following

Identifiers	Binary representation
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C0	110 000
C1	110 001
C2	110 010
C3	110 011
C5	110 101

- Now put these identifiers into a table of **four pages**. Each page can hold at most two identifiers, and each page is indexed by two-bit sequence 00, 01, 10, 11.
- Now place **A0, B0, C2, A1, B1, and C3** in a binary tree, called **Trie**, which is branching based on the last significant bit at root. If the bit is 0, the **upper branch** is taken; otherwise, the **lower branch** is taken. Repeat this for next least significant bit for the next level.

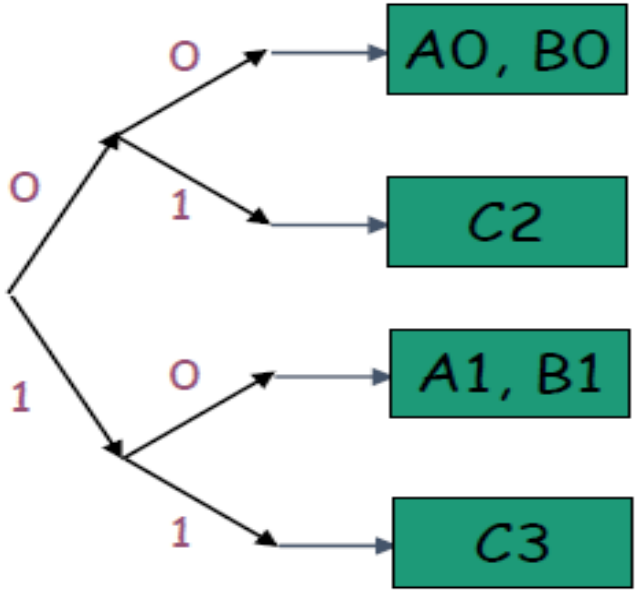
Dynamic Hashing Using Directories

- Now place **A0, B0, C2, A1, B1, and C3** in a binary tree, called **Trie**, which is branching based on the last significant bit at root.
- If the bit is 0, the **upper branch** is taken; otherwise, the **lower branch** is taken. Repeat this for next least significant bit for the next level.



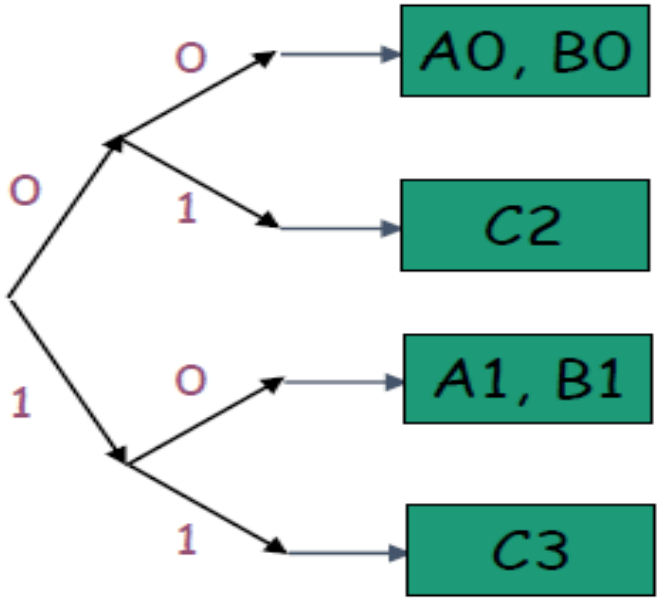
A Trie To Hold Identifiers

- Consider placing these identifiers into a table that has four pages.
- Each page can hold at most two identifiers, and each page is indexed by the two-bit sequence 00, 01, 10, and 11, respectively.
- We use the two low-order bits of each identifier to determine in which page of the table to place it. Note that we select the bits from least significant to most significant.



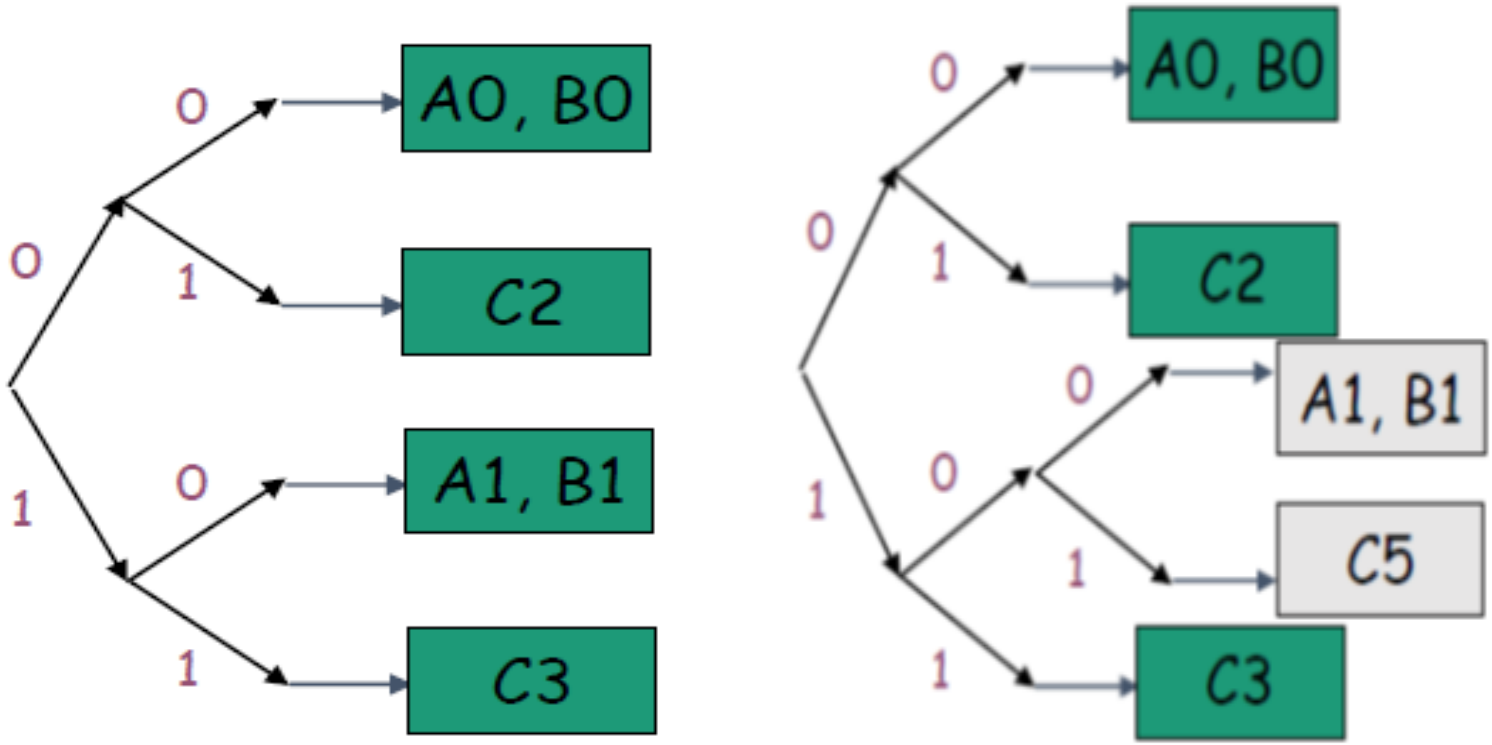
A Trie To Hold Identifiers

- Branching at the root is determined by the least significant bit. If this bit is zero. the upper branch is taken; otherwise. the lower branch is taken.
- Branching at the next level is determined by the second least significant bit. and so on. A0 and B0 are in the first page since their two low-order bits are 0 and 0.
- The second page contains only C2. To reach this page. we first branch on the least significant bit of C2 (i.e., 0) and then on the next bit (i.e .001). The third page contains A1 and B1.



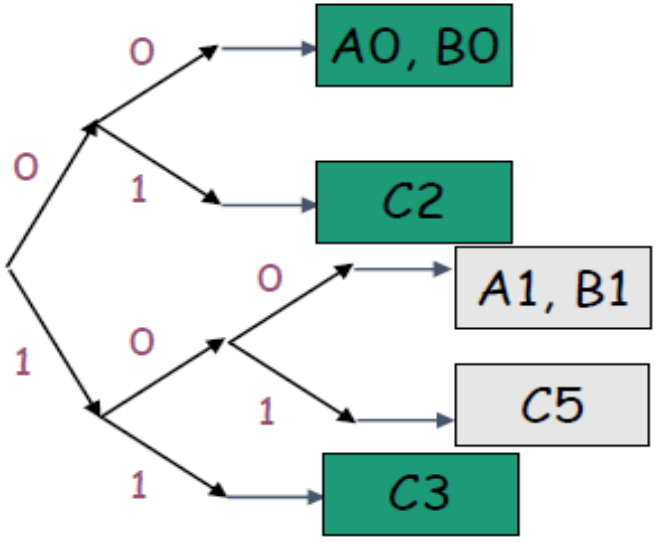
Dynamic Hashing Using Directories

- Inserting a new identifier, say C5, Since the two low-order bits of C5 are 1 and 0. we must place it in the third page. However, this page is full, and an overflow occurs.



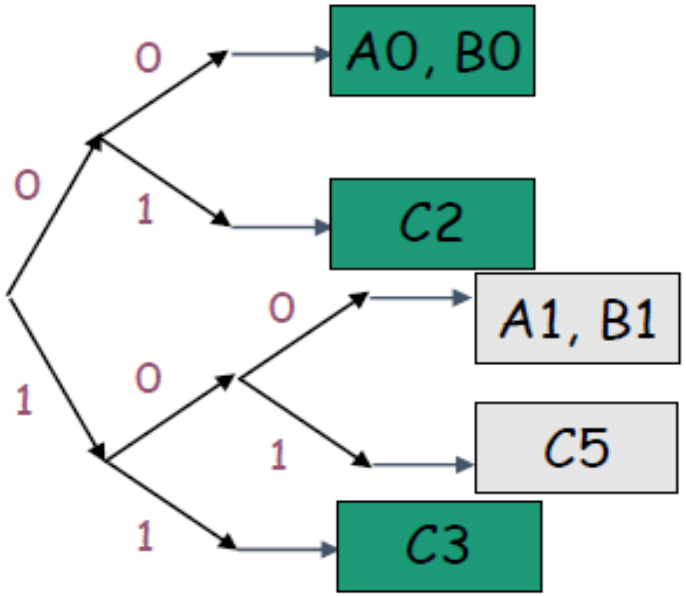
Dynamic Hashing Using Directories

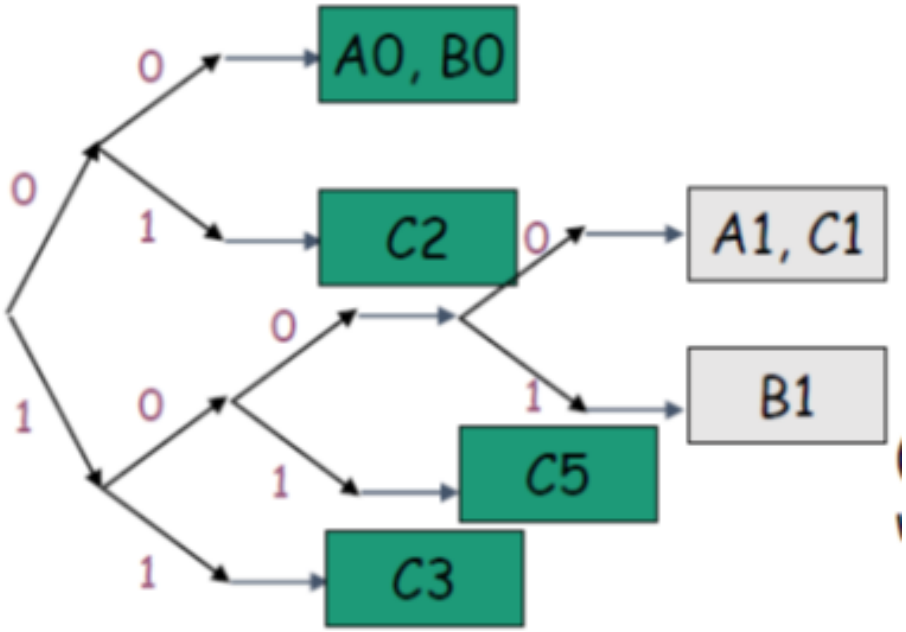
- To reach this page. we first branch on the least significant bit of Al or B1. This bit is one for both Al and B1. Next, we branch on the next bit. which is zero for both.
- The last page contains C3. with a bit pattern of 11.
- Notice that the nodes of this trie always branch in two directions corresponding to 0 or 1. Only the leaf nodes of the trie contain a pointer to a page.



Dynamic Hashing Using Directories

- If we now try to insert the new identifier C1. an overflow of the page containing A1 and B1 occurs.
- A new page is obtained, and the three identifiers are divided between the two pages according to their four low-order bits.
- A new page is added. and the depth of the trie increases by one level.





Analysis of Directory-Based Dynamic Hashing

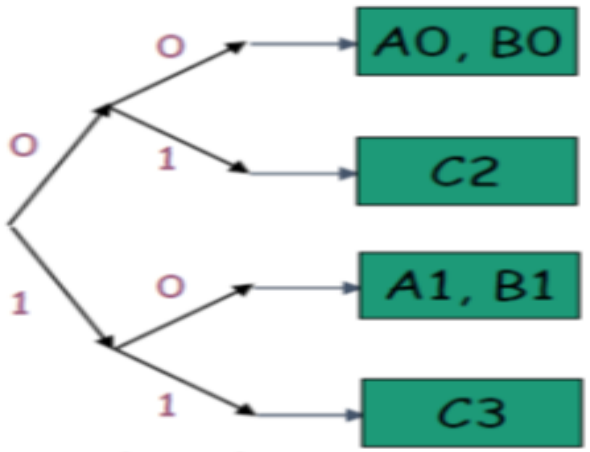
- The most important of the directory version of extensible hashing is that it guarantees only two disk accesses for any page retrieval.
- We get the performance at the expense of space. This is because a lot of pointers could point to the same page.
- One of the criteria to evaluate a hashing function is the space utilization.
- Space utilization is defined as the ratio of the number of records stored in the table divided by the total number of space allocated.
- Research has shown that dynamic hashing has 69% of space utilization if no special overflow mechanisms are used

Dynamic Hashing Using Directories

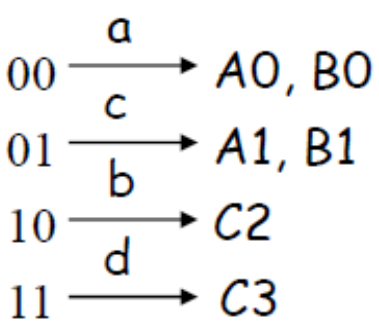
- A directory is a table of page pointers. If k bits are needed to distinguish the identifiers, the directory has 2^k entries indexed 0, … , 2^{k-1}.
- To find the page for an identifier, it uses the integer with binary representation equal to the last k bits of the identifier.
- The page pointed at by this directory entry is searched. Figure 8.10 shows the directories corresponding to the three tries in Figure 8.9.
- The first directory contains four entries indexed from 0 to 3 (the binary representation of each index is shown in Figure).

Dynamic Hashing Using Directories

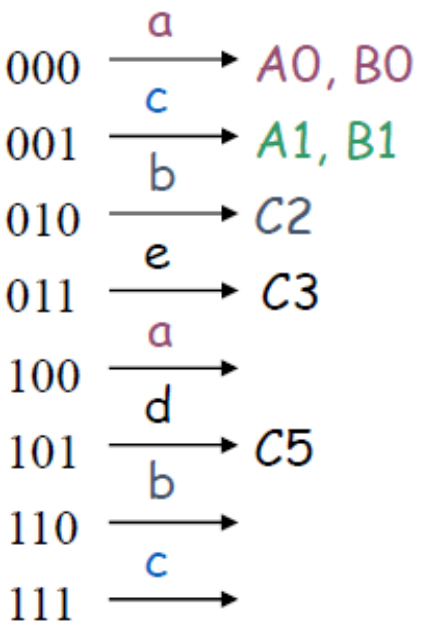
- Each entry contains a pointer to a page. This pointer is shown as arrow in the figure 8.10. The letter above each pointer is a page label.
- The page labels were obtained by labelling the pages of Figure (a) top to bottom, beginning with the label a.
- The page contents are shown immediately after the page pointer.
- To see the correspondence between the directory and the trie, notice that if the bits in the directory index are used to follow a path in the trie (beginning with the least significant bit). we will reach the page pointed at by the corresponding directory entry.



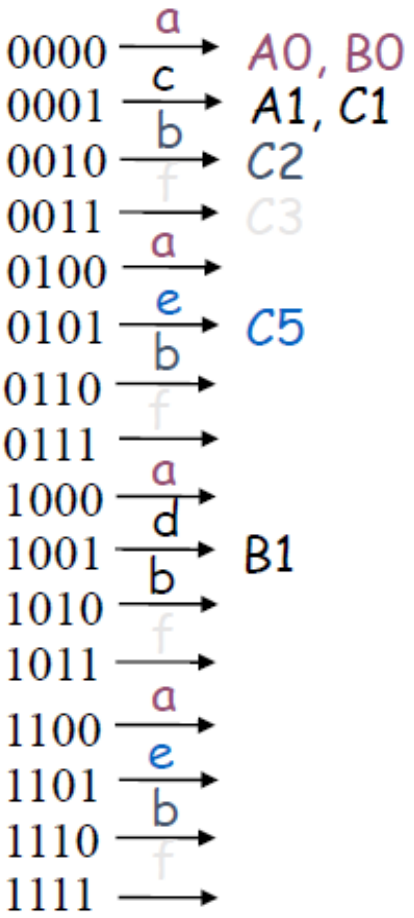
Trie Collapsed into Directories



(a) 2 bits

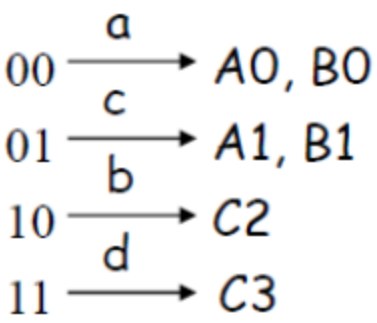


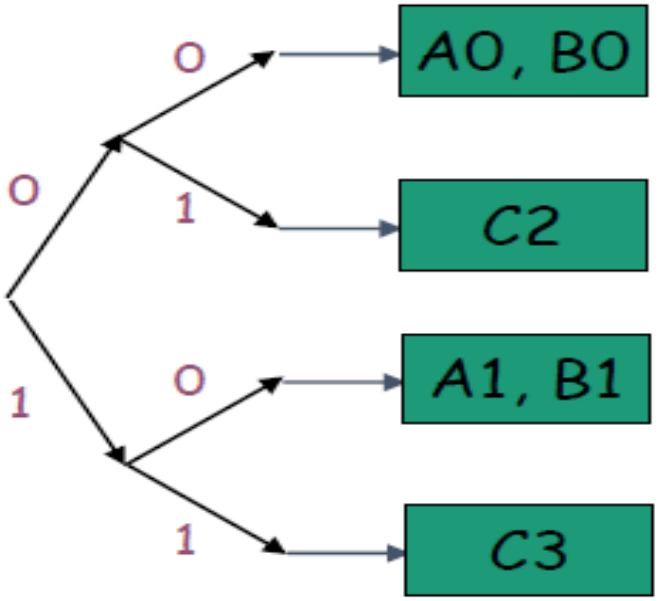
(b) 3 bits



© 4 bits

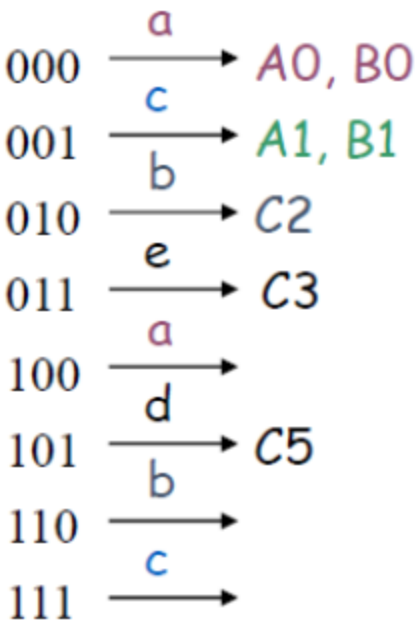
Trie Collapsed into Directory

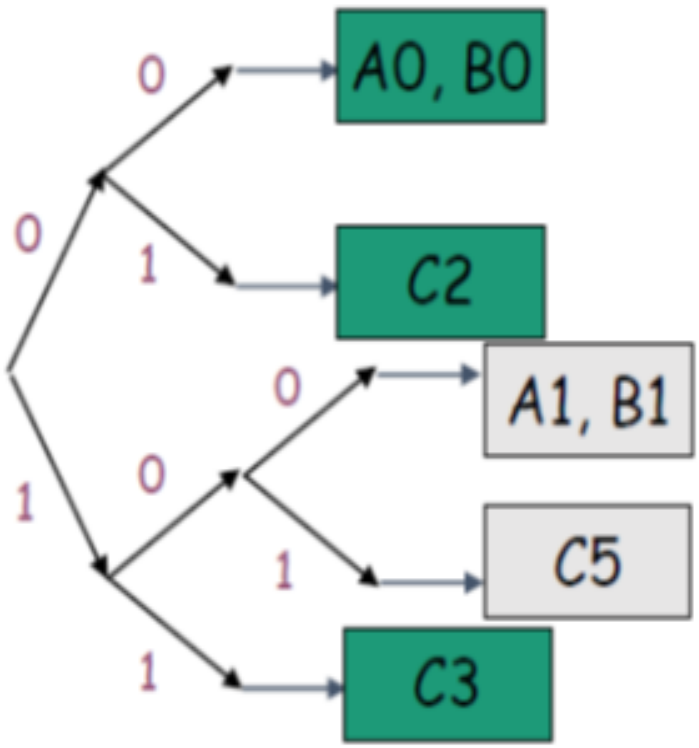




(a) 2 bits

Trie Collapsed into Directory

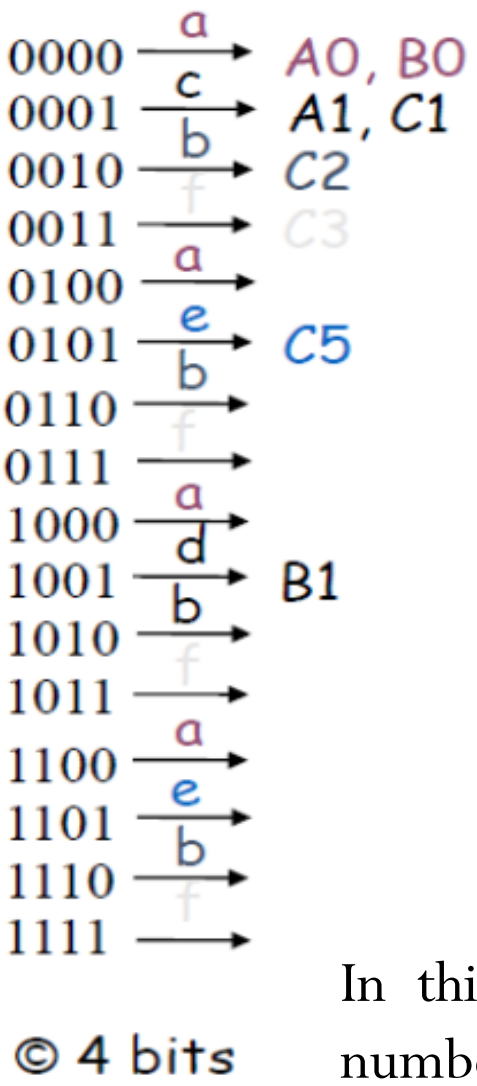


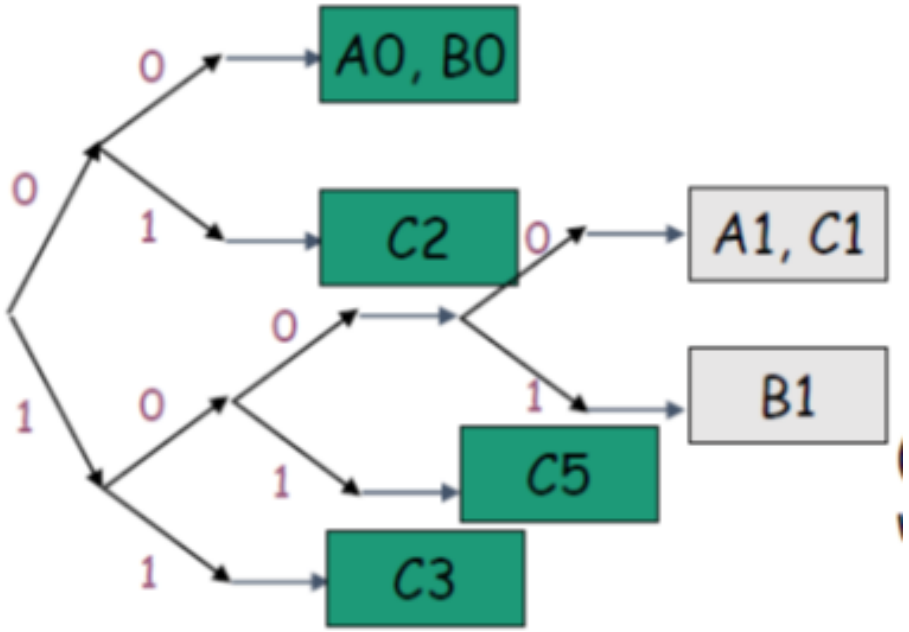


(b) 3 bits

Page **a** of the second directory has two directory entries (000 and 100) pointing to it.

Trie Collapsed into Directory





In this Figure there are six pages with the following number of pointers, respectively: 4, 4, 1, 1, 2, and 4.

Hashing With Directory

- Using a directory to represent a trie allows table of identifiers to grow and shrink dynamically.
- Accessing any page only requires two steps:

First step: use the hash function to find the address of the directory entry.

Second step: retrieve the page associated with the address

- Since the keys are not uniformly divided among the pages, the directory can grow quite large.
- To avoid the above issue, translate the bits into a random sequence using a uniform hash function. So identifiers can be distributed uniformly across the entries of directory. In this case, multiple hash functions are needed (a family of hash functions). Because, at any point, we may require a different number of bits to distinguish the new key

Family of hash functions

- a family of uniform hash functions:

$$hash_i: key \rightarrow \{0 \ldots 2^{i-1}\}, 1 \leq i \leq d$$

- If the page overflows, then we use $hash_i$ to rehash the original page into two pages, and we coalesce two pages into one in reverse case.
- Thus we hope the family holds some properties like hierarchy.

Overflow Handling

- A simple hash function family is simply adding a leading zero or one as the new leading bit of the result.
- When a page identified by i bits overflows, a new page needs to be allocated and the identifiers are rehashed into those two pages.
- Identifiers in both pages should have their low-order I bits in common. These are referred as ***buddies***.
- If the number of identifiers in buddy pages is no more than the capacity of one page, then they can be coalesce into one page.
- After adding a bit, if the number of bits used is greater than the depth of the directory, the whole directory doubles in size and its depth increases by 1.

Analysis of Directory-Based Dynamic Hashing

- The most important of the directory version of extensible hashing is that it guarantees only two disk accesses for any page retrieval.
- We get the performance at the expense of space. This is because a lot of pointers could point to the same page.
- One of the criteria to evaluate a hashing function is the space utilization.
- Space utilization is defined as the ratio of the number of records stored in the table divided by the total number of space allocated.
- Research has shown that dynamic hashing has 69% of space utilization if no special overflow mechanisms are used.

Advantages for Dynamic Hashing Using Directories

- Only double the directory rather than the whole hash table used in static hashing.
- Only rehash the entries in the buckets that overflows

Issues of Trie Representation

Two major factors that affects the retrieval time.

- Access time for a page depends on the number of bits needed to distinguish the identifiers.
- If identifiers have a skewed distribution, the tree is also skewed.

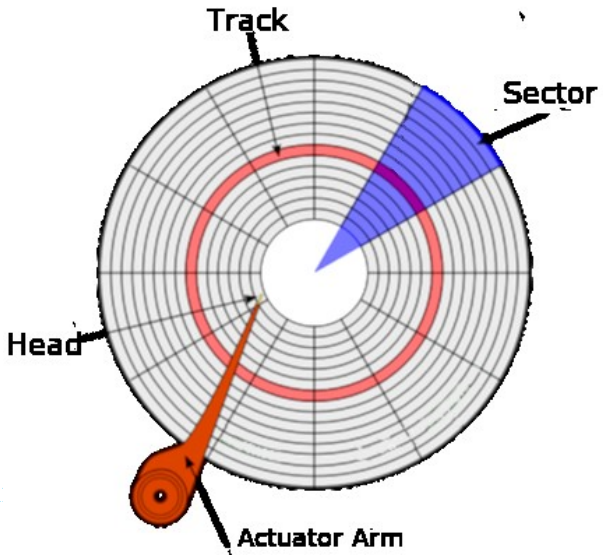
Directoryless Dynamic Hashing

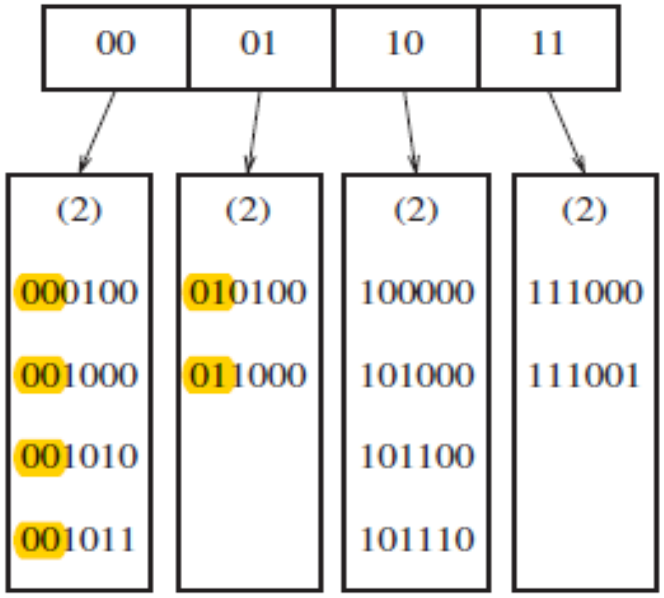
Example 1 : Extendible hashing scheme

- Let us assume that at any point we have N records to store; the value of N changes over time.
- Furthermore, at most ***M* records** fit in one disk block. (Let $M = 4$)
- If either probing hashing or separate chaining hashing is used, the major problem is that collisions could cause several blocks to be examined during a search, even for a well-distributed hash table.
- Furthermore, when the table gets too full, an extremely expensive rehashing step must be performed, which requires $O(N)$ disk accesses.

Example 1 : an extendible hashing scheme for these data.

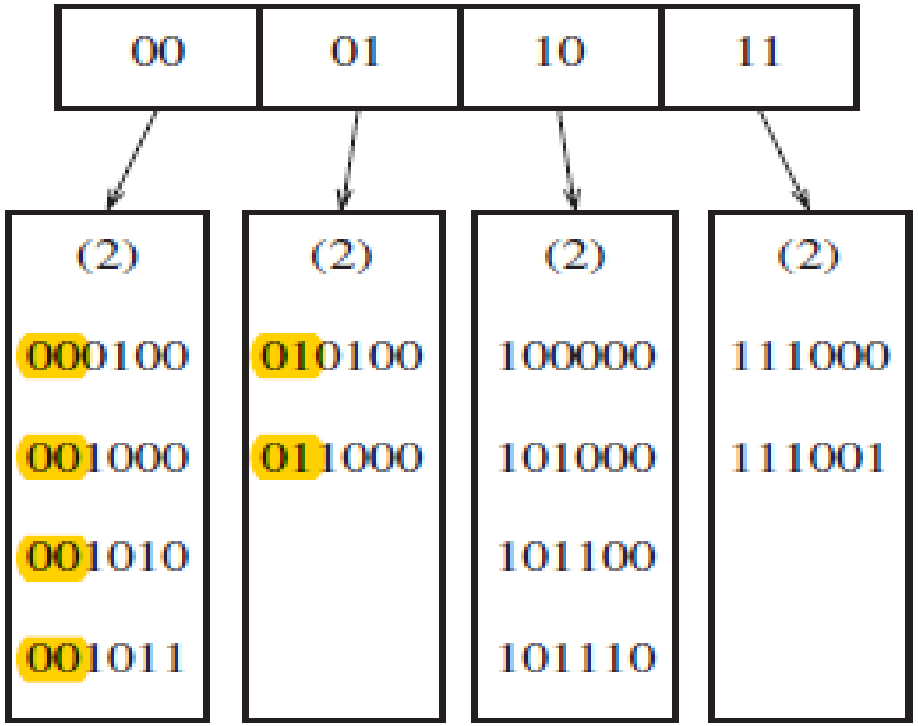
- Let the data consists of several 6-bit integers.
- The root of the "tree" contains four pointers determined by the leading two bits of the data.
- Each leaf has up to $M = 4$ elements. It happens that in each leaf the first two bits are identical; this is indicated by the number in parentheses.
- D will represent the number of bits used by the root, which is sometimes known as the **directory.**





Example 1 : an extendible hashing scheme for these data.

- The number of entries in the directory is thus $2D$. dL is the number of leading bits that all the elements of some leaf L have in common. dL will depend on the particular leaf, and $dL \leq D$.



Extensible Hashing

Fagin et al. present a method, called *extensible hashing*, for solving the above issues.

- A hash function is used to avoid skewed distribution. The function takes the key and produces a random set of binary digits.
- To avoid long search down the trie, the trie is mapped to a directory, where a directory is a table of pointers.
- If k bits are needed to distinguish the identifiers, the directory has 2^k entries indexed 0, 1, …, 2^{k-1}
- Each entry contains a pointer to a page.

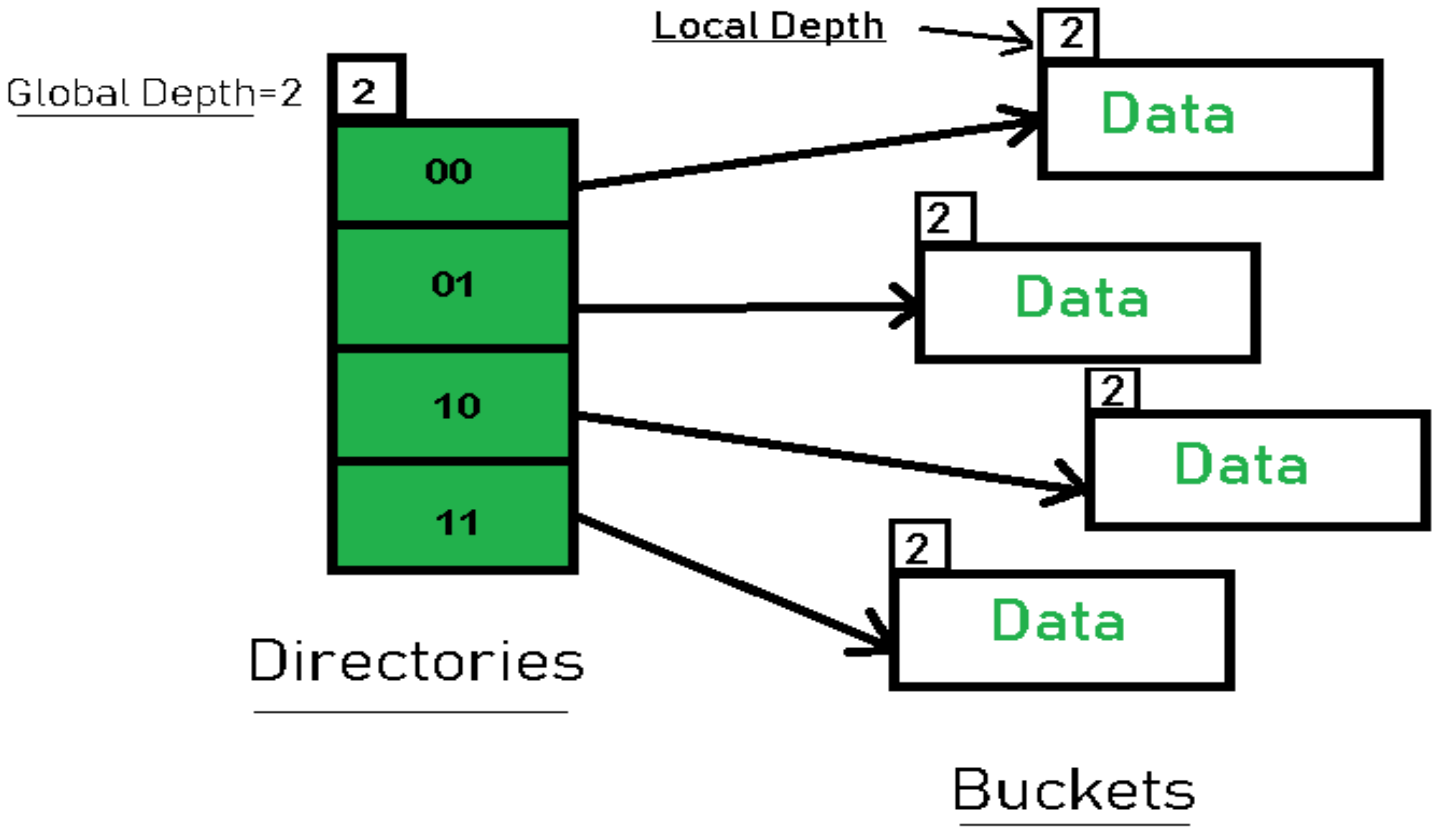
Dynamic Hashing Using Directories

- **Extendible Hashing** is a dynamic hashing method wherein directories, and buckets are used to hash data.
- It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Main features of Extendible Hashing:

- **Directories:** The directories store addresses of the buckets in pointers.
- An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.

Basic Structure of Extendible Hashing:

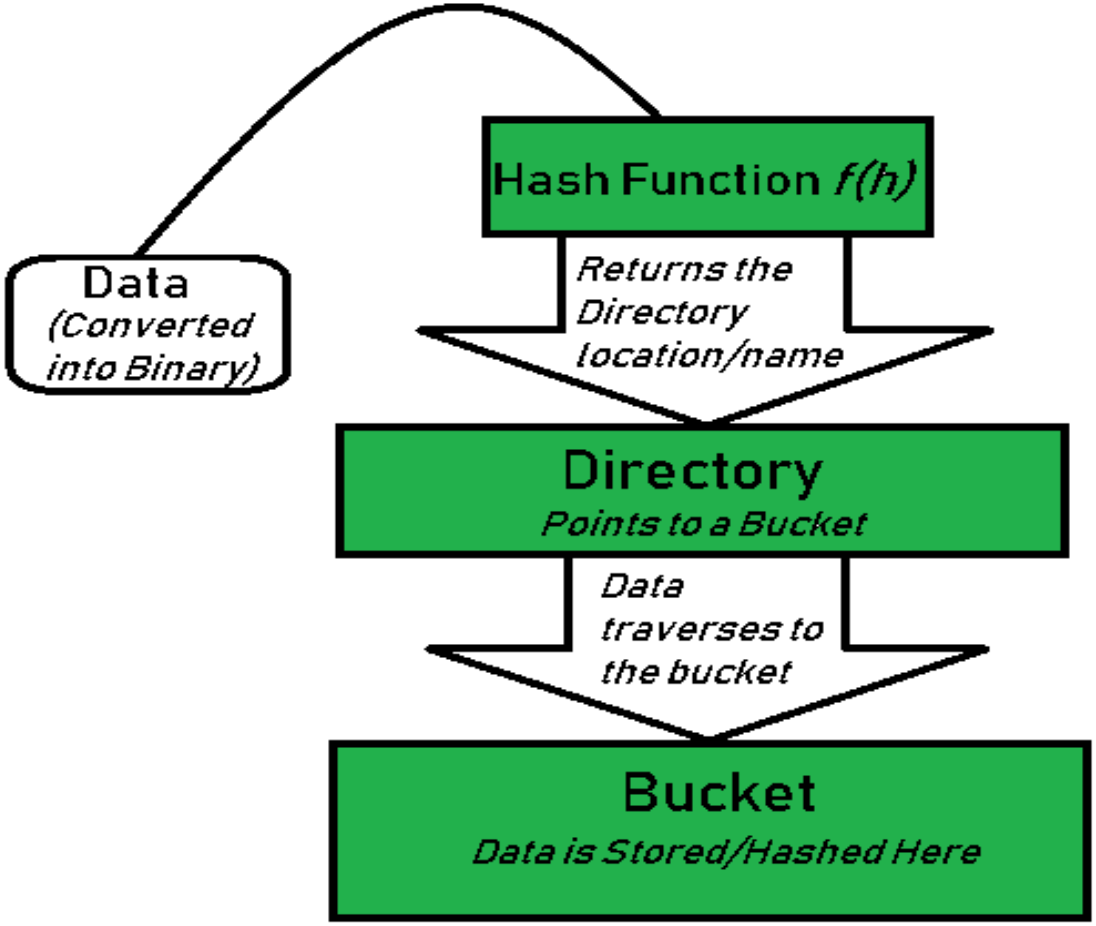


Extendible Hashing

Frequently used terms in Extendible Hashing:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories $= 2^{\text{Global Depth}}$.
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.
- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

Basic Working of Extendible Hashing:



Basic Working of Extendible Hashing: 1/3

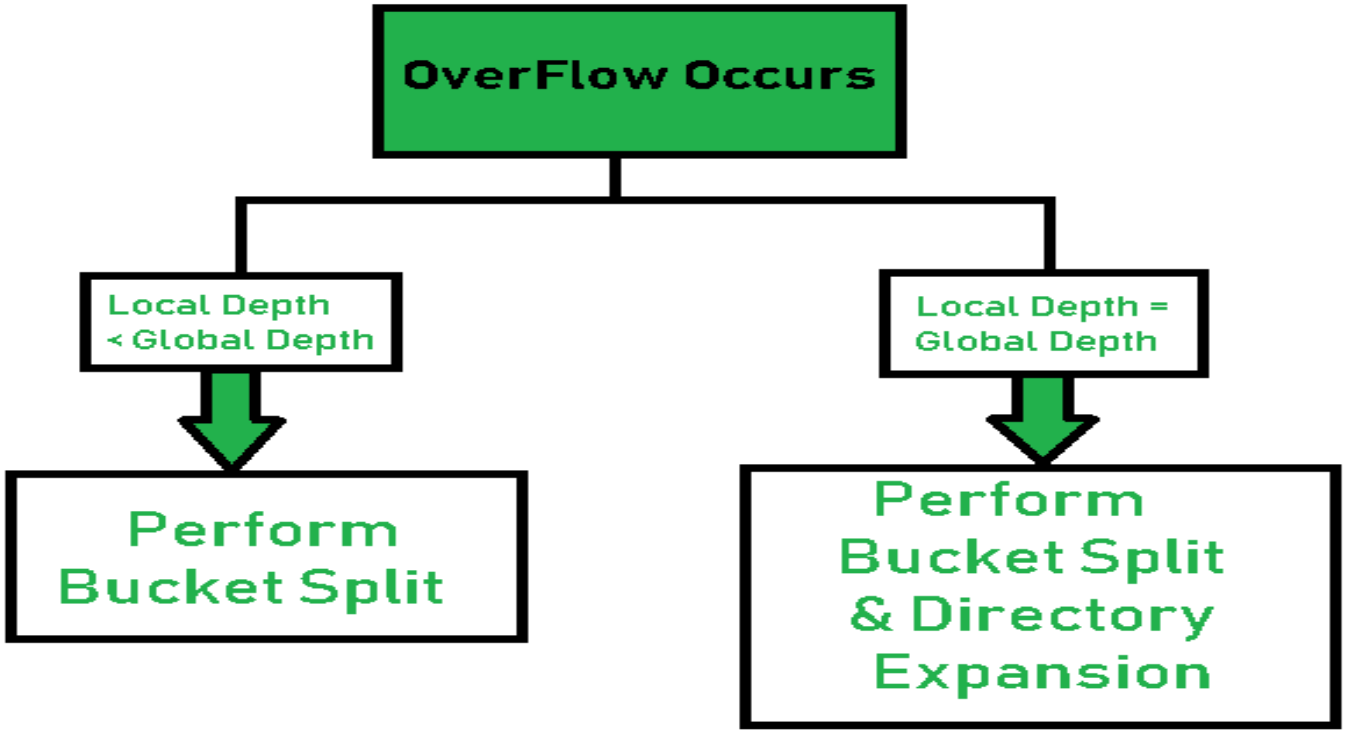
- **Step 1 – Analyze Data Elements:** Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
- **Step 2 – Convert into binary format:** Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.
- **Step 3 – Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.
- **Step 4 – Identify the Directory:** Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id. Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.

Basic Working of Extendible Hashing: 2/3

- **Step 5 – Navigation:** Now, navigate to the bucket pointed by the directory with directory-id 001.
- **Step 6 – Insertion and Overflow Check:** Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.
- **Step 7 – Tackling Over Flow Condition during Data Insertion:** Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.
- First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.
  - **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers. Directory expansion will double the number of directories present in the hash structure.
  - **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.

Basic Working of Extendible Hashing: 3/3

- **Step 8 – Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.
- **Step 9 –** The element is successfully hashed.



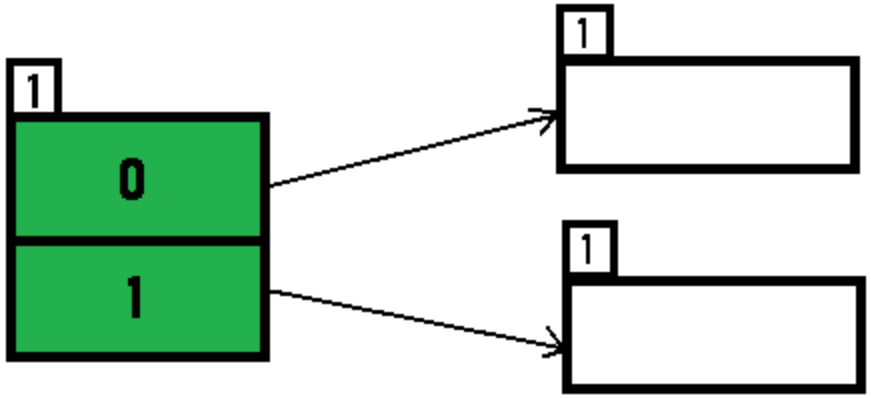
Example # Extendible Hashing : 1/

- Let us assume records with key **16, 4, 6, 22, 24, 10, 31, 7, 9, 20, & 26**
- **Assume the Bucket Size = 3**
- **Hash Function:** for the global depth X. Then the Hash Function returns LSBs of X.
- binary forms of each key to be inserted are as follows:

16- 10000
4- 00100
6- 00110
22- 10110
24- 11000
10- 01010
31- 11111
7- 00111
9- 01001
20- 10100
26- 11010

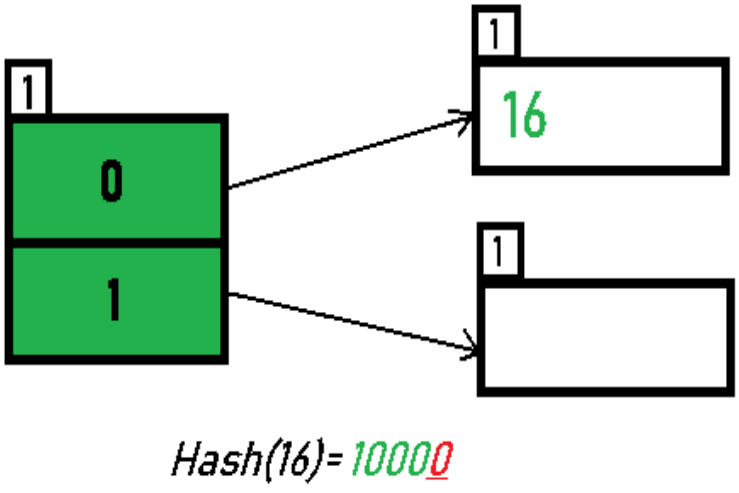
Example # Extendible Hashing : 1/

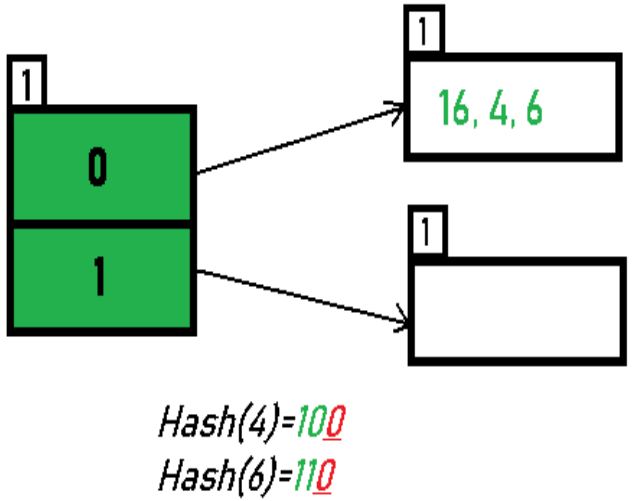
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



Example # Extendible Hashing : 1/

Inserting 16: The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.

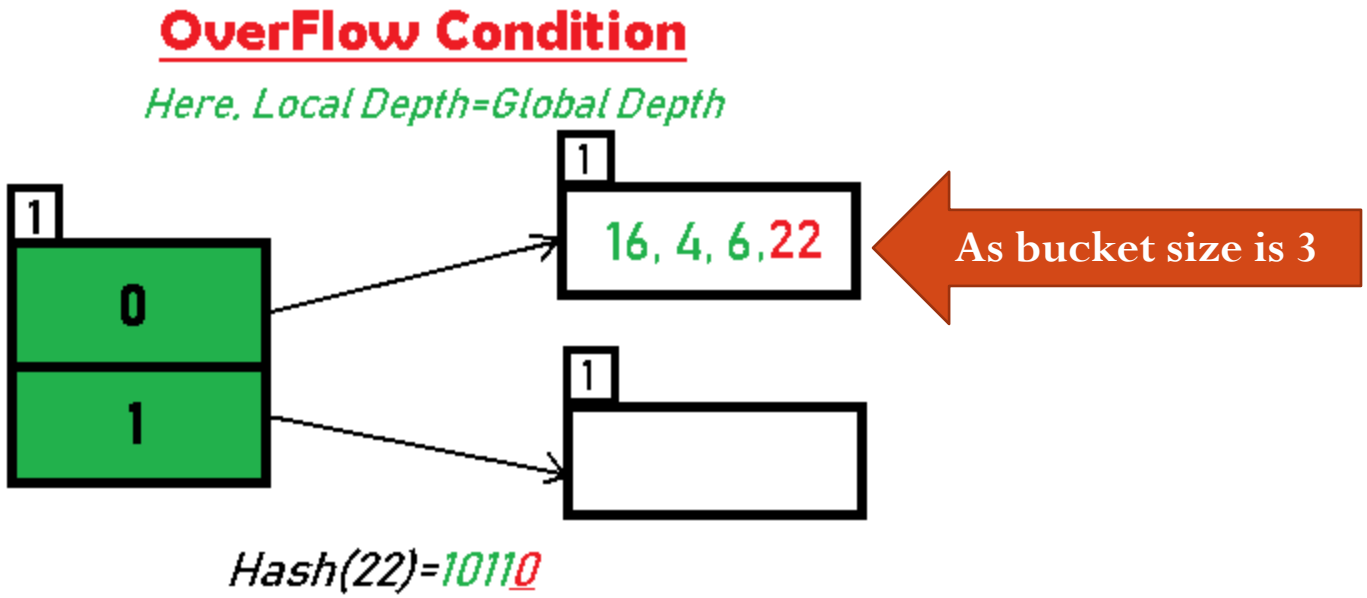




Inserting 4 and 6: Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as above:

Example # Extendible Hashing : 1/

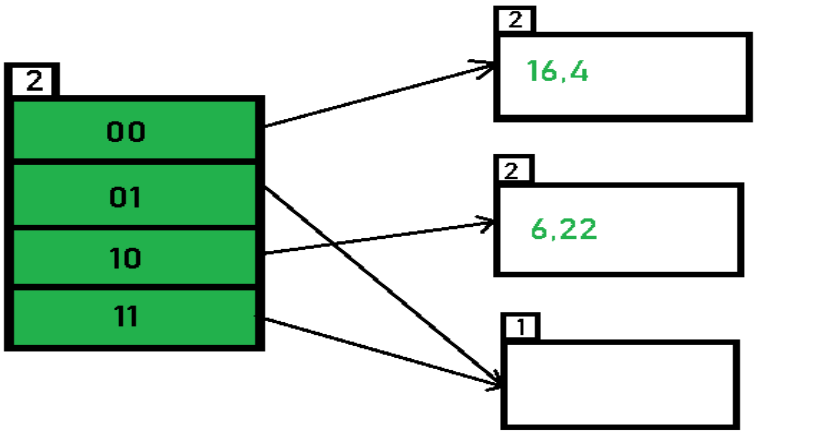
- **Inserting 22:** The binary form of 22 is 10110. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.



Example # Extendible Hashing : 1/

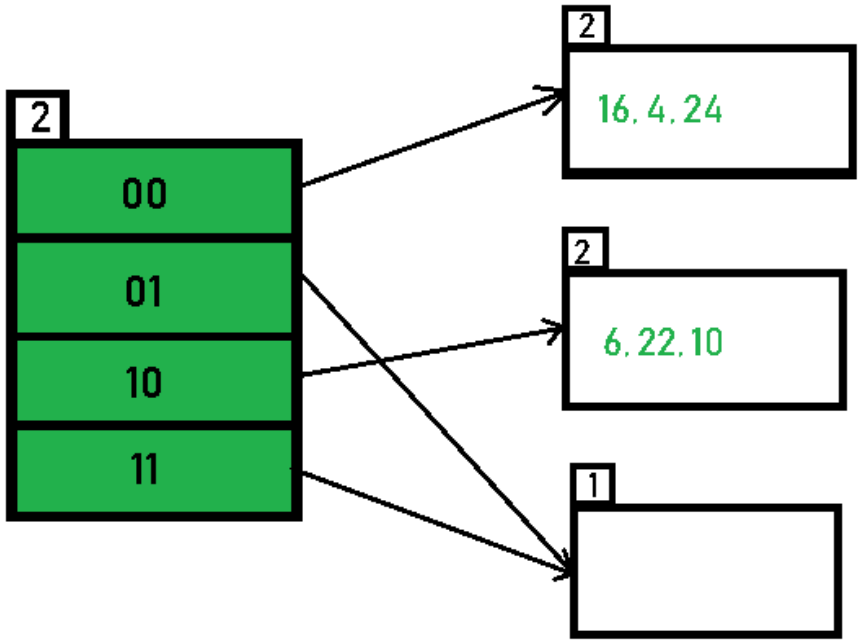
- As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the **bucket splits** and directory expansion takes place.
- Rehashing the numbers present in the overflowing bucket to take places after the split.
- As, the global depth is incremented by 1, the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t **2 LSBs**.[16(100**00**),4(001**00**),6(00110),22(10110)]

After Bucket Split and Directory Expansion



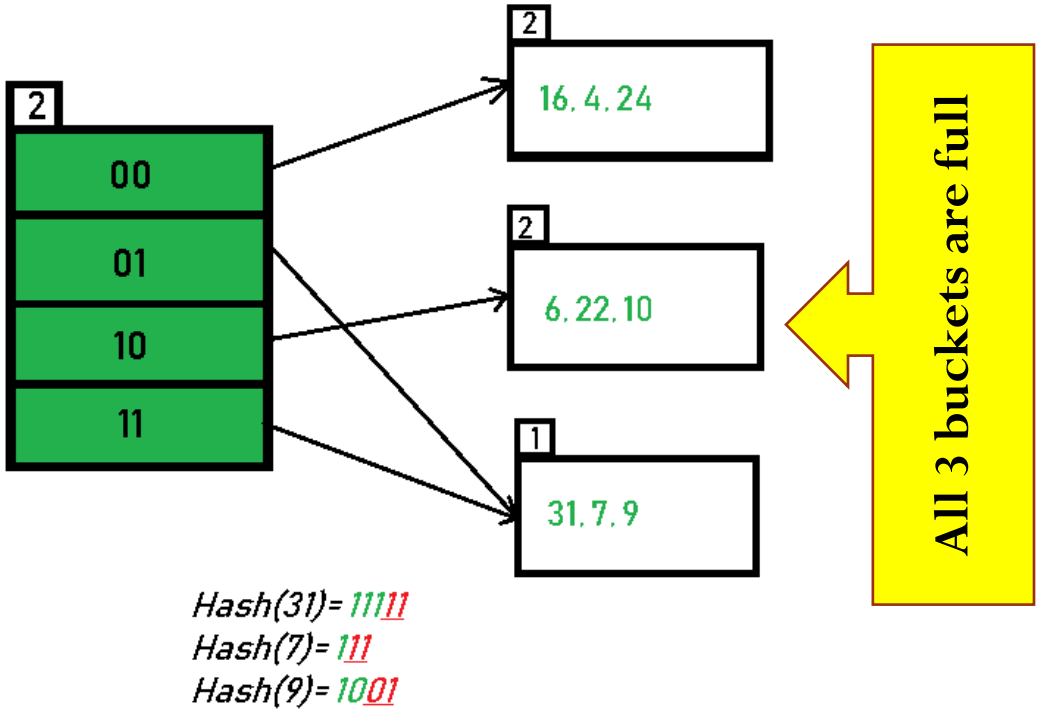
Example # Extendible Hashing : 1/

- **Inserting 24 and 10:** 24(11**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



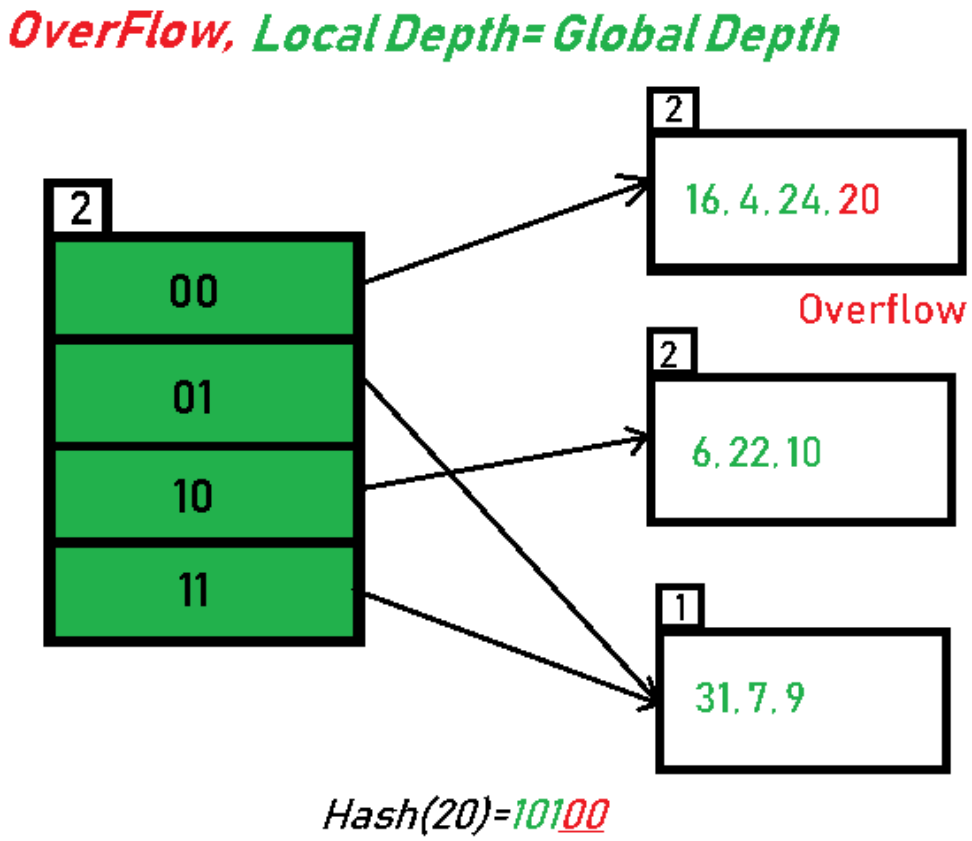
Example # Extendible Hashing : 1/

- **Inserting 31,7,9:** All of these elements[31(11111), 7(111), 9(1001)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



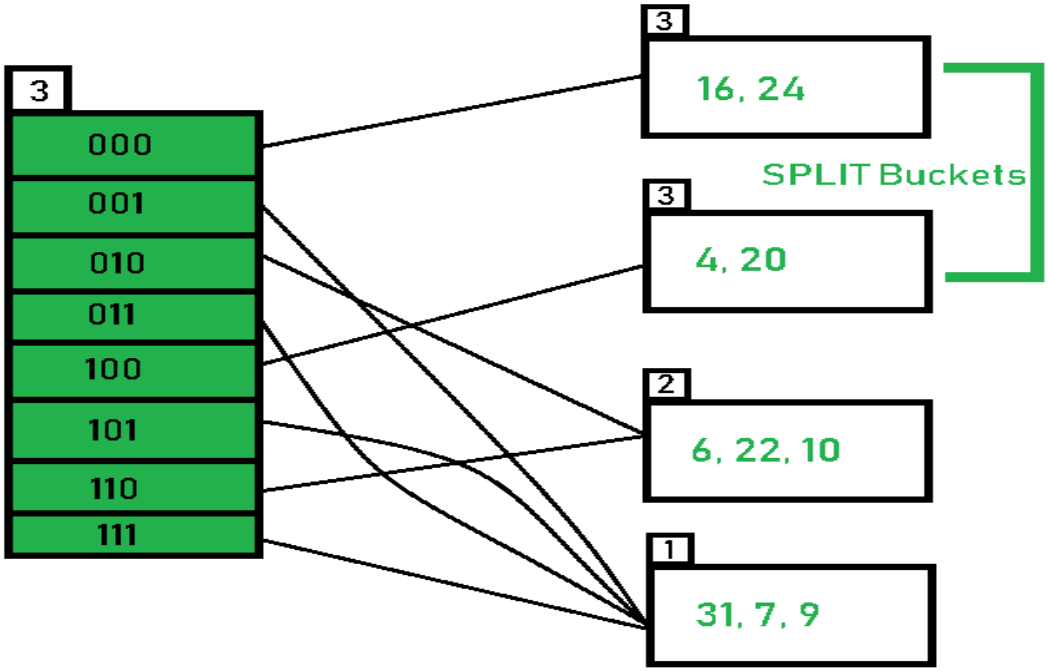
Example # Extendible Hashing : 1/

- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.



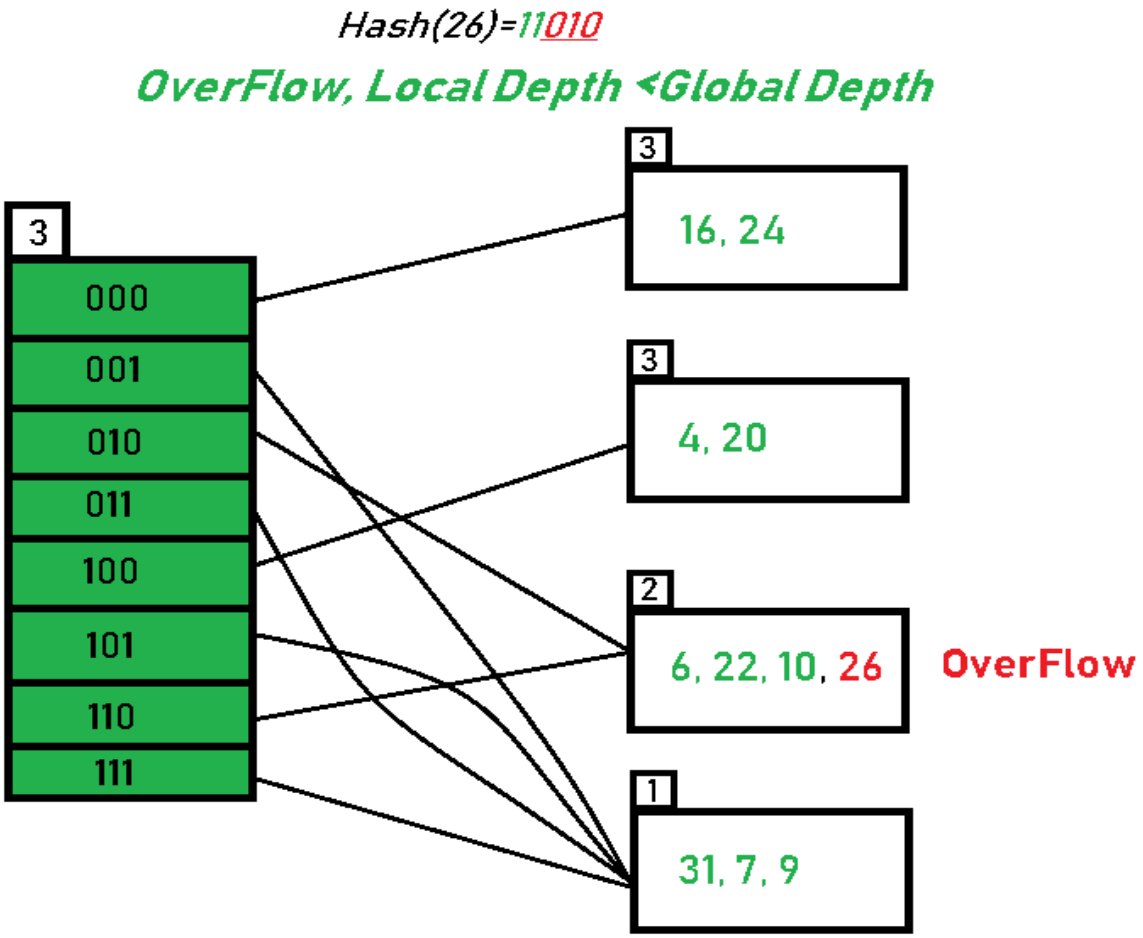
Example # Extendible Hashing : 1/

- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting.
- Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



Example # Extendible Hashing : 1/

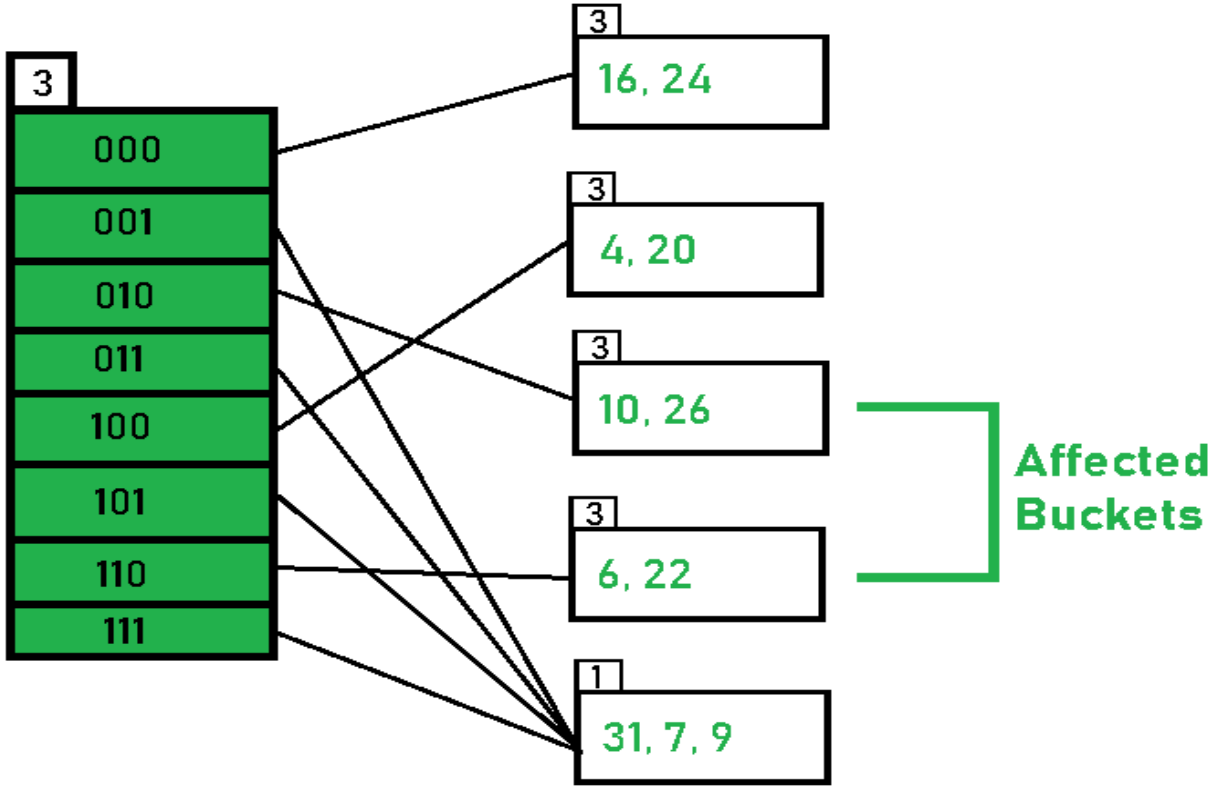
- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



Example # Extendible Hashing : 1/

- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.

6- 00110
22- 10110
10- 01010
26- 11010



Advantages of Extendible Hashing

- When the index exceeds one page only the upper so many bits may be checked to determine if a key hashes to a bucket referred to in this page of the index. Although the mechanism is different than a tree, the net effect is not that much different.
- Extendible Hashing allows the index to grow smoothly without changes to the hash function or drastic rewriting of many pages on disk.
- Data retrieval is less expensive (in terms of computing).
- No problem of Data-loss since the storage capacity increases dynamically.
- With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations of Extendible Hashing

- The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
- Size of every bucket is fixed.
- Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
- When the index is doubled additional work is added to the insertion of a single record. Moreover, when the whole index cannot not fit in memory substantial input and output of pages between memory and disk may occur.
- When the number of records per bucket is small we can end up with much larger global levels than needed. Suppose we have only two records per bucket and 3 records have the same key for the last 20 bits. Since no Local depth of a bucket cannot be bigger than the Global Depth, we will have global depth of 20, even though most of the Local Depths are in range from 1 to 5 bits.

Example 1:

In an extendible hashing environment the directory level – directory depth- (d) is given as 10. In this environment the size of each directory row is given as 4 bytes. Assume that the data bucket size is 2400 bytes and the record size is 400 bytes. Answer the following questions.

[A] In an environment like this how many directory entries would you have (i.e., specify the number of rows in the index table)?

Solution: Directory depth's being 10 means 10 binary digits are used to distinguish the key values of the present records, thus there are $2^{10} = 1024$ directory entries (rows in the index table).

[B] If each directory entry requires 4 bytes what is the total size of the directory?

Solution: As there are 1024 entries and each entry requires 4 bytes; total directory size is 1024*4 = 4096 bytes = 4 KB.

Example 1:

[C] What can be the maximum file size in terms of number of records?

Solution: The maximum file size occurs when each row in the index table points to a separate data bucket and all the buckets are full. As the size of a bucket is 2400 bytes and that of a record is 400 bytes, in each bucket we can have 2400/400 = 6 records. As there are 1024 directory entries, the maximum number of buckets we can have is also 1024 → maximum possible file size in terms number of records = 1024 * 6= 6144 records.

[D] In a file like this what is the minimum number of buckets with bucket depth (p) of 9?

Solution: As discussed above, each entry in the index table could point to a separate data bucket in which case all buckets will have bucket depth of 10. Thus minimum number of buckets with depth 9 is zero (0).

Example 1:

[E] In a file like this what is the maximum number of buckets with bucket depth (p) of 9?

Solution: As we have the directory depth 10, there should be at least two data buckets with bucket depth 10 (which hold the records that have caused a split and increased the directory depth from 9 to 10). All the remaining buckets can have bucket depth 9, which means every two entry in the index table will point one data bucket and we will have the maximum possible number of buckets with depth 9. Thus, maximum number of data buckets with bucket depth 9 = (1024 - 2) / 2 = 511.

[F] In a file like this what is the minimum number of buckets with bucket depth (p) of 10?

Solution: As also discussed in (e), there should be at least 2 data buckets (which hold the records to have caused the split and the increase of the directory depth to 10) with bucket depth 10, to have a directory depth of 10.
→ minimum number of data buckets with bucket depth 10 = 2.

Example 1:

[G] In a file like this what is the maximum number of buckets with bucket depth (p) of 10?

Solution: The maximum number of data buckets with bucket depth 10 is 1024 (the number of entries in the index table), which occurs when each directory entry points to a separate bucket.

[H] In an environment like this what is the minimum and maximum I/O time to read 10 records. Assume that we keep the directory in main memory.

Solution: In this question we keep in mind that we need to read whole buckets whether we want to access a single record or all records stored in a bucket.

In the minimum case, we need only 2 bucket reads as 10 records can be stored at least in 2 buckets. Thus, the minimum time is 2*(s+r+dtt) = 2*25.1 = 50.2 msec. (If we assume that all records are stored consecutively, we could even ignore the s and r)

In the maximum case, we need to make separate bucket accesses for each record to be read. Thus, maximum time is 10*(s+r+dtt) = 251 msec (which could even get larger if we take the maximum rotational latency '2r' instead of r)

Example 2:

In a linear hashing environment we have 30 primary area buckets. Bucket size is 2400 bytes and record size is 400 bytes, LF (load factor) of the file is 0.67 (or 67%).

[A] What is the hashing level h?

Solution: The number of primary area data buckets is always between 2^h and 2^{h+1}, as at some point in linear hashing we have all buckets hashed at the same level and then the addition of records disturbs the balance causing the split of buckets into two, with each new bucket hashed at one level more than the current hashing level (and this goes on until all the buckets are hashed at level h+1, which is when the hash level of the complete table becomes h+1). As we have 30 primary area data buckets →

$2^h <= 30 < 2^{h+1}$ → $2^4 <= 30 < 2^5$ → $h = 4$.

[B] What is the boundary value (bv)?

Solution: The boundary value is the first value yet with hash level h (which is 4 in this example). As we have 14 values (as calculated in (c)) before the boundary value and we start with 0000, bv is the binary equivalent of 14 (with 4 digits) → bv= 1110.

Example 2:

[C] How many primary area buckets are hashed at level h?

Solution: We know that in the linear hash table we have an equal number of buckets hashed at level h+1 at the top and bottom of the hash table, as those buckets come in pairs with the ones having a '0' at the beginning in the top part and those with a '1' at the bottom. The buckets hashed at level h are in the middle in the hash table. Let x be the number of primary area data buckets hashed at level h, then the number of data buckets hashed at level h+1 and starting with a '0' is equal to $2^4 - x$ (which also equals the number of those starting with a '1') As we have 30 buckets as total → $2*(2^4 - x) + x = 30$ → $x = 2$ (2 primary area buckets are hashed at level h(4)).

[D] How many primary area buckets are hashed at level (h+1)?

Solution: As there are 2 primary area buckets hashed at level h, the remaining $30 - 2 = 28$ primary area buckets are hashed at level h+1(5).

Example 2:

[E] How many records do we have in the file?

Solution: As bucket size is 2400 bytes and record size is 400 bytes, each bucket can hold maximum 2400/400 = 6 records. Let M be the capacity of the primary area in terms of number of records → M = 30*6 = 180. We have load factor = 0.67 = 2/3 → load factor = number of records/M → number of records = load factor * M = 2/3 *180 = 120 records.

[F] After inserting how many records does the value of bv (boundary value) change?

Solution: The boundary value changes whenever we try to insert more than bucket factor * load factor number of records. As we have bucket factor = 6 and load factor 2/3 → after inserting 6*2/3 = 4 records (i.e. when we try to insert the 5^{th} record) the boundary value will change.

Example 2:

[G] In this configuration what is the minimum I/O time to access a record?

Solution: The minimum access time occurs if we are able to access the desired record with a single disk access which takes s+r+dtt = 25.1 msec time. (Still better it can be if the disk head is already at the right place letting us also ignore the s (seek) and r (rotational latency)).

References

- https://en.wikipedia.org/wiki/Eight_queens_puzzle
- www.cs.cornell.edu/~wdtseng/icpc/notes/bt2.pdf
- https://www.chelponline.com/dynamic-hashing-12755
- https://www.geeksforgeeks.org/extendible-hashing-dynamic-approach-to-dbms/
- http://bluehawk.monmouth.edu/rclayton/web-pages/s06-503/dhash.html
- http://www.cs.bilkent.edu.tr/~canf/CS351Fall2010/cs351lecturenotes/week6/index.html

Thanks for Your Attention!

Exercises

Suggested Homework

[1] Implement a spelling checker by using a hash table. Assume that the dictionary comes from two sources: an existing large dictionary and a second file containing a personal dictionary.

[2] Output all misspelled words and the line numbers on which they occur. Also, for each misspelled word, list any words in the dictionary that are obtainable by applying any of the following rules:

a. Add one character.

- b. Remove one character.
- c. Exchange adjacent characters.

Suggested Homework

[3] Show the result of inserting the keys 10111101, 00000010, 10011011, 10111110, 01111111, 01010001, 10010110, 00001011, 11001111, 10011110, 11011011, 00101011, 01100001, 11110000, 01101111 into an initially empty extendible hashing data structure with $M = 4$.

[4] Using buckets of size 3 and a hash function of mod(key, 5) and bucket chaining enter the following records (only the key values are shown) into an empty traditional hash file. Create chains of buckets when needed.

42, 57, 16, 52, 66, 77, 12, 25, 21, 33, 32, 14