Search
(Section 8)

Introduction
Basic search techniques
Tree searching
General search trees
Hashing

By:

Pramod Parajuli,
Department of Computer Science,
St. Xavier's College,
Nepal.

Introduction

Internal/embedded key

If a search key is within a given file or table at certain location, then it is known as internal key.

Roll no.	Name
1	Ram
2	Shyam
…	…
N	XYZ

External key

If we are working in an index file or table and the actual records are in another file/table, then it is known as external key.

Main table

Roll no.	Class ID
1	C8
2	C6
…	
N	C8

Student detail

Roll no.	Name
1	Ram
2	Shyam
..	..
N	XYZ

Class detail

Class ID	Description
C8	B.Sc. II
C6	B. Sc. I
..	..

Primary key and secondary key

If a given file is to be sorted according to a single field then the field is known as **primary key**. The values of such field should not duplicate. If the field that we are searching contains duplicate entries, then to find the right record, we must specify another field that purifies the search. The extra field that is used to purify the search is known as **secondary key**.

The general idea behind search is just to search a given key in a given file. If the key is found, then the address or other related records are returned, otherwise NULL value is returned. Some algorithms add the search key if the key can not be found in the file. Such algorithm is known as **search and insertion** algorithm.

If the entire search file or table is in main memory then it is known as **internal search** and if the search file also spans to storage media, then it is known as **external search**.

Basic search techniques

Sequential searching

Sequential search is the simplest kind of searching technique. Here, a search key is defined and records in a file are compared with the key. If the key is found, then the position is returned otherwise NULL or -1 value is returned.

A file can be an array or a linked list. In case of an array, the search key is compared with the first element of the array and is continued up to the last one. In linked list also, the comparison is done from start node and continued until end of the linked list is not found.

Array

```
int seqSearch(int array[], int n, int key){

      for(counter = 0; counter < n; counter++){
            if(key == array[counter]){
                  return counter;
            }
      }
      return -1;
}
```

Linked list

```
linkedList *seqSearch(linkedList *start, int key){
      linkedList *tempNode;

      if(start == NULL){
            return NULL;
      }

      for(  tempNode = start;
            tempNode != NULL && key != tempNode -> data;
            tempNode = tempNode -> next);

      return tempNode;
}
```

For some implementation, the search key is added at the end of the current file so that the search process will definitely get the key. In this case, anyone is sure to get the search key. Therefore, we are sure to get the search key.

```
int seqSearch(int array[], int n, int key){
      array[n] = key;

      for(counter = 0; array[counter] != key; counter++){
      }

      if(counter < n)
            return counter;
      else
            return -1;
}
```

Same logic is also implemented for linked list implementation.

Optimization

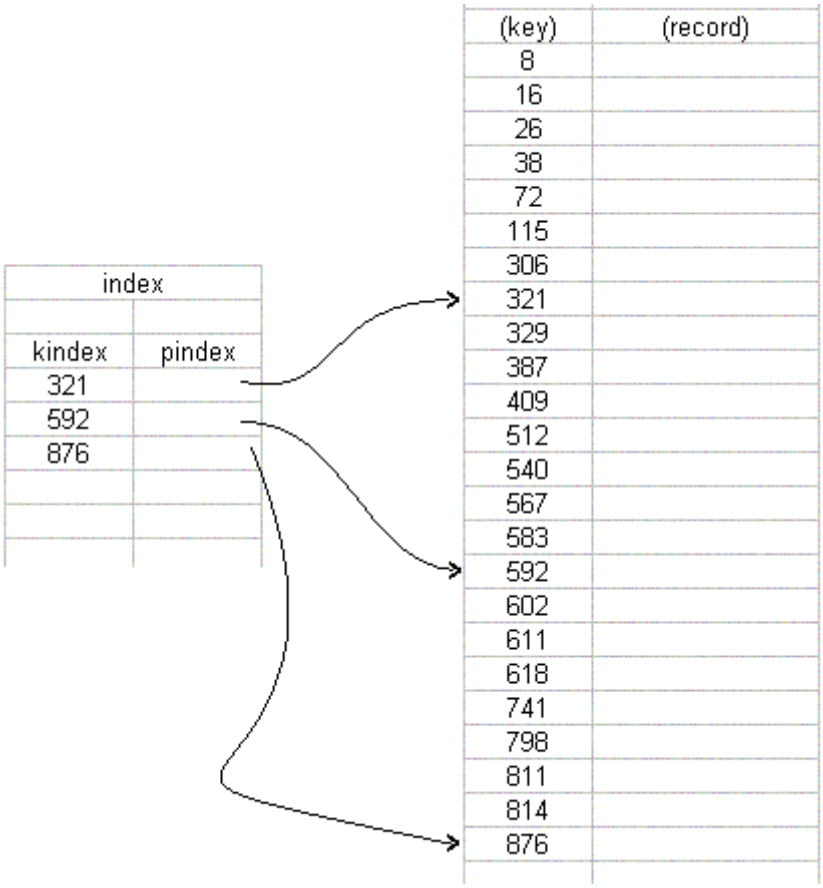
Put most frequently used data at the beginning of the file. To do this, create a field called 'priority' or 'probability' and assign high priority/probability for most frequently used data. If the file is disturbed very much, then run a managing function that sorts the key according to the priority.
Another way can be to move a successful search key to the front. It is known as '**move-to-front**'.
Third method is to exchange a successful search key with the key that precedes it. This way, most frequently used key will automatically move to front. It is known a '**transposition.**'

Benefits of ordered list in sequential search

- It is possible to check for the presence of a search key (just check the lower and upper boundary) and only after that begin the search.
- The search time is reduced to N/2 on the average case.

Indexed sequential search

If the file is very much large, then an auxiliary file is created called as 'index' and pointers to the original file are stored. Now, our search will take place in the index file. The key is found and to read the whole record, the original file is referenced.



The index file contains a key-index field of only selected data. At a given particular instance, the search is only needed for 321, 592, and 876. The index file also contains a pointer-index that contains the actual address of the keys in the original file. To search the record for key 592, the address is searched in the index file and then the record of the key 592 is directly read from the original file.

Many indexed file system (NTFS and others) use indexed (dynamic) addressing to allocate the files faster.

If the original file is very large, then secondary index are created and search is done.

To delete a record, it is quite unwise to rearrange all the records. Rather, the key is deleted from index file only and then in original file, a flag is to indicate the validity of the record.

Insertion in the original file might cause lots of problems as there might not be any slot between two index values in the index files. Solution to this problem could be to search for the nearest deleted file or increase the size of the index file as in hash file. We will see how the size of the file is increased in '**hashing**' section.

Binary search

We already have seen binary search. Have a look at binary **search sub-section** in 'Recursive' topic – page – 11.

In case of indexed sequential search, since the index keys are sorted, use of binary search algorithm to find the index keys also increases the performance of the indexed sequential search. But, if lots of insertions and

deletions occur, then binary search can not be implemented. Also, only in case of the array implementation, binary search can be implemented.

Interpolation search

Interpolation search is also applied over ordered array only. The general idea is same as of binary search but the calculation of the 'mid' pointer is slightly different and quite efficient. Instead of the direct calculation of the 'mid' pointer, the value is predicted according to the existence of the items in the array as;

$$mid = low + (high - low) * \frac{(key - array[low])}{(array[high] - array[low])}$$

After finding the value of mid, the value is compared and then left sub-file and right sub-file are created. Again, the search is done until if we don't find the value or low < high.

The time complexity for the comparisons is around $\log_2(\log_2 n)$ for uniformly distributed elements. If the elements are not uniformly distributed, then the performance of interpolation search is degraded to that of sequential search.

For non-uniformly distributed elements, the performance can be improved by using **robust interpolation search** (**fast search**). The problem in interpolation search is that, the difference between lower boundary and upper boundary can go down up to 1 and 0 also. If this happens, then it works as sequential search. The general idea behind fast search is to maintain a gap between lower and upper boundary so that whenever the search procedure jumps to either left or right, it reduces the search region significantly.

To do the task, a new variable '**gap**' is introduced. Value of gap is maintained so that it will always be less than (mid-low) and (high-mid).

Algorithm

```
1. Set, gap = sqrt(high – low + 1)
2. Repeat steps 3, 4, and 5 until low <= high
3. temp (possibility) = low + (high – low) * (key – array[low]) / (array[high] – array[low])
4. mid = min (high – gap,    max(temp, low + gap))
5. if(key < array[mid])
         high = mid – 1
         gap = sqrt(high – low + 1)
   else
         low = mid + 1
         gap *= 2
6. Finish.
```

Fast search improves the search time significantly up to O(log log n). But if the keys are uniformly distributed, then the performance drops due to calculation overhead of temp, mid, and gap. In worst case, it gives efficiency of $(O(\log n)^2)$.

Tree searching

We already have already seen the logic behind addition, deletion, and maintenance of binary trees. Here, we will discuss on efficiency of trees only.

If the tree is a complete or strictly binary tree, then the performance goes up to O(logn). But, for skewed trees, the performance is reduced to O(n).

General Tree searching

We saw definition of general tree in 'Trees' section. One example given was 'B-trees.'

Multiway search tree

A multiway search tree is a node that might contain more than one element and more than two children. The children must be within the range of the given elements between which they lie. All of the children are also multiway search tree.

While defining multiway search tree, we specify the order of the tree. A multiway search tree of order N may contain <= N children and < N number of elements.

Refer to fig on page 425.

Semi-leaf: A node with at least one empty child.

B-tree

Balanced multiway search tree. If the order of B-tree is given to be 'N', then;

1. All leaf nodes are null children.
2. All leaf nodes should be in same level.
3. All internal nodes may have at most 'N' children and at least N/2 children except root node. Root node may have at least two children i.e. with only one key value.
4. Should maintain BST properties.

Binary tree is a 2 – tree with all children at same level.

Illustration

Insertion

While inserting the elements in the B-tree, the value is compared with the existing values in the node and then proper place is found using tree traversal. After the value is kept on the node, the node must be checked for validation of B-tree property. If the number of elements is more than N-1, then mid element is upgraded to the parent node.

If a node becomes invalid due to less number of elements, then it is merged with elements in other nodes.

Class demonstration:

Make a B-tree of order 5 inserting the keys:

10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 240, 30, 120, 140, 200, 210, 160

Ex: Show result of inserting keys into initially 3-2 tree (B- tree of order 3).

3, 1, 4, 5, 9, 2, 6, 8, 7, 0

Show deletion of 0 & 9 in order.

Deletion

For deletion, just remove the value. After removing, the B-tree must be balanced so that it maintains B-tree property. To do the job, the elements are upgraded to parent, degraded to the child, and merged with elements in other nodes.

Homework 6: Insert into B-tree of
- i. order of 3
- ii. order of 5

659, 767, 702, 157, 728, 102, 461, 899, 920, 44, 774, 264, 384, 344, 973, 905, 999

B^+ trees

In B-tree, when we require reading all of the elements, then the traversal is quite inefficient due to back-n-forth movement between parent and children. To improve the efficiency of B-tree, all of the elements are stored in the leaf nodes and the leaf nodes are linked together using a linked list. This way, the performance while reading the element is improved.

Refer to fig on page 462.

Hashing

Until this time, we stored the data in some location, set a search key, and find the search key by comparing with the elements in the array or linked list. The process of comparing and calculating possible location of the desired key is quite time consuming. Hashing is a technique that allows direct access of the search key.

For this purpose, the key is stored in such a way that future retrieval is fast i.e. direct access.

Features of hashing;

- an algorithmic approach
- searching time: independent of 'n'
- oriented to file management

The general idea behind hashing is to store the elements in an array in such a way so that the value of the element itself determines appropriate storage location.

0	100
1	181
2	872
3	653
…	…

If the search key is 872, then just calculate the radix for one's position and then access (array + 2 * sizeof(data)) address.

If the number of elements 'N' is large enough, then a huge amount of the space is required to hold the elements in this fashion. Therefore, to have good optimization of the space and **collision resolution** hashing algorithms are used.

The function that is used to determine the appropriate place for the element is known as **hash function**.

A common hash function (direct/open addressing)

hash value = (key) mod (table size)
= key % table_size

The chances of collision in open addressing is very much high. When the number of elements goes larger than that of table size or two numbers with same radix appear, then there will be collision. To resolve the collision, linear probing technique is used.

Linear probing

Search for next available address. For the purpose, if f(i) is the increment function, then calculate for i if collision occurs. However, if two or more elements compete for same hash location, then there will be clustering problem.

Note:

The size of element i.e. number of digits should not have any impact on the size of table. For example, if the size of largest element that is stored in the table is 10 digits, then there might be total of 200 elements only.

Hash algorithms

1. **End folding**

   Suppose, ten digit key: 1234567890 which may be reduced to fit 50,000 storage locations by carrying out the steps.

   i. 12 | 34567 | 890
   ii. 34 567
   12 890

   47457

   iii. Result > 50,000 ?
   iv. Yes: subtract 50,000 (until > 50,000)
   v. No: address ← result.

2. **Mid-square hashing algorithm**

2.1 Square the key and select part of it

Suppose key values 4-digit numbers and maximum number of records is 100. Now, let's see how the keys are transformed into address;

Mid square hashing transformation

Key	Square	Address	
1456	21,19,936	19	
1892	35,79,664	79	
9467	896,24,089	24	←homekey
..	..	..	
2434	59,24,356	24	←synonym
9672	935,47,584	47	

2.2 Square & fold

$(821455)^2$ = 6,74,78,831

6747
8831

15578

Now, address = 15578 % 1000 = 5578 (assuming 4-digit address).

3. Division by prime

i. Decide a packing density (load factor) for a file and calculate N.

Let, n be 85,000 (number of records)
N is required addresses.

Packing density (a) = n / N Therefore, N = n / a

Suppose a suitable packing density be 85% i.e. 85% of space fill is desired.

Storage position required = (85,000 * 100) / 85
= 100,000

ii. Select a prime near to 100,000, say 99991.
iii. Perform (key mod prime) i.e. key % 99991.
iv. Now, the remainder lie between 0 – 99991 that are the addresses.

e.g. Elements are 4 digit numbers and maximum number of records = 100

For this system, prime number selected: 97.

key	address	
1456	1	1456 % 97
1892	49	1892 % 97
…	…	…
3646	57	
7875	18	
9467	58	
2434	9	
9672	69	

Collision Resolution techniques

Double Hashing

a) When collision occurs, a new hash function is defined.
$h_2(x) = R - (x \bmod R)$
where, R is nearest prime number from its table size.

If collision occurs again, probe linearly.

b) Strategy: increment function: $f(i) = i.\ h_2(x)$
for i^{th} collision, new has value is obtained by:

(original hash value + $i.h_2(x)$) % tableSize
hash function = x % 10

	6
	5
	4
43	3
	2
11	1
	0

putting 83, we will have collision.

So, $h_2(x)$ = 7 – (x mod 7)
= 7 – (83 % 7)
= 1 (so put in 1)

Re-hashing

a) If collision, compute new hash function using old hash values as input to a rehash function like;

$h_r(x)$ = (old hash value + constant) % tableSize

b) When packing density (load factor) is given, (say 70 %)

Resolute collision linearly, but when any insertion exceeds packing density, new table is created whose size is nearest prime that is twice as large as the old table. The old table is scanned and all elements are inserted into new table.

e.g. Input: 13, 15, 6, 24: hash function: x % 7

6	0
15	1
	2
24	3
	4
	5
13	6

Now, inserting 23, at (23 % 7 = 2) location, we will have packet density reach > 70%.
So, new table size = 17 (nearest prime of double old table size)

Now, has table will be:

	0
	1
	2
	3
	4
	5
6	6
23	7
24	8
	9
	10
	11
	12
13	13
	14
15	15
	16

and, new hash function: x % 17

Quadratic Probing

- Attempt to correct problem of clustering
- Forces the problem key move a considerable distance from the initial collision.

Strategy:

$$f(i) = i^2$$

For i^{th} collision,

$$\text{re-hash value} = (\text{old hash value} + i^2) \% \text{ table size}$$

Open Hashing (chaining)

All of the above hashing use fixed number of storage space.

- Hash table as an array of pointers
- Each entry in the table is linked with a linked list. Synonyms are attached to the end of the linked list.

Exercise all of the hashing methodology for given input

Input:		1, 16, 49, 36, 25, 64, 0, 81, 4, 9
Hash function:		h(x) = x % 10
Double hashing	$h_2(x) = 7 - (x \bmod 7)$	
Re-hashing	a)	$h_r(x)$ = (3 + old hash value) % 10
	b)	packing density = 75 %

Disjoint Sets

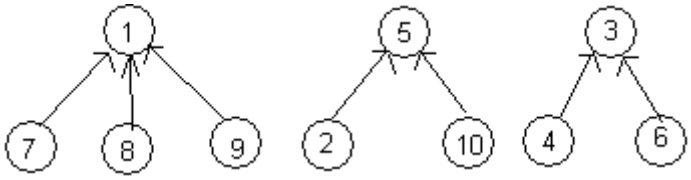
If Si and Sj, such that i ? j, are two sets, then there is no element that is both in Si & Sj.

e.g.

n = 10

S1 = {1, 7, 8, 9}, S2 = {2, 5, 10}, S3 = {3, 4, 6} are three disjoint sets.

One possible tree representation of the sets can be:



Note: for each set, linkage of nodes from children to parent, rather than parent to children.

The operations, we are interested are:

1. Disjoint set union

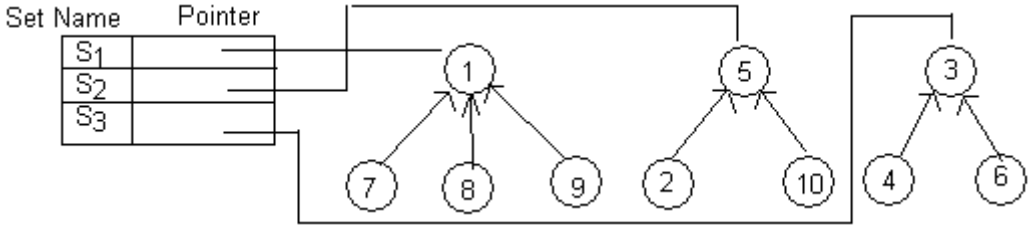
   Si U Sj = all elements 'x' that is in Si or Sj.

   After operation Si U Sj, the sets Si & Sj does not exist independently, they are replaced by Si U Sj in the collection of sets.

2. Find(i)

   Given the element I, find the set containing 'i'. Thus 4 is in set S3 & 9 in set 1.

Data representation for S1, S2, and S3



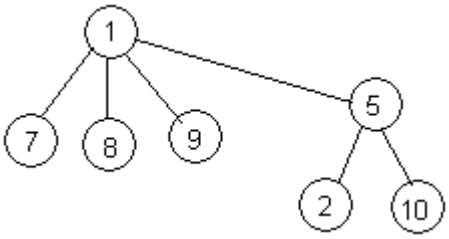
Array representation of S1, S2, and S3

- Array element gives the parent pointer of the corresponding tree.
- Notice that the root nodes have a parent of -1.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
P	-1	5	-1	3	-1	3	1	1	1	5

Union & Find operations

- make one of the trees a sub-tree of the other.



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
P	-1	5	-1	3	1	3	1	1	1	5

Simple algorithm for union:

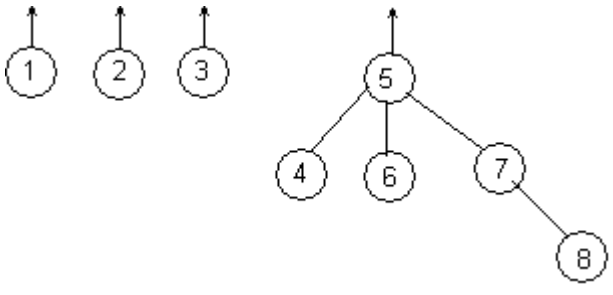
```
simpleUnion(int i, int j){
      P[i] = j;
}

simpleFind(int i){
      while(P[i] >= 0)
            i = P[i];
      return i;
}
```

Smart Union Algorithms

Union by weight (size)

- If the number of nodes in the tree with root 'i' is less than that of 'j', make 'j' the parent of 'i', otherwise make 'i' the parent of 'j'.
- To implement the weighing rule, maintain count field (no. of nodes of a tree) in the root of every tree as a negative no. instead of -1.

Consider the trees:



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
P	-1	-1	-1	5	-5	5	5	7

Union algorithm with weighing rule

```
weightedUnion(int i, int j){
// union sets with roots i.e. j, i!=j      using the weighing rule,
// P[i] == -count[i] and   P[j] == -count[j]

      int temp = P[i] + P[j];
      if(P[i] > P[i]){                          // 'i' has fewer nodes
```

```
            P[i] = j;
            P[j] = temp;
      }
      else{
            P[j] = i;
            P[i] = temp;
      }
}
```

i			[3]		[5]			
P			5		-6			

Ex: Define union by height (rank): Put –ve height.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
P	0	0	0	5	-2	5	5	7

Ex: Show the result of the following sequence of instructions.

Union(1, 2), Union(3, 4), (3, 5), (1, 7), (3, 6), (8, 9), (1, 8), (3, 10), (3, 11), (3, 12), (3, 13), (14, 15), (16, 17), (14, 16), (1, 3), (1, 14) when unions are performed.

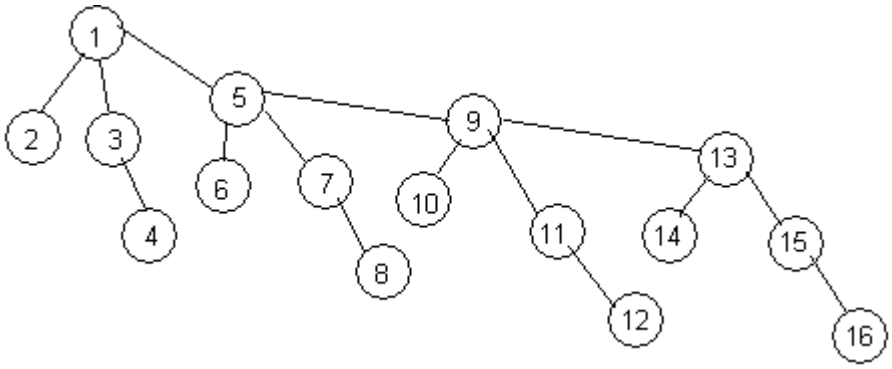
a) arbitrarily
b) by height
c) by size

Path Compression

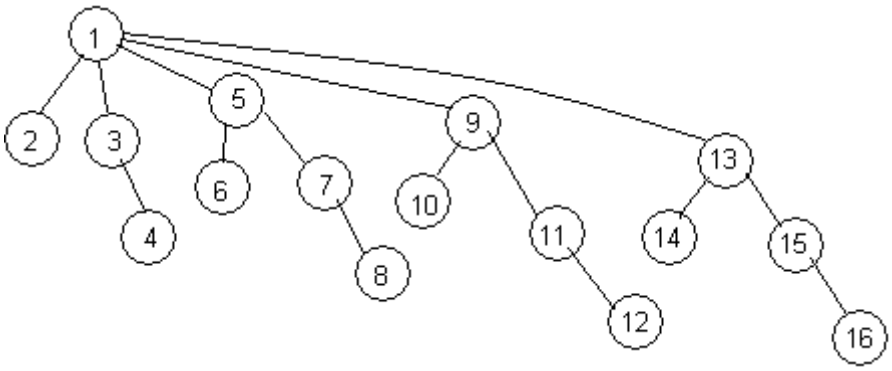
- Deep node brought near the root
- Is performed during a find operation and is independent of the strategy used to perform unions.

Suppose the operation is find(i). Then the effect of path compression is that every node on the path from 'i' to the root has its parent changed to the root.

Consider the following worse case tree for n = 16.



Effect of path compression after find(15)



Find algorithm with collapsing rule:

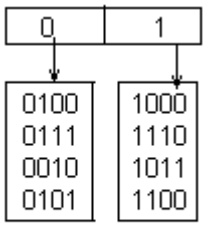
If 'j' is a node on the path from 'i' to its root and P[i] ? root[i], then P[i] to root[i].

```
Collapsing_find(int i){
/* find the root of the tree containing 'i'.  Use path collapsing rule
to collapse all nodes 'i' to the root */
      int r = i;
      while(P[r] > 0)
            r = P[r];

      while(i != r){
            int s = P[i];
            P[i] = r;
            i = s;
      }
      return (r);
}
```

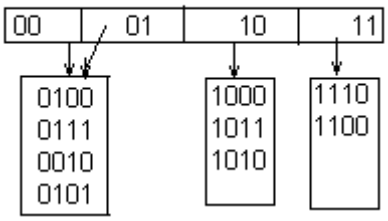
Extensible hashing

A mixed up concept of B-tree and Radix sort.

Keys per block (m) = 4
Insert keys of 6-bit lengths



After inserting 1010



Break only fulfilled table. If the table itself is large, again use internal indexing.