Chapter-10

Images and texts

INTRODUCTION TO TIME-SPACE TRADEOFFS

- Sometimes, faster data retrieval is required in database applications.
- Algorithms that guarantee faster data retrieval can be designed at the cost of additional memory space.
- An algorithm needs be space or time-efficient, depending on the application context.
- Sometimes it is not easy to design efficient algorithms about time and space.
- Selecting one kind of efficiency for a given algorithm over the other is called a time–space tradeoff.
- Retrieval time can be reduced at the cost of additional memory space in the following two ways:

Input enhancement

- **Input enhancement** It is also called preconditioning or preprocessing.
- It is a technique for improving an algorithm’s performance by acquiring additional input data statistics.
- The additional information may require extra memory space, but this requirement can be justified as this approach drastically reduces retrieval time.

Prestructuring

- **Prestructuring** is another technique for enhancing the time efficiency of algorithms by improving access structures based on prior computations.

Conception of Linear Sort

- The best performance of a sorting algorithm is O(n log n), provided by merge sort and quicksort. Linear sorting aims to improve performance by sorting in linear time O(n).
- It is possible to sort in linear time using the concept of time–space, which requires additional space.

Counting Sort

- Counting sort is a distribution-based linear sort based on the ranking concept.
- For a number x in an array, the x rank is computed.
- The rank of an element x is defined as the number of elements that are less than x plus one.
- Thus, the rank indicates where x should be placed in the target sorted list.
- Counting sort is an example of time–space tradeoff, as the input data statistics is collected and used for sorting purposes

Algorithm Counting Sort

Algorithm countingsort(A[1 .. n])

```
%% Input: Array A[1 .. n] whose elements are to be sorted
%% Output: B[1 .. n] is the sorted output of array A
Begin
     max = findmax(A[1 .. n])          %% Find the maximum of array A
     for i = 1 to max                  %% Initialize the counter array C
         C[i] = 0
     End for
     for i = 1 to max do               %% Find frequency
         C[A[i]] = C[A[i]) + 1
     End for
     for i = 2 to max do               %% Find the cumulative frequency
         C[i] = C[i] + C[i - 1]
     End for
     for i = max to 1 step - 1         %% Distribute the elements
         element = A[i]
         B[C[element]] = element
         C[element] = C[element] - 1
     End for
End
```

Example – Use counting sort and sort the elements of an array

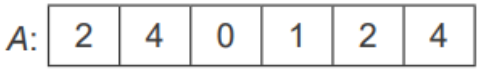
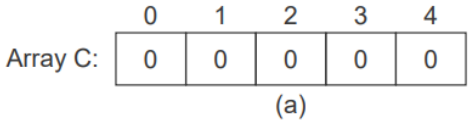


Fig. 10.1a Initial Array



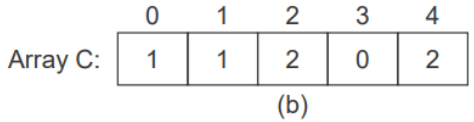


Fig. 10.1b Count sort (a) Initial array *C* and (b) Frequency of array

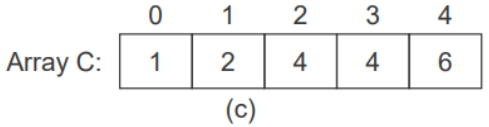


Fig. 10.1c Cumulative Frequency of Elements

Elements	0	1	2	3	4
Frequency	1	1	2	0	2
Cumulative frequency (or distribution)	1	2	4	4	6

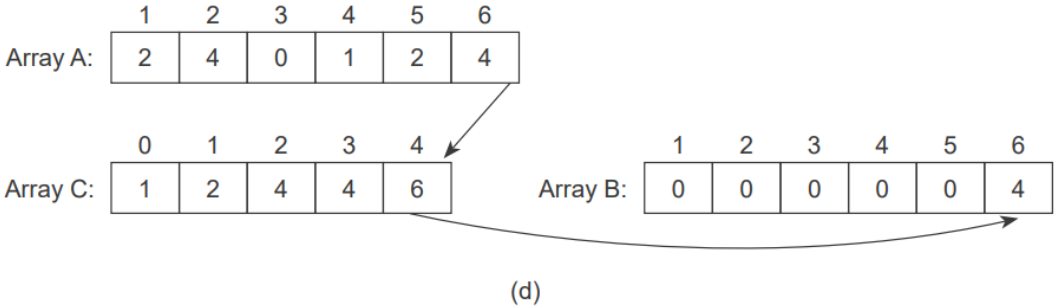


Fig. 10.1d Counting sort (d) Distribution of element 4

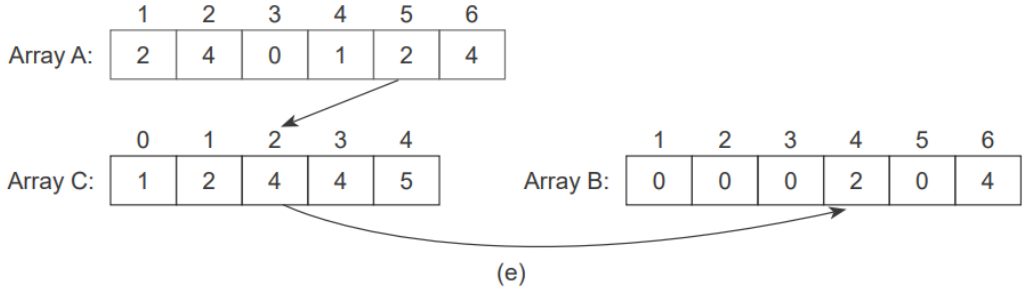


Fig. 10.1e Counting sort (e) Distribution of element 2

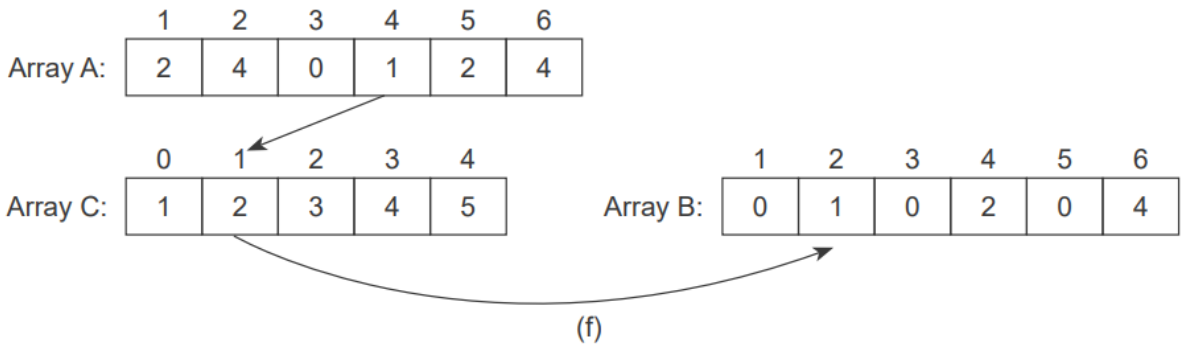


Fig. 10.1f Counting sort (f) Distribution of element 1

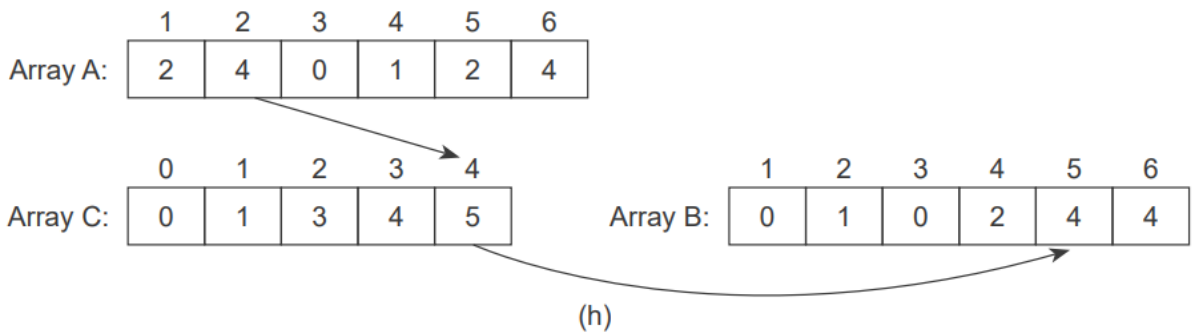


Fig. 10.1h Counting Sort (h) Distribution of Element 4

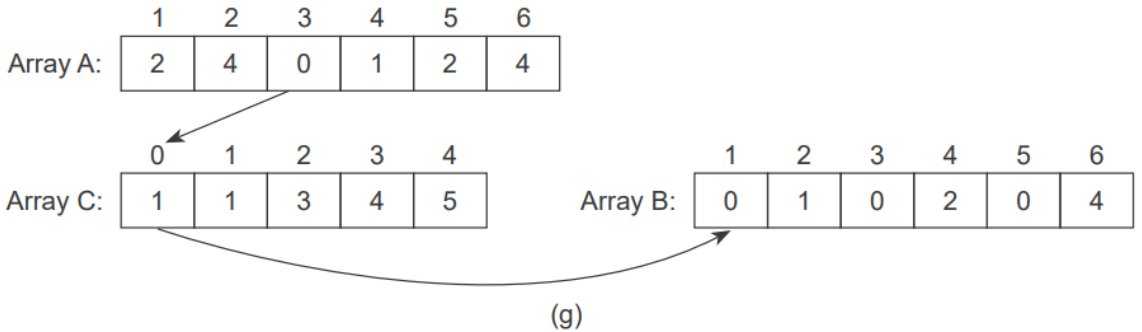


Fig. 10.1g Counting sort (g) Distribution of element 0

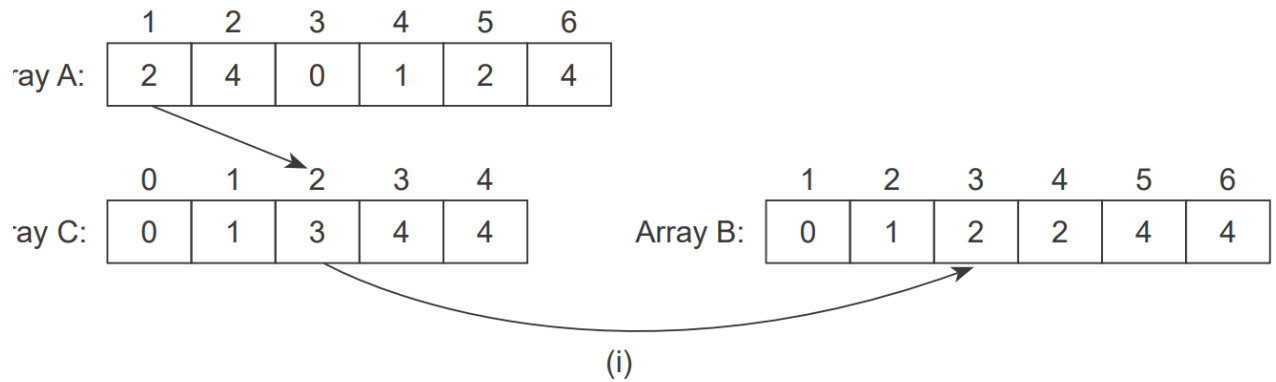


Fig. 10.1i Counting sort (i) Distribution of element 2

Advantages and Disadvantages and Complexity

- **Advantages**
  - The advantages of counting sort are that it is very stable and based on the distribution concept.
- **Disadvantage**
  - It is only effective if the elements of an input array have a limited range.
  - Its disadvantage is that counting sort depends on the input data
- **Complexity Analysis of counting sort**
  - The run time of counting sort becomes O(n)

Bucket Sort

- Bucket sort is another non-comparison-based sorting algorithm for sorting numbers, strings, and other data types in linear time.
- Bucket sort works better if the elements are in a fixed range and have no redundancy.
- Bucket sort is an example of linear sort, as it uses an additional space for sorting.
- Hence, it is an example of a time–space tradeoff.

Informal Procedure for Bucket sorting

- **Step 1**: Given an array A, determine the array length, which is assumed to be n.
- **Step 2**: For each element e of array A, do the following:
  - **2a**: If bucket[i] is empty, put the element e into the bucket.
  - **2b**: Otherwise, use insertion sort and put element e into bucket B[i].
- **Step 3**: Concatenate all the buckets into a sorted list.

Bucket Sort Algorithm

Algorithm bucketsort(A[1 .. n], m)

```
%% Input: Unsorted array A[1 .. n] and m is the number of buckets
%% Output: Sorted array A
Begin
  %% initialize m buckets
  for i = 1 to m do
    bin[i] = ∅
  end for
  %% Insert elements
  for i = 1 to n do
    bin[A(i)] = A[i]
  end for
  %% Combine and print sorted values
  for i = 1 to n do
    print bin[A(i)]
  end for
End
```

Example - Let us assume that the numbers to be sorted are as follows:
{8, 17, 24, 50, 48, 42}

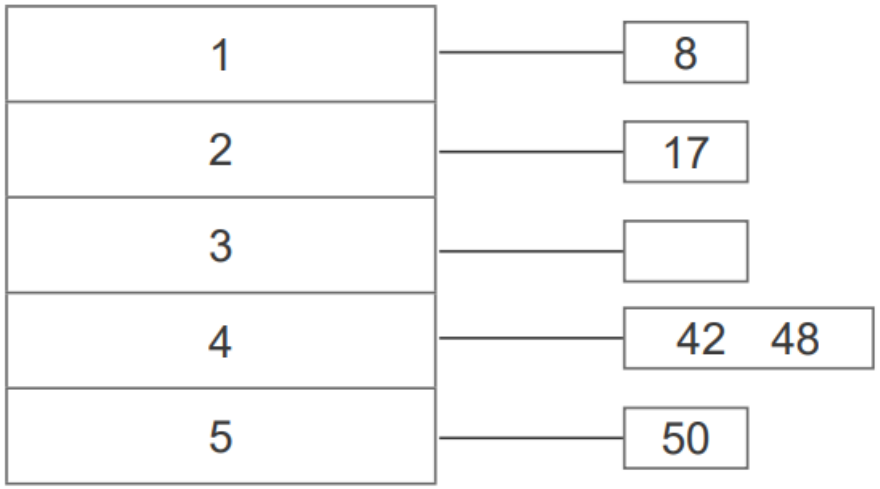


Fig. 10.2 Bucket sort

Example- Let us assume that the numbers to be sorted are as follows:
0.18 0.12 0.32 0.42 0.41 0.47 0.83 0.92 0.97

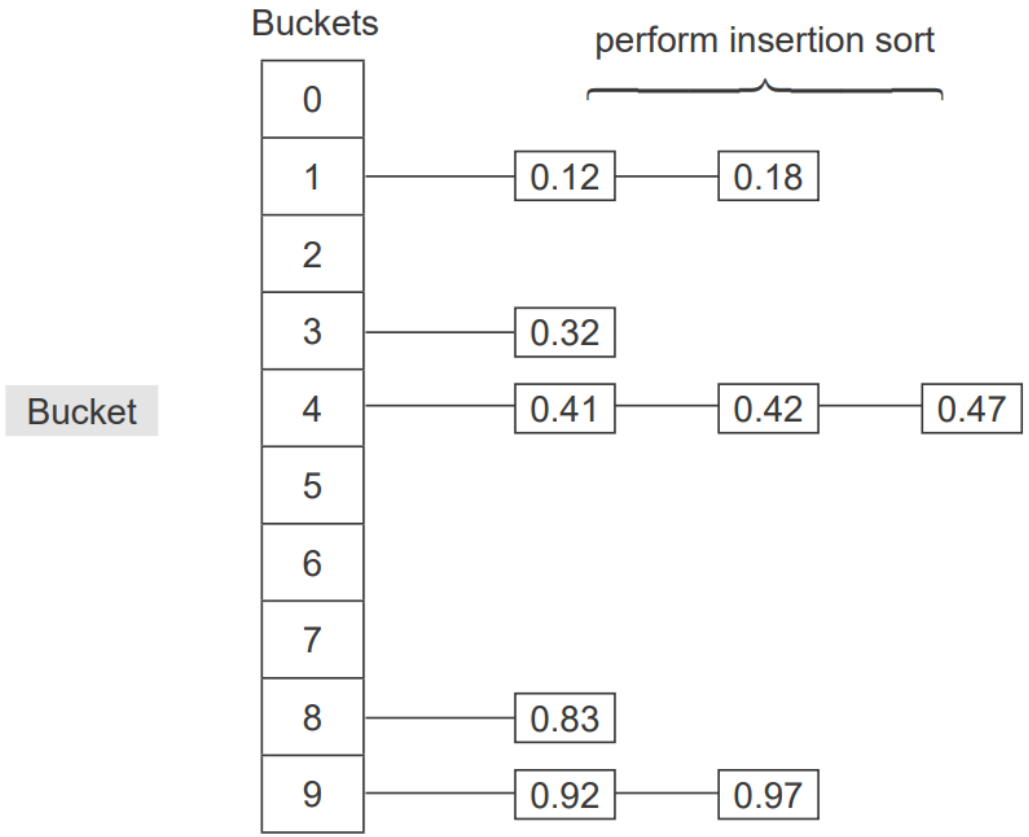


Fig. 10.3 Distribution of numbers into appropriate buckets

Advantage and Disadvantage and Complexity

- **DISADVANTAGE:**
    - The advantage of bucket sort is that it works well for a limited range of data
- **DISADVANTAGE:**
    - It depends on the input data.
    - The number of buckets is an issue because if the input elements have a wide range, more buckets are required, increasing the algorithm's time complexity
- **COMPLEXITY:**
    - The final complexity of the algorithm is $O(n + m) \approx O(n)$ time, assuming that the number of buckets is small

Radix Sort

- Radix sort is another distribution-based sorting technique
- In other words, the sorting is not based on the comparison operator
- This sort takes the structure of the numbers that need to be sorted. By structure, it means the numbers' radix are considered
- Recollect that the radix of a decimal number is 10, and the radix of the English alphabet is 26.
- The idea is to maintain buckets of size that equal the radix
-

- Then, every individual digit of each digit is considered and distributed
- There are two types of distribution, as shown Below

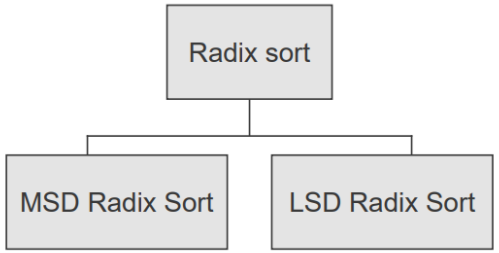


Fig 10.4 Types of Radix Sort

- In MSD (Most Significant Digit) radix sort, the distribution of the digit is based on the most significant digit (From left to right most digit) to the least significant digit.
- MSD radix sort is useful for strings

- On the other hand, in LSD (Least Significant Digit) radix sort, the distribution is from the right-most digit to the left-most digit
- LSD radix sort is useful for numbers

Informal passes

- **Step 1:** The least significant digit in 1’s position is distributed to one of the respective buckets. After that, the numbers are collected from the buckets sequentially, and the second pass starts
- **Step 2:** The middle digit, the digits at position 10’s, is distributed. After that, the numbers are collected from the buckets sequentially, and the second pass starts
- **Step 3:** In the third pass, all digits at the position of 100’s are distributed. As the given numbers have a length of 3, the algorithm terminates. The algorithm can generally be extended to the length of the given digits.

Example-Use radix sort to sort the following set of numbers of array A:
606, 301, 762, 543, 342

Table 10.2 Distribution of numbers based on the LSD

Index	Elements	Index
		0
3 0 1	301	1
	762,342	2
7 6 2	543	3
3 4 2		4
		5
5 4 3	606	6
6 0 6		7
		8
		9

Table 10.3 Distribution of numbers based on the middle digit

Index	Elements	Index
	301, 606	0
3 0 1		1
		2
7 6 2		3
3 4 2	342, 543	4
		5
5 4 3		6
6 0 6		7
		8
	762	9

Table 10.4 Distribution of Numbers based on the MSD

Index	Elements	Index
		0
		1
3 0 1		2
6 0 6	301, 342	3
3 4 2		4
	543	5
5 4 3	606	6
7 6 2	762	7
		8
		9

Informal algorithm for Radix Sort

- **Step 1**: Read the set of numbers
- **Step 2**: Let radix be r, and the number of digits of the given elements is d
- **Step 3**: Create r queues numbered Q[0] to Q[r]
- **Step 4**: For all digits i = 1 to d:
  - **4a**: Distribute the numbers into appropriate buckets
  - **4b**: Collect the numbers from the buckets sequentially.

Algorithm for radix sort

Algorithm radixsort(A[1 .. n])

```
%% Input: Array A[1 .. n] of unsorted numbers
%% Output: Array A of sorted numbers
Begin
   create queues Q[0] .. Q[r]                          % r is the radix, 10 queues need %
                                                       %% to be created for r = 10
   for k = 1 to d                                      % d is the maximum number of %
                                                       % % digits of numbers
          for i = 1 to n do
               digit = extractdigit(A[i])              %% Extract digits and distribute
               Q[digit] = enqueue(A[i])                %% Enqueue the number
         End for
         for j = 0 to r                                %% Collect the numbers
             while Q[j] is not empty
                   Append Q[j] to array A
              End while
         End for
   End for
End
```

Complexity Analysis

- For d number of passes, the running time is given as O(d(n + r)).
- As d is a constant, it can be observed that the algorithm’s running time reduces to linear time.

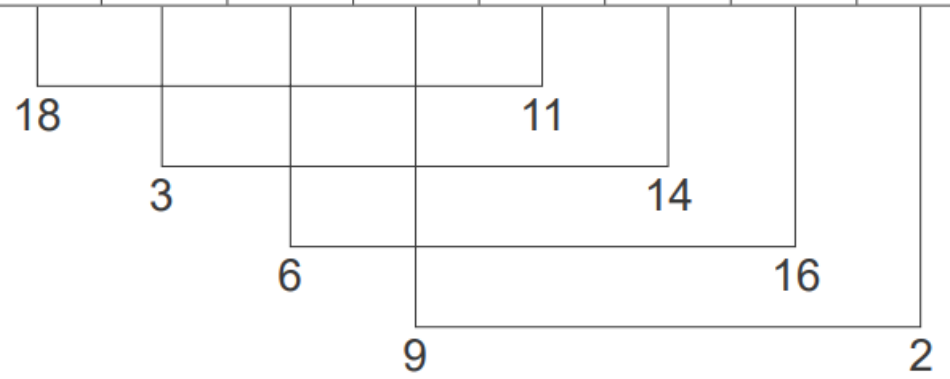
Shell Sort

- Shell sort is another example of a time-space tradeoff
- It is an in-place comparison-based sorting.
- In this sort, the far-apart elements are compared instead of only adjacent elements
- This sort is also known as the diminishing increment sort
- The sorting method depends on the concept of a gap
- The gap specifies the elements that need to be compared
- The gap decreases and ends as 1, where only the neighboring elements are compared exactly like the insertion sort

Example - Sort the elements as shown in the Table below using Shell sort.

Index	1	2	3	4	5	6	7	8
Elements to be sorted	18	3	6	9	11	14	16	2

Index:	1	2	3	4	5	6	7	8
Elements:	18	3	6	9	11	14	16	2



Index	1	2	3	4	5	6	7	8
Elements to be sorted	11	3	6	2	18	14	16	9

Example Contd.

Index:	1	2	3	4	5	6	7	8
Elements:	11	3	6	2	18	14	16	9

11 6 18 16

3 2 14 9

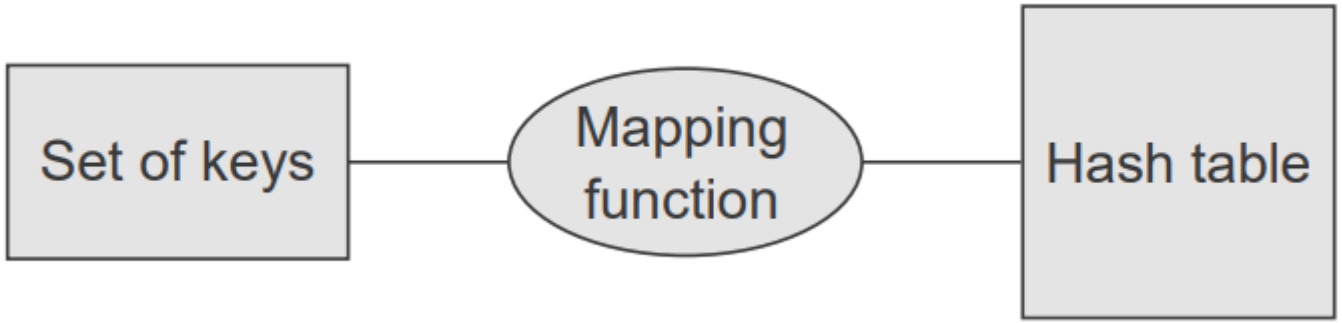
Index	1	2	3	4	5	6	7	8
Elements to be sorted	6	2	11	3	16	9	18	14

Index:	1	2	3	4	5	6	7	8
Elements:	2	3	6	9	11	14	16	18

Hashing and Hash Tables

- Hashing is a technique used for storing and retrieving keys rapidly
- It is a concept of distributing keys in an array called a hash table using a predetermined function called a hash function
- Let us first illustrate the concept of hashing using a realworld scenario. Suppose that a teacher in a classroom needs to call a student directly
- This step is possible as each student has a unique roll number. The student’s name may not be unique, as many students may have the same name

- Therefore, a roll number is a better choice, and accessing a student based on the roll number is an example of direct access
- Thus, the idea of hashing is based on direct addressing or accessing, which states that the key k can be stored at the kth location of an array, and one can retrieve the key by looking at the kth location



Introduction to Hash Function

- A hash function is a mapping function that maps a key to a value called hash value or simply hash
- After this mapping process, the hash value represents the key
- Thus, hashing is useful for creating indexes for locating keys in a larger database.
- A hash code, using a hash (or mapping) function, can mathematically be calculated as follows:

$$h(k) = k \bmod M$$

- Here,
  - k is the given key, h(.) is the hash function
  - M is the hash table size

- Using this hash code, data are stored in the hash table.
- The hashing process involves deriving and mapping a hash code to a hash table
- For example,
  - let us assume, for simplicity, that a hash function is given as hash(k) = hash(key) mod 10.
  - If the keys are 11, 22, and 34, then the following hash codes are obtained:
    - hash(11) = 11 mod 10 = 1
    - hash(22) = 22 mod 10 = 2
    - hash(34) = 34 mod 10 = 4

Quality of Hash Function

Box 10.2 Quality of Hash Functions

The quality of hash functions determines the quality of a hash table. Some of the important properties of hash functions are as follows:

1. Hash functions should be easy to compute.
2. They should be fast to compute hash codes faster.
3. Their functions should be stable. Stability means that the output of hash functions always remains the same for a given input key. In other words, a hash function cannot be a random function.
4. The output of a hash function should be distributed evenly throughout the entire range of a hash table.

Quality of hash Function Construction

Box 10.3 Quality of Hash Function Construction

Two parameters characterize hash tables:

Capacity This is the maximum number of entries that a hash table holds.

Fill factor This is the percentage of items filled by a hash function.

These factors influence the efficiency of hash table algorithms, expressed in terms of the load factor. The load factor is the ratio of the number of elements to the table size. This fact is given as follows:

Load factor (α) = number of entries/table size = n/M
(where number of entries = n)

If the numbers of entries (n) are eight and the total capacity of the hash table (M) is 10, then the load factor is given as 0.8. If the load factor reaches a critical value, say 80%, then the hash table has to be expanded by creating additional entries. Hash functions are one of the most important components of the hashing process.

Division Method

- The simplest way to construct a hash function is to divide a non-negative key by M and consider the reminder a hash code
- This method is called the division method and can be defined mathematically as follows:

$$\text{hash}(k) = k \bmod M \text{ or } \text{hash}(k) = k \bmod M + 1$$

- Normally, the value of M (i.e., the size of a hash table) is chosen as a prime number
- The following numerical example illustrates the computation of a hash function.

Mid-Square Method

- As per this method, the given key k is squared and then the hash address is determined by chopping an equal number of digits from both sides of the hash function
- For example, consider the key 127
  - As per this method, the square of 127 is taken:

    hash(127) = 16129
  - By chopping two digits from both sides, one gets 1 as the hash code

Folding Method

- As per this technique, the key k is split into many subparts $k_1, k_2 \ldots k_n$
- The splitting is such that all the split keys have the same number of digits
- Then all subparts are added, and all the carries of the leading digits are ignored during the process
- For example, consider a key 1274
  - As per this technique, let us first split the key into two subparts of two digits each

    hash(127) = 16129
  - Similarly, let us consider key 4684. This step leads to the following hash code:

    hash(4684) = 46 + 84 = 130 = 30 (as the leading carry is ignored)

Digit Extraction Method

- As per this technique, a predefined digit of the given number is considered the hash address
- Let us consider a given number of 14,568
- Extracting the digits at 1, 3, and 5 positions leads to a hash code 158.

Rotating Hash Method

- As per this method, any hash method (such as the division or mid-square method) can generate a hash code
- Then, rotation is performed by shifting the digits to get a new hash address
- For example, a hash address 20021 can be rotated as 12002 by folding the digits
- Once hash functions are used to generate the code, the keys are stored in a single-dimensional array called a hash table
- A given key k should be retrieved by looking at the kth location
- it is impossible to achieve this in practice, as a hash function may map two keys to the same location

Hash Table Operation

Hash table operation	Description
initialize()	Initialize a hash table
insertkey(*H*, *k*)	Insert a given key in the hash table
searchkey(*H*, *k*)	Search a target key
deletekey(*H*, *k*)	Delete a given key if it is available in the hash table
display(*H*)	Display the contents of the hash table

Algorithm Initialize

Algorithm initialize(H)

```
%% Input: Hash table H
%% Output: Hash table H with null values
Begin
    for all the locations of the H
      Set contents of H as null
    End for
End
```

Insert Key

Algorithm insertkey(H, k)

```
%% Input: Hash table H, key k
%% Output: Hash table H with a key inserted
Begin
  read(key)
  location = hash(k)
  case(location)
    = Null: 'No memory available' Exit
    Otherwise: location = key              %% Store key
  End case
End
```

Search Key

Algorithm searchkey(H, k)

```
%% Input: Hash table H, key k
%% Output: Hash table H with the key inserted
Begin
  read(key)
  location = hash(key)
  case(location)
      = Null: ‘Key is not available’ Exit
      Otherwise: if (k == location then return(location)) End if exit
  End case
End
```

Delete Key

Algorithm deletekey(H, k)

```
%% Input: Hash table H, key
%% Output: Hash table H with a key inserted
Begin
  read(key)
  location = hash(key)
  case(location)
      = Null: 'Key is not available' Exit
      Otherwise: if (key == location) then
                 location = -1
                 End if
                 exit
  End case
End
```

Display

Algorithm display(H)

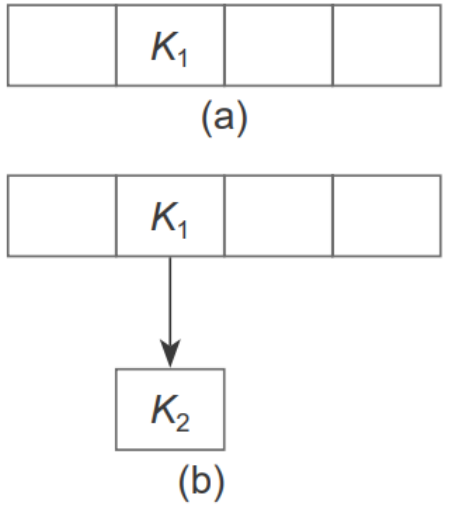
```
%% Input: Hash table H
%% Output: Contents of Hash Table H
Begin
  for all the locations of Table H
    write locations, item
  End for
End
```

Collision Resolution Techniques

- A collision happens when a hash function assigns the same hash code to two or more different keys.
- Broadly, the collision-resolving schemes are of the following two types:
  - Open hashing (separate chaining)
  - Closed hashing (open addressing)

Open Hashing

- Open hashing is a technique of resolving collision by creating a chain using a singly linked list
- This technique is also known by other names such as closed addressing or separate chaining method
- Let us assume that two keys
- k1 and k2 cause collision
- k1 being the first key occupies the first slot, as shown in Fig
- If the hash k1 function generates the same hash value for key k2, then a link is created between the keys k1 and k2, as shown in Fig



Open Hashing Contd.

- It can be observed that this technique does not create any extra k1 addresses in the hash table
- Instead, the method chains the keys that cause the collision. Here, the items are chained using a linked list
- In this case, a linked list is also called a chain. Therefore, this process k2 is called separate chaining

H(*k*)	0	1	2	3	4	5	6	7	8	9	10
8									8		
14				14							
27						27					
36											

↓

36

The efficiency of open hashing

- In open hashing, insertion, deletion, and searching operations for a single element requires O(1) time
- If there are n keys, hash operations require at most O(n) time
- Therefore, the worst-case complexity analysis of hash table operations would be O(n)
- The efficiency of a hash table is defined as the number of attempts made to locate a given key
- If S(α) is the average number of probes for a successful search and U(α) that for an unsuccessful search, then the efficiency of chaining is given as follows:

$$S(\alpha) = 1 + 1 \text{ and } U(\alpha) = e^{-\alpha} + \alpha$$

- Here, α is the load factor. The load factor $\alpha \geq 0$ in chaining as the number of entries is not constrained, unlike in open addressing where the number of entries cannot exceed the table size
- Generally, α ranges from 0 to 1 for open addressing, and $\alpha \geq 0$ for chaining

Closed Hashing(or open Hashing)

- Closed hashing is a technique used to store all the elements in the hash table without creating extra links
- The idea is to store a key in a free hash cell
- If the cell is occupied, then a new slot is generated
- If that is also occupied, then a new slot is generated again
- This process is repeated till the key is accommodated
- This kind of process that is used to compute alternative addresses is called probing

Closed Hashing(or open Hashing) Contd.

- Probing is the technique of resolving collisions, as it helps handle empty slots
- For example, linear probing probes the next element in a sequential manner
- Sometimes, a cluster is formed when there is a concentration of keys around the primary hash location
- This results in reduced performance, as further hashing creates more clusters
- This cluster is called the primary cluster

Closed Hashing(or open Hashing) Contd.

- The following are some of the techniques used for solving the problem of primary clusters:
  - Linear probing
  - Quadratic probing
  - Double hashing

Types of probing and hashing

- **Linear probing**
  - Linear probing is a technique where the interval between successive probes is usually 1
  - This technique allows a key to the next available location
- **Quadratic probing**
  - Unlike linear probing, quadratic probing does not inspect hash tables individually
  - Instead, it inspects the hash table as a square of its probing
  - Mathematically, quadratic hashing is given as follows:

    $$h(k) = (k + i\ 2\ ) \bmod M, \text{ where } i = 1, 2, 3, \ldots$$

  - Here i is the probing that is made, and M is the hash table size, which is chosen to be a prime number
  - Thus, probing would be performed at locations 1, 4, 9, 16, etc
  - The computation complexity of quadratic probing is O(1) for successful and unsuccessful searches.
  - Quadratic probing is better than linear probing, as only M/2 comparisons are required. Thus, the complexity for n elements can be given as O(n).

Types of probing and hashing Contd

- Double hashing
  - Quadratic probing results in a secondary cluster
  - A secondary cluster refers to a scenario where two keys are mapped by a hash function to the same location, and its probing sequence also remains the same
  - Much space is wasted in quadratic probing
  - In addition, there is no guarantee that all the hash table slots will be visited
  - Double hashing can be used instead of quadratic probing to overcome these disadvantages
  - In short, if the first hash function is unsuccessful, a second rehash function is used
  - Double hashing can be expressed as follows:

$$h(k) = (h1(k) + i \times h2(k)) \bmod M$$

  - It can be observed that two hash functions have been used in this expression.

M-Ary Trees

- In database applications, one deal with a large amount of data
- Hence, a BST can be called a two-way tree
- One can generalize this finding to an M-way tree, where M is the number of children that a node can have
- M is also known as the branching factor.
- M-way trees are also known as k-ary trees, where k indicates the number of children a node can have
- Thus, an M-way search tree is a data structure that has the following properties:
  - Each node of an M-way tree has M-1 values or keys per note
  - Each node of an M-way tree has M subtrees.
- This approach is extremely useful in large applications storing data as nodes of an M-way tree.

M-Ary Trees Contd.

Box 10.4 Need for Indexing

A typical database application may have billions of records, but storing them in the primary memory is difficult. Hence, they are stored in the secondary memory; a record is moved from the secondary to the primary memory whenever required. As secondary storage involves the usage of slow magnetic tapes and relatively faster magnetic disks, a bottleneck is created due to slow disk access. In such cases, B-trees are useful, as they help reduce the number of disk accesses.

For example, in magnetic tapes and disks, data are stored in blocks, and a block may have many records. Every block corresponds to an M-way tree's interior and exterior nodes, and each node may have many keys. When data are stored on a disk, the number of accesses increases. Therefore, the objective is always to reduce the number of accesses. Increasing the *M* value of an *M*-way tree As a result, the number of disk accesses is reduced, decreasing access time. If the value of *M* is increased, the *M*-way tree would be imbalanced, as discussed in Chapter 6. A B-tree (or a balanced search tree) can solve this problem.

B-Tree

- A B-tree is a data structure that facilitates faster access to data stored in the secondary memory by creating and using indexes to access a huge amount of data
- This kind of indexing is similar to using indices in a book to access related content rapidly
- In addition, a B-tree is suitable for creating a dictionary by storing keys and records, so that one can access the corresponding record by providing a key
- The important jargon of B-tree is shown in Box below

B-Tree Contd.

Box 10.5 Jargon Related to B-Trees

Hierarchy: A node can have multiple keys or elements. A node can have a parent and children. Collectively, this structure gives the hierarchy of nodes. A node can be a leaf node (Node with no children), an Internal node (Node with children) or a special node called a root node.

Leaf node: The nodes in the last level of the B-tree do not have any children.

Order: The number of keys or elements a node can have is called order.

Order of the Tree: The number of maximum children that a B-tree can have is called the order of the tree.

All B-tree leaves are at the same depth as the tree. This constraint leads to a balanced M-way search tree. This property is known as depth property. This property implies that all the leaves of a B-tree have the same depth. A B-tree is always balanced; only its height grows during an insertion operation.

The best-case height of the B-tree is given as

$$\log_m(n+1), \text{ where } n > 0 \text{ is the entries in the tree.}$$

The worst-case height is given as

$$h \leq \log_d\left(\frac{n+1}{2}\right), \text{ where } d = m/2$$

The maximum number of elements in the B-tree is given as $m^{h+1} - 1$. If the order of the B-tree is 4 and the height is 2, then the B-tree can have $4^{2+1} - 1 = 63$ elements and be accessed in 2 disk reads.

Properties of B-tree

- A B-tree of order m, where m is the maximum number of children, whose node should satisfy the following properties:
  - The root node should not be empty or leaf node. In that case, The root has at least two children
  - Besides the root and leaf, every node can have the children, maximum m and minimum n/2
  - All nodes, other than the root node, have maximum m-1 and minimum m/2 keys
  - All leaves should have the same level.
  - A set of keys that are in sorted order. Thus, an internal node of a B-tree is an ordered set of elements and a set of child pointers.
  - A flag to indicate whether a node is a leaf or not. It is true if a node is a leaf
  - The key separates the range of keys stored in the child node of that node.

Variants of the B-tree

Box 10.6 **2–3 Trees and 2–3–4 Trees**

A 2–3 Tree is a balanced tree, a B-tree when the order is 2. In 2–3 trees, every non-leaf node has 2–3 children. All leaves are at the same level or depth. All data is stored in leaf nodes, and the path length from the root to the leaf is the same. All data are ordered from left to right. The following Fig. 10.10 shows the example of 2–3 trees.

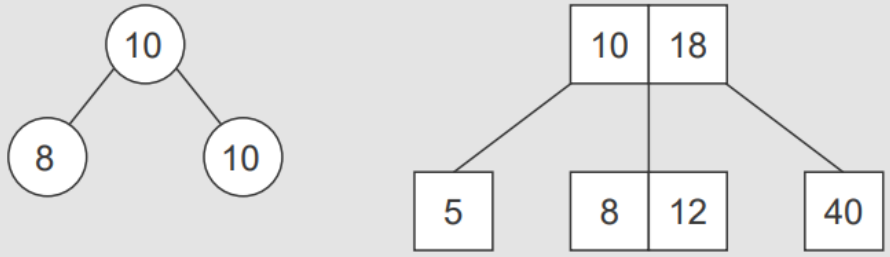


Fig. 10.10 2–3 Trees

Let us consider the sample B-tree shown in Fig. 10.11. Fig. 10.11 shows a B-tree where the internal node can have two, three, or four children. This kind of tree is also called a 2–3–4 tree.

Like a BST, it can also be observed here that the values stored in the left subtree are less than the value of the root, and those of the right subtree are greater than the value of the root. The middle subtree

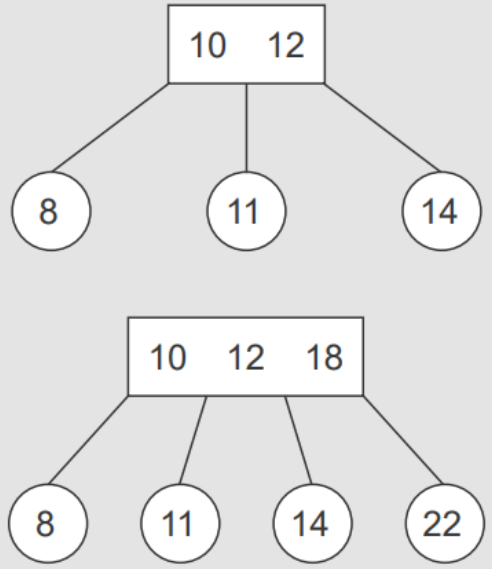


Fig. 10.11 2–3–4 Trees

points to the middle values. In addition, one needs to examine more than one node element to find the target key.

2–3 trees and 2–3–4 trees are more like AVL trees, where the tree is balanced but not useful for storing more elements; hence, B-tree is more useful.

B-tree Operations

B-tree operation	Description
searchkey(*x*, key)	For a given B-tree B, the operation is to find the target key if present; else, a status message should be returned as false.
insertkey(*x*, key)	For a given B-tree, the operation is to insert a new key with its value.
Delete a key	For a given B-tree, the operation is to delete a given tree.

Searching For Target Key

Algorithm search(x, key)

```
%% Input: x is the root node of a B-tree, and the key is the value
%% that is sought
%% Output: Returns the value of the key if present in the B-tree
Begin
  for all nodes of B, do
      if  key is in x  then
          return(key)
      else if (x is leaf) then
          return(false)
      else
          return(search(child(x), key))
      End if
      End if
  End for
End
```

Searching For Target Key

- The informal algorithm for inserting an item is given below:
  - Step 1: Read the element to be inserted and traverse the B-tree to check whether it is already present
  - Step 2: If the key is absent in the B-tree, find an appropriate location to insert the element using the search process
    - 2a: If the node has m–1 keys, the item can be inserted, and the process is over.
      - Otherwise, If the node already has m–1 keys (In that case, the node is already full, one cannot insert the element, as it violates the rules of a B-tree
      - This step leads to an error condition called an overflow). In that split has to be done. So, insert the key in the increasing order
      - Split the node by pushing the median up
      - If the parent node too cannot accommodate the promoted value, then split the parent node. Continue the insertion process till the given key is accommodated Step 3: Return the B-tree with the inserted node if the insertion is successful.

Insertion of terms into B-Tree

Box 10.7 Overflow and Split

Overflow can happen when the node already has the maximum number of keys allowed. When a new element is inserted in that node, the node has to be split.

The split is a balancing operation normally encountered in an insertion operation of B-trees and ensures that the B-tree conditions are not violated. The node is split when it has $2m–1$ keys into two nodes of $m–1$ keys. The middle key is to be moved to the parent node. If the parent node is also full, the parent node will encounter overflow. The split process is continued until the item is accommodated.

Example-Insert the following elements into the B-tree of order 5-10, 20, 15, 30, 60, 80, 90, 100.

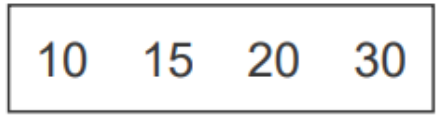


Fig 10.12 Initial Elements of B-tree

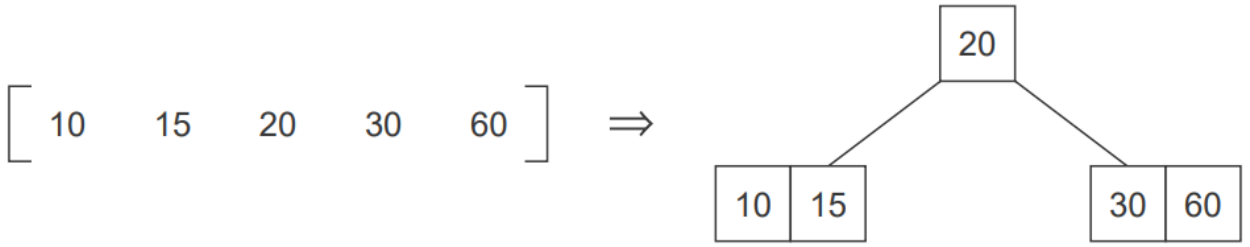


Fig 10.13 After the insertion of Element 60

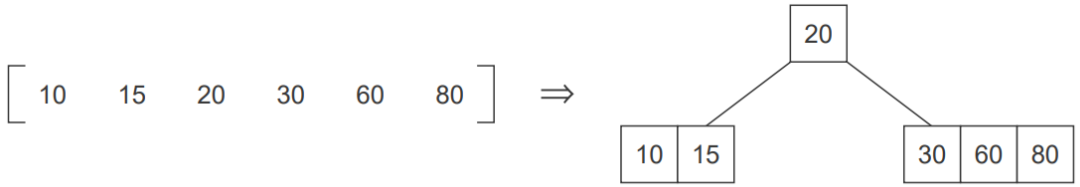


Fig 10.14 After the insertion of Element 80

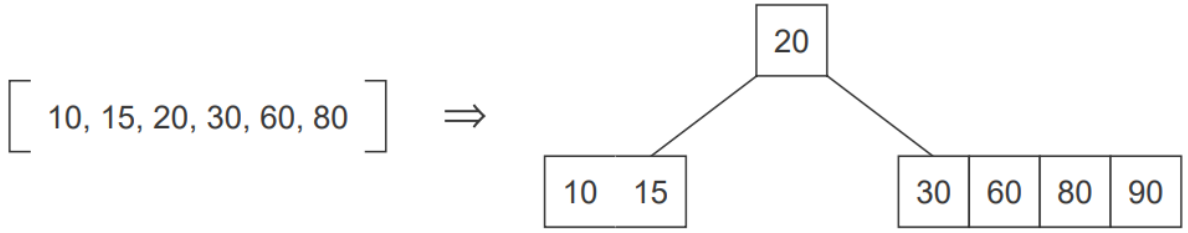


Fig 10.15 After the insertion of Element 90

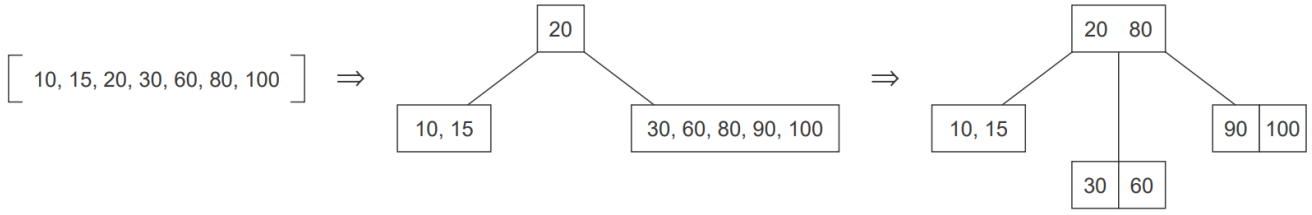


Fig 10.16 After the insertion of Element 100

Insert key Algorithm

Algorithm insertkey(x, key)

```
%% Input: Root node x of B-tree and key
%% Output: B-tree with inserted key
Begin
    if (isfull(x)) then
        split(x)
        search(x, key)
    end if
    case(x)
         x is leaf: place key in x
         otherwise: z = findappropriatechild(Ci, x)
             insertkey(z, key)
    End case
End
```

Algorithm for is full

Algorithm isfull(x)

```
%% Input: Root node x of B-tree
%% Output: Status message
Begin
    check the count of keys in x
    if (count = m/2) then
       return(true)
    End if
End
```

Deletion of item from B-Tree

- Step 1: Check if the item to be deleted is present. If a key or element is not present, the process ends here
- Step 2:
  - Case 1: Key is present in the leaf node: If the key is present in the leaf node, then check if the node has more than the required keys (m/2)
  - If so, then delete the key
  - The process is over.
  - Case 2: If a key is in the internal node: Then,
    - Check the left child of the node where the key is present
    - If the node has more than the minimum number of keys (m/2), find the predecessor (max key) in the left subtree, copy it to the parent node, and move the intervening element down to the node, where the key will be deleted
    - Recursively delete the maximum key in the left subtree.
    - If the left subtree has fewer keys, then check the right tree
    - If the right tree has more than the minimum number of keys (m/2), find the successor (minimum key) and copy it to the parent node
    - Move the intervening node down to where the key will be deleted
    - Recursively delete the successor from the right subtree
  - Case 3: If the node has neither left nor right, and the child node has more than (m/2) elements or keys, then create a new node y merging the two leaf nodes and the intervening node
- Step 4: Repeat the above process if the parent node has less than (m/2) nodes and underflow is encountered

Underflow and Merging

Box 10.8 Underflow and Merging

Controlling the underflow is the key to the efficiency of the B-tree. The condition of underflow happens when the node has only the minimum required. Hence, removing a key violates the principle of B-tree.

The following two operations are required for handling underflow:

1. **Borrow and Rotation**

Borrowing is a simple B-tree balancing operation, where a B-tree node can borrow a key from a sibling if the size property of the B-tree is not violated. For example, consider the sample B-tree shown in Fig. 10.17.

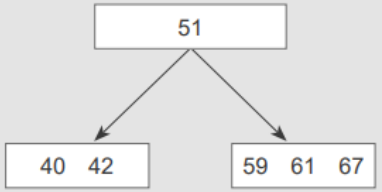


Fig. 10.17 Sample B-tree

As a B-tree balance operation allows borrowing of a key from a sibling node, let us assume that the node {40, 42} borrows a key from a sibling {59, 61, 67}; then the resulting B-tree would be as shown in Fig. 10.18. This resulting B-tree is not correct, as key 59 > 51. Therefore, one must rotate to get a correct B-tree, as shown in Fig. 10.19.

Thus, rotation is an operation that is performed to ensure that the properties of a B-tree are not violated. It can be observed that a rotation operation moves down the parent key value to the left child and copies the first key of the right child to the parent node. Sometimes, borrowing and rotation may not solve the problem. In that case, a merging operation is required.

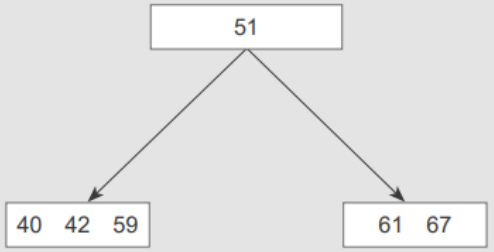


Fig. 10.18 Resulting Invalid B-tree

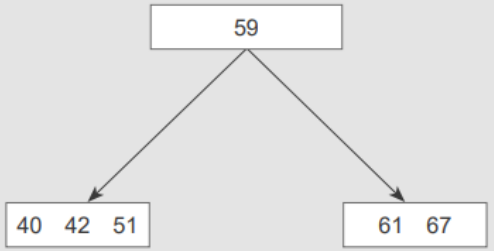


Fig. 10.19 B-tree after Rotation

2. **Merge Operation**

Merging is another balancing operation used to handle an underflow condition and is encountered in the deletion operation of a B-tree. In rotation, the node that encounters the underflow condition borrows a key from its left or right node. Then the key from the parent node is brought to the merged node. If the parent node has an underflow condition, it is merged with its siblings. If the parent node is also a root, then it is replaced by its only child.

Complexity Analysis

- Search Algorithm : $O(\log_T n)$
- TotaL Search Algorithm: $O(\log n)$
- Insertion and Deletionalgorithm: $O(\log n)$

B+ Trees

- B+ tree is an extension of B-trees. The properties of the node in the B+ tree are given below:
  - Each node in the B+ tree, except the root and leaf, can have m-children and a minimum of m/2 children
  - Each node, except the root node, can have a maximum of m–1 minimum as (m/2) keys
  - All the leaf nodes are at the same level
  - All keys are in sorted order
  - All the internal nodes / non-leaf nodes contain only key and not data
  - Data is present only at the leaf level
  - Doubly linked lists link all data.

Difference between B+ and B- Tree

B–Tree	B+ Tree
Data is present in all nodes.	Data is present only in the leaf node.
Leaf node data is not linked.	All data are linked by a doubly linked list, ensuring sequential access.
Not suitable for a full scan, and it results in the cache miss.	Suitable for a full scan.
Data is closer to the root.	All data is in leaf nodes only.
Takes little memory	More storage is required, as keys are stored in many places.