Part A

1. An overflow block in an index sequential file
   a. increase retrieval time
   b. decrease retrieval time
   c. make insertion easier
   d. both b and c
   e. both a and c

2. An index in which there is one index entry for each record is called a
   a. Spanned index b. Sparse index c. Dense index d. None of these

3. An example for a single level ordered index
   a. Secondary index b. Clustered index c.Grid file d. None of these

4. Average time complexity of an normal interpolation search is
   a. a.O(n) b.$O(\log_2 n)$ c. $O(\log_2 (\log_2 n))$ d.$O(\log_2 n)^2$

5. Response time depends upon
   a. access time and data transfer time
   b. seek time and data transfer time
   c. latency time
   d. latency time and seek time

6. The size of a record-list is always one for
   a. Inverted index file b. Multi list file c. Cellular list d. none of the above

7. Which among the following is not an indexing method used in Index Sequential file
   a. Implicit index b. Limit index c. Inverted index d. both a and b

8. Time taken by a read write head to move from one track to another is
   a. Seek time b. Latency time c. Data transfer time d. none of the above

9. Data blocks on magnetic tape are separated by
   a. EOF b. IBG c. Track d. Sector

10. Seek time of a fixed head drive is
    a. greater than movable disk drive
    b. less than movable disk drive
    c. zero
    d. none of the above

11. Access time depends on
    a. access time and data transfer time
    b. seek time and data transfer time
    c. latency time

   d. latency time and seek time

12. An attribute which uniquely identifies a record is called
    a. Primary key b. Candidate key c. both a and b d. none of the above

13. In sequential files deletion is performed by
    a. Deletion marker b. Shifting c. both a and b d. none of the above

14. In a multilevel indexing scheme maximum number of index that can accommodate in a single index table is 4. Given 22 records. How many level of indirection is required.
    a. 1 b. 2 c. 3 d. none of the above

15. Example for a dynamic hashing technique
    a. Mid square b. Extendable hashing c. divide by prime d. both a and b

16. Updates to be made on a sequential file is stored in a
    a. Auxiliary file b. Master file c. Transaction file d. none of the above

17. A Bucket Address Table contains
    a. key, bucket address
    b. length of key, bucket address
    c. length of the key, key, bucket address
    d. none of the above

18. A DONTAG contains
    a. access_record, attribute, value, next_record
    b. access_record, current_attribute, next_attribute, next_record
    c. attribute, value, next_record
    d. none of the above

19. In Chaining method, the records falling under the same bucket are
    a. stored as arrays
    b. connected by pointers
    c. stored as B-trees
    d. none of the above

20. Which of the following is not a static hashing technique.
    a. Division b. Extendable hashing c. Folding d. Digit Analysis

21. The balanced factor of a AVL tree can be
    a. –1,0,+1 b. –2, –1,0,+1,+2 c. –2,0,+2 d. None of these

22. Key value appear only once in the case of a
    a.B-tree b. B^+ tree c. both a and b d. None of these

23. In a threaded binary tree, a right thread point to

a. inorder predecessor
b. inorder successor
c. right sibling
d. None of these

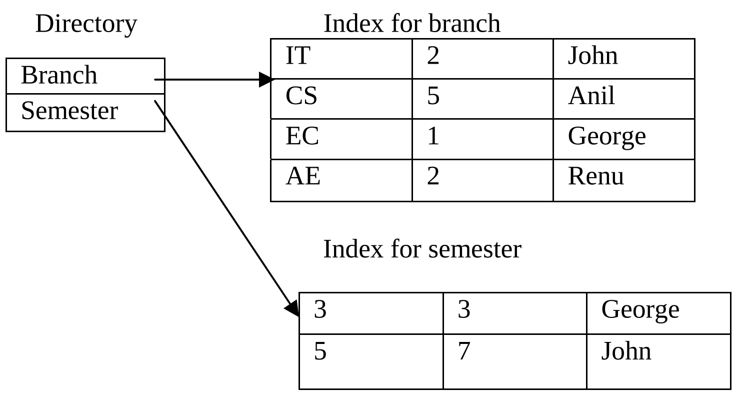
Part B

24. Explain direct files.
25. The directory structure of an index sequential file is as given below

Key	**Page#**
25	3
26	43
28	56
30	103

Create the implicit index for the above file.

26. Explain how to perform insertion in Index Sequential File.
27. The directory structure of a multi list file is as given below



Directory

Branch
Semester

Index for branch

IT	2	John
CS	5	Anil
EC	1	George
AE	2	Renu

Index for semester

3	3	George
5	7	John

Explain the selection steps for performing the following queries
i. Select all students with branch=”IT” and Semester = 5
ii. Select all students with branch=”IT” or Semester = 5

28. Write down the recursive algorithm for binary search. Illustrate with an example
29. Illustrate extendable hashing with suitable example.
30. Consider the numbers 9,11,13,16,65,70,71. Which search method is best for these numbers. Justify your answer. (Assume the numbers are stored in the sorted order)
31. Explain various single level order indexes.
32. Suppose that you have an ordered file with 20,000 records stored in a disk with block size of 1024 bytes. All the records are of equal size and are unspanned with record length equal to 100 bytes. A primary index is created and each index entry is of 15 bytes. How many block access is required to locate a record.
(Assume that you are using binary search technique)
33. Write short note on static hashing.
34. Explain chaining.
35. Given a hash function h(x) = first character of x. There are 26 buckets with 2 slots per bucket. The elements to be inserted be the names of students in a class. Let the names be Tom, Zen, Bob, Alice, Zion, Ziya, Arun, Karl, Benny, Kane.

Draw the hash table after inserting the identifiers. It is given that the system uses linear probing method to handle overflow.
(Assume that the elements are inserted in the same order specified above.)

36. Compare and contrast B-trees and B^+ tree.
37. What are the advantages of using dynamic hashing method.
38. Create a height balanced tree with elements as the months of a year. Show necessary steps.
(Assume that the elements are inserted in the order Jan, Feb, …)
39. Write short note on Sequential files.
40. Explain Folding.
41. What is a B-tree.
42. What is a Weight balanced tree
43. Write down first-fit storage allocation algorithm.
44. Explain Secondary key retrieval on files.
45. Write short notes on
    i. Single level ordered index
    ii. Multilevel index.
46. Discuss the various static hashing functions
47. Explain various collision resolution techniques
48. Write short notes on Threaded Binary Tree
49. Explain Buddy System
50. Write short notes on Garbage Collection and Compaction.

Answers

Part A

1. e. both a and c
2. c. Dense index
3. b. Clustered index
4. c. $O(\log_2 (\log_2 n))$
5. a. access time and data transfer time
6. a. Inverted index file
7. c. Inverted index
8. a. Seek time
9. b. IBG
10. c. zero
11. d. latency time and seek time
12. c. both a and b
13. b. Shifting
14. c. 3
15. b. Extendable hashing
16. c. Transaction file
17. c. length of the key, key, bucket address
18. a. access_record, attribute, value, next_record

19. b. connected by pointers
20. b. Extendable hashing
21. b. –1,0,+1
22. a B-tree
23. b. inorder successor

Part B

24. Memory is divided into bucket and buckets are again divided into slots. A slot can contain n records. A key value is mapped directly to a bucket by performing some arithmetic manipulation of the key by a hash function. We can find the record by searching the buckets.

25. For Implicit indexing, instead of writing both key and address, specify address only. If a key value doesn't exist specify that with any random value that may never come.

    Page#
    3
    43
    -2000 (Random value for key 27)
    56
    -2000 (Random value for key 29)
    103

26. Index sequential file consists of data plus one or more levels of indexes. When inserting a records, since we have to maintain the sequence of records, we have to shift the records. In order to avoid this task the records that overflow their logical area are shifted into a designated overflow area and pointer is provided in the logical area or associated index entry points to the overflow location. Multiple records belonging to same logical area are chained to maintain logical sequencing. This simplifies insertion but increases search time.

27. fgf

28.
```
int binsrch (int a[], int x, int low, int high)
{
        int mid;
        if (low > high)
                return(-1);
        mid = (low+high)/2;
        return (x = = a[mid] ? mid : x < a[mid] ? binsrch (a,x,low,mid-1) :
                                                  binsrch (a,x,mid+1,high))
}
```
    Working :

    Consider the numbers 2,5,8,9,11. Search for 5.

Pass 1:

binsrch(a,5,0,4)

mid = 2, compare between a[2] and 5, since a[2] is greater call binsrch(a,5,0,1)

Pass 2:

binsrch(a,5,0,1)

mid = 0, compare between a[0] and 5, since a[0] is smaller call binsrch(a,5,1,1)

Pass 3:

binsrch(a,5,0,1)

mid = 1, compare between a[1] and 5, it is equal hence return with mid as 1. Hence element is found at 2nd position.

29. Extendable hashing handles growth and shrinkage by splitting or adding bucking. Instead of using value produced by the hash function as bucket addresses, some variable number of bits from these value are used as key for entries in another level of indirection called Bucket Address Table (BAT). An entry in a BAT contains <length (of key), key, bucket address> . Length field varies as bucket becomes full and new buckets are created.
As an example consider a hashing system in which each bucket can hold almost 2 records.
Initially the file contains 3 records 000, 001 and 010. Hence the hash table will be as follows.

Length	**Key**	**Bucket**
1	0	B1
1	1	B2

The buckets B1 and B2 will be as below

B1	
0	P0
2	P1

B2	
1	P1

After adding records 010 and 100. Hash table will be

Length	**Key**	**Bucket**
2	00	B11
2	10	B12
1	1	B2

The buckets will have entries as below

B11	
0	P0
4	P4

B12	
2	P2

B2	
1	P1

30. The numbers are 9,11,13,16,65,70,71

Binary Search :

mid = (low+high)/2;

Interpolation Search :

mid = low + (high-low)*((key – k(low)) / (k(high)-k(low))

Robust Interpolation Search :

p = low + (high-low)*((key – k(low)) / (k(high)-k(low))
gap = sqrt(high-low+1)
mid = min (high – gap, max(p, low + gap))

As an example consider case when key = 9

Binary Search

Pass 1:
mid = (0+7)/2
= 3
Pass 2 :
mid = (0+2)/2
= 1
Pass 3 :
mid = (0+0)/2
= 0
No of search : 3

Interpolation Search

Pass 1 :
mid = 0 + (6-0)*((9 – 9)) / (71-9))
= 1
No of search : 1

Robust Interpolation Search

Pass 1:
p = 0 + (6-0)*((9 – 9)) / (71-9))
= 0
gap = sqrt(6-0+1)
= 2
mid = min(5,max(0,2))
= 2

Pass 2:
p = 0 + (2-0)*((9 – 9)) / (11-9))
 = 0
gap = sqrt(1-0+1)
 = 1
mid = min(0,max(0,1))
 = 0

No of search : 2

	No of searches required		
Key	**Binary Search**	**Interpolation Search**	**Robust Interpolation Search**
9	3	1	2
11	2	2	3
13	3	3	1
16	1	4	2
65	3	2	1
70	2	1	2
71	3	1	3

31.

- Primary index : Ordered file with two fields. Primary key and pointer to disk block. One index entry for each disk block. And hence it is a sparse index.
- Clustering index : Records of a file are physically ordered on a non key field which does not have a distinct value for each record. Each record in index has 2 fields. First field is same as clustering field in data file and next is a pointer to one entry for each distinct clustering field, a pointer to first block in data file that has a record with that value.
- Secondary index: Index created for a secondary key field.

32.

No of records	: 20000
Block size	: 1024
Record size	: 100
No of records/block	: 1024/100 = 10
No. of blocks needed	: 20000/10 = 2000
Size of 1 index	: 15
No. of index/block	: 1024/15 = 68
No. of index entries needed	: 2000
No. of index blocks required	: 2000/68 = 30
No of search needed	: $\log_2(30) + 1$

33.

- Identifiers are stored sequentially in fixed size table called hash table.

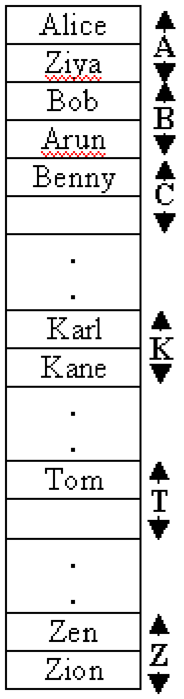
- The address of an identifier is obtained by computing some arithmetic function *f* on the key *x*
- The hash table is partitioned into buckets, each bucket consists of slots, each slot being large enough to hold a record.
- Loading factor of a hash table is given by
  l = n/(s*b) where
  n - no. of identifiers
  s – no. of slots
  b – no. of buckets

- E.g., for static hash function are Mid-Square, division, folding, digit analysis

34.

- Chaining is an overflow handling technique while using a static hashing method. An overflow is said to occur when a new identifier is hashed into a full bucket.
- Maintains a list of identifiers, one list per bucket organized as link list. Each list contains all the synonyms for that bucket.
- A search involves computing the hash function *f(x)* and examining the identifiers in the list.
- Each chain has a head node, usually smaller than other nodes (because it contains only the link field) and head nodes are arranged sequentially.

35.

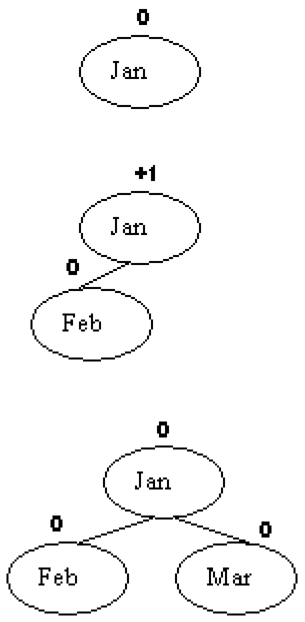


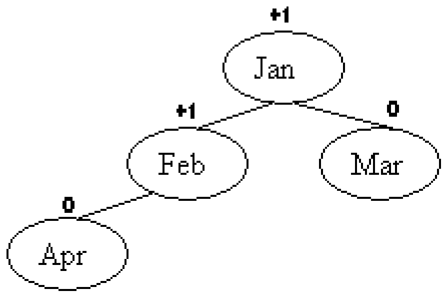
36.

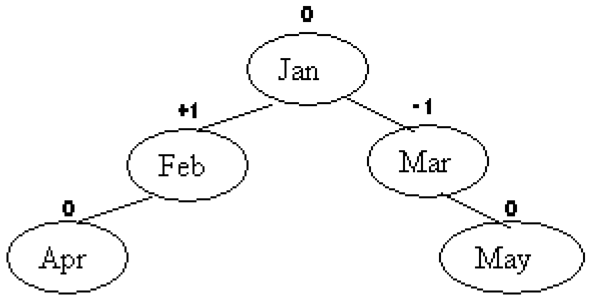
B- Trees	B^+ - Trees
Key value appear only once, and can be	Some of the key values may be

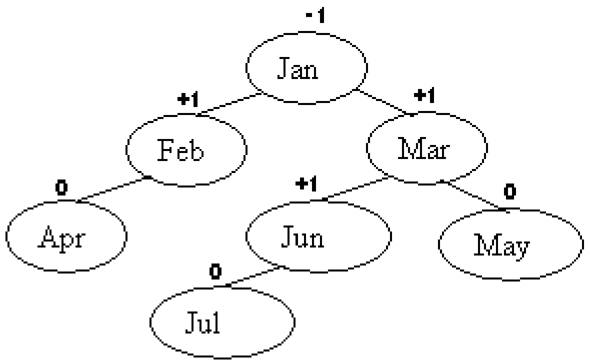
anywhere in the tree	repeating. All keys should be there in the last level
Insertion and deletion can be anywhere in the tree.	Insertion and deletion are done always at the leaf nodes
Less storage area required	More area compared to B -Trees
Eg.	Eg.

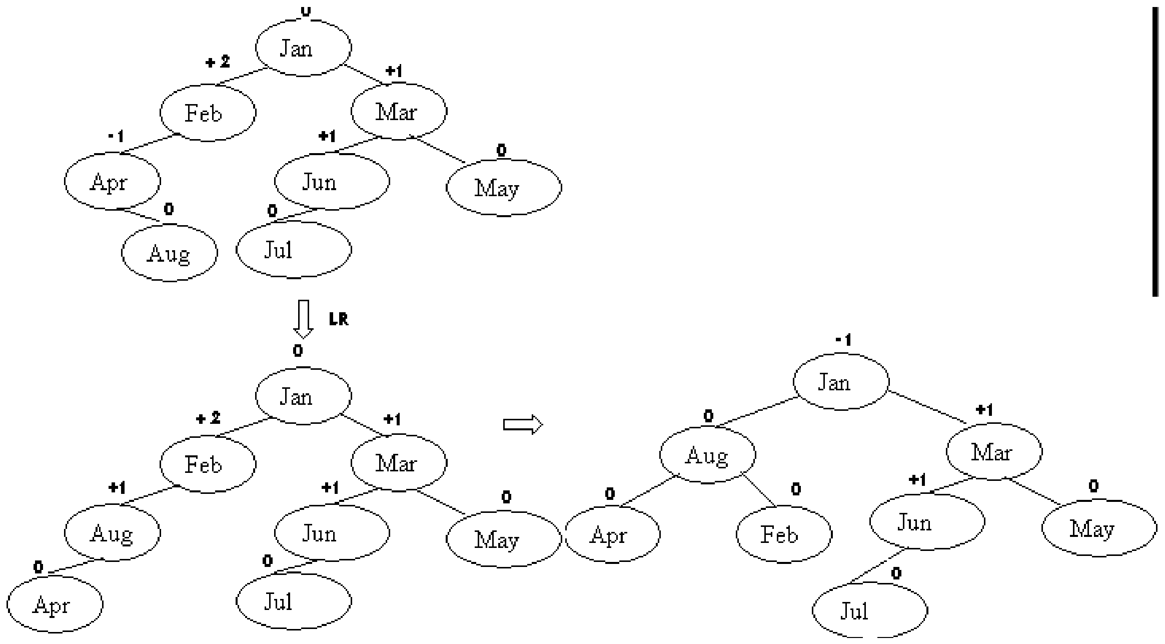
37. Static hash function statically allocate a portion of memory. If the memory allocated is too small, then when data exceeds capacity of hash table, the entire file must be reconstructed. And if too large, space is wasted. The purpose of dynamic hashing is to retain the fast retrieval time of conventional hashing while extending the technique so that if it can accommodate dynamically increasing and decreasing file size without penalty.
38. fgf

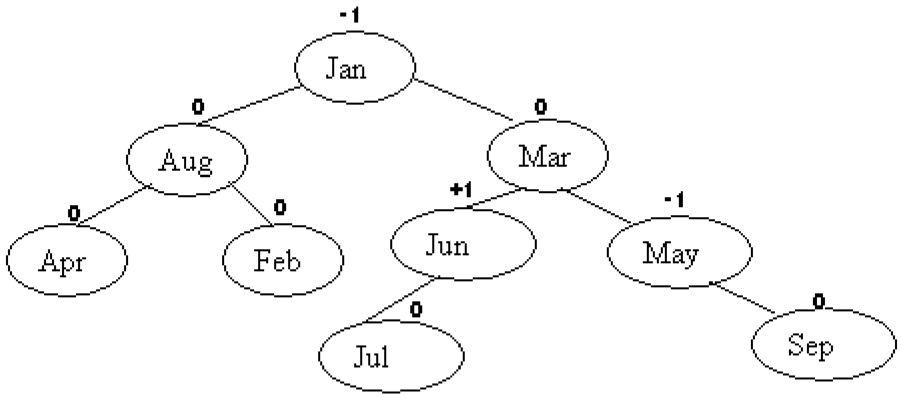


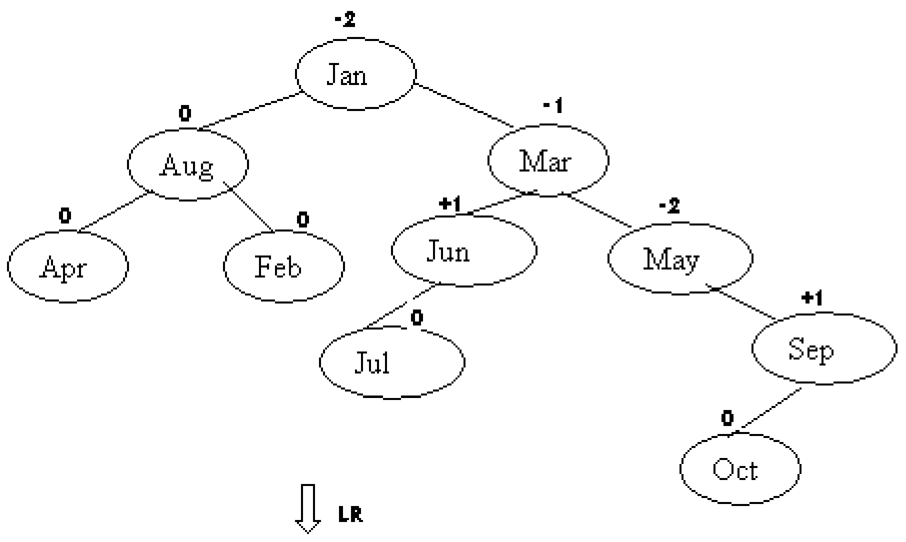

+1
Jan
+1
Feb
0
Mar
0
Apr

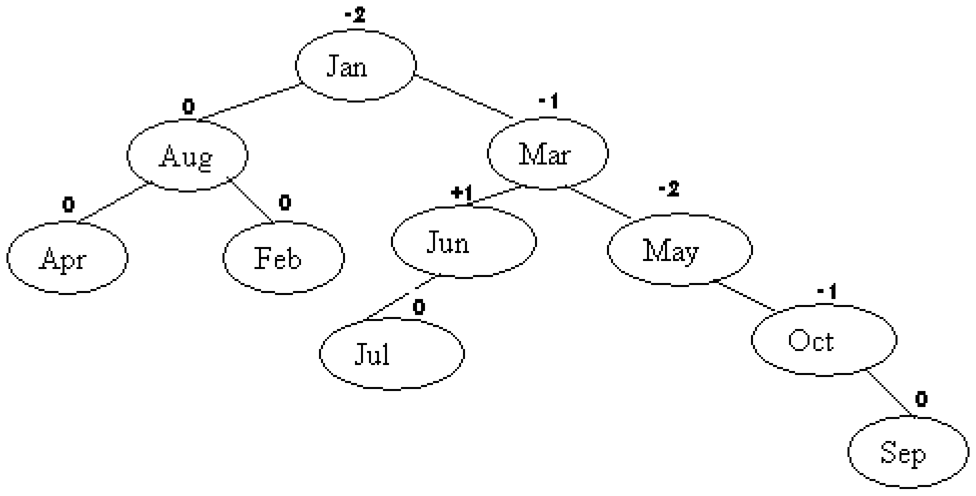

0
Jan
+1
Feb
-1
Mar
0
Apr
0
May

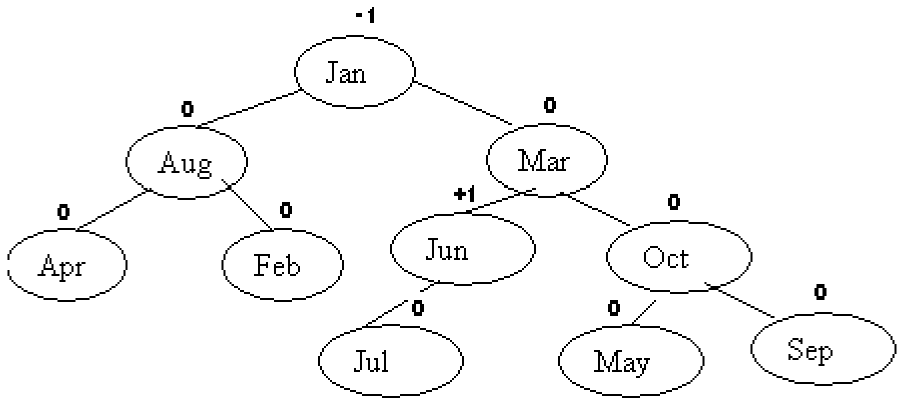

-1
Jan
+1
Feb
+1
Mar
0
Apr
+1
Jun
0
May
0
Jul

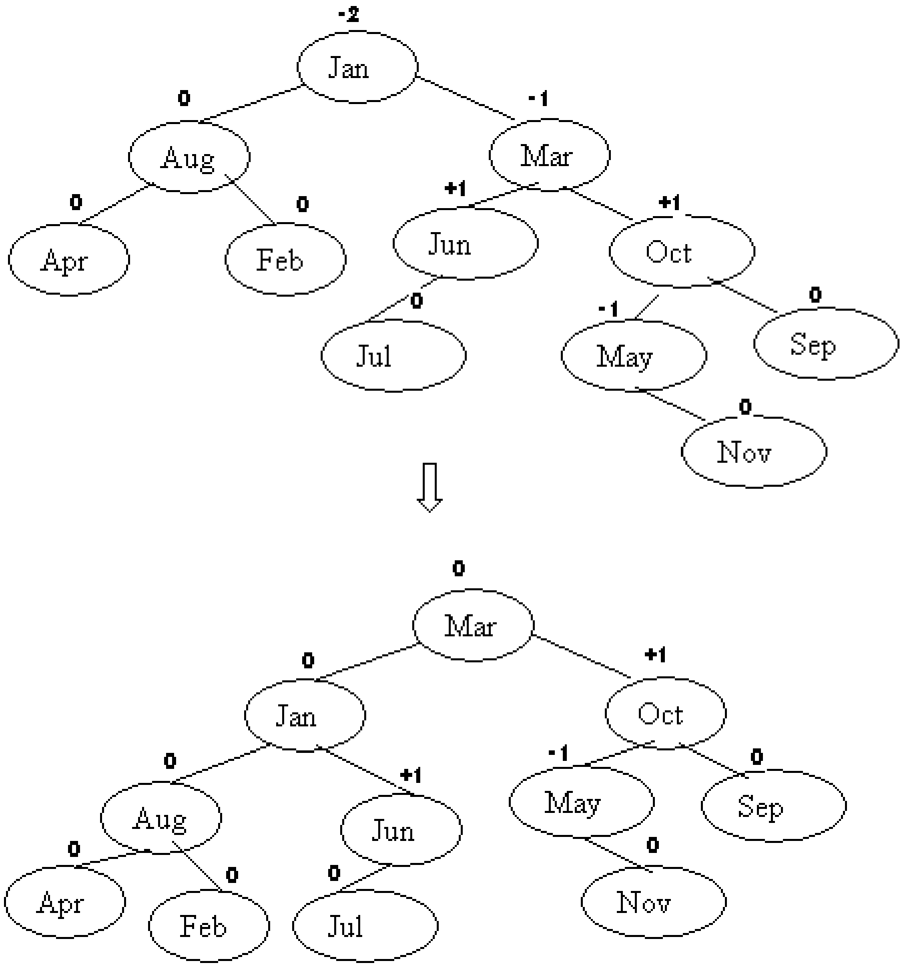

Jan
Feb
Mar
Apr
Jun
May
Aug
Jul
LR
Jan
Feb
Mar
Aug
Jun
May
Apr
Jul
Jan
Aug
Mar
Apr
Feb
Jun
May
Jul

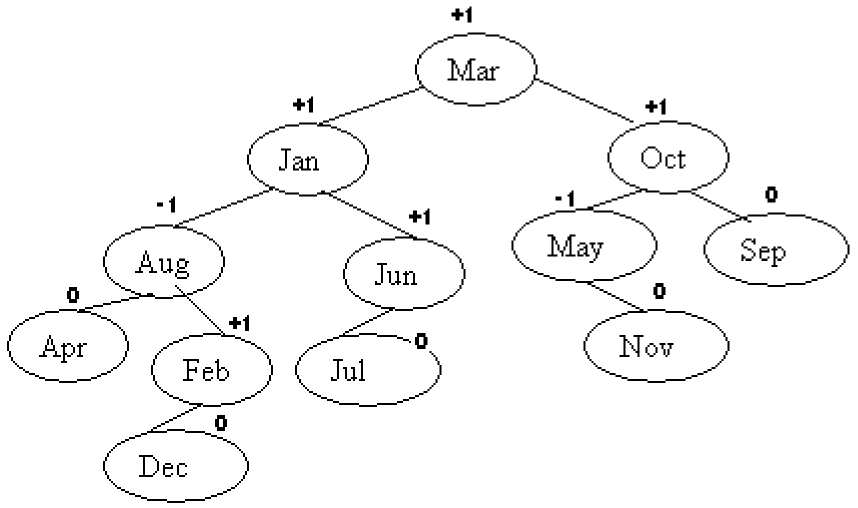

Jan
Aug
Mar
Apr
Feb
Jun
May
Jul
Sep


-2 Jan
0 Aug
-1 Mar
0 Apr
0 Feb
+1 Jun
-2 May
0 Jul
+1 Sep
0 Oct
LR


-2 Jan
0 Aug
-1 Mar
0 Apr
0 Feb
+1 Jun
-2 May
0 Jul
-1 Oct
0 Sep


-1 Jan
0 Aug
0 Mar
0 Apr
0 Feb
+1 Jun
0 Oct
0 Jul
0 May
0 Sep





39. In sequential file,
    - Records are maintained in the logical sequence of their primary key values. Search for a given record in this files requires, on average, access to half the records in the file.

- Adding a record necessitates shifting of all records from the appropriate point to the end of file to create space for the new record.
- Updating requires creation of a new file. Records are copied to the point where amendment is required. The changes are then made and copied into the new file. Following this, the remaining records are then made copied into the new file.
- To reduce the cost per update, all request are batched, sorted in the order of the sequential file, and then used to update the sequential file in a single pass. Such a file, containing the updates to be made to a sequential file, is called a transaction file.

40. Folding is an example of a uniform hash function (a function that does not result in a biased use of hash table for random inputs. i.e., probability that $h(x) = i$ to be $1/b$ for all buckets i) . In this method the identifier is partitioned into several parts, all but possibly the last being of the same length. These partitions are then added together to obtain the hash address for x. Folding itself is divided into two, depending on how we are performing the addition.

In the first method called *shift folding*, all but the last partition are shifted so that the least significant bit of each lines up with the corresponding bit of the last partition. The different partitions are added together to get $h(x)$.

e.g.

x = 123456789. The partitions are $P_1 = 123$, $P_2 = 456$, $P_3 = 789$

h(x) = 123 + 456 + 789 = 1368.

In the second method called folding at the boundaries, the identifier is folded at the partition boundaries, and digits falling into the same partition are added together to obtain $h(x)$. This is equivalent to reversing alternate partition and then adding.

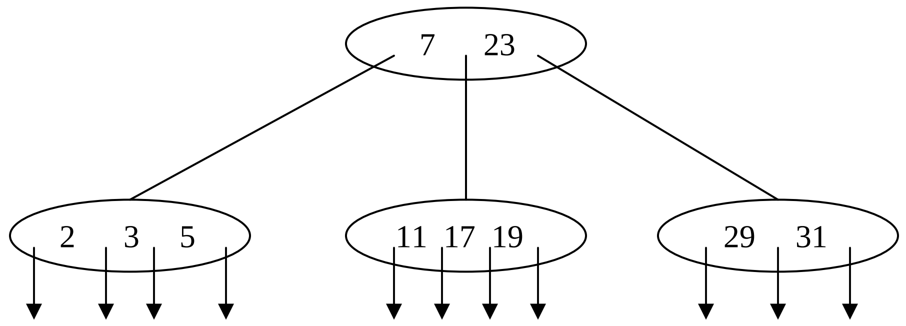
e.g.

x = 123456789. The partitions are $P_1 = 123$, $P_2 = 654$, $P_3 = 789$

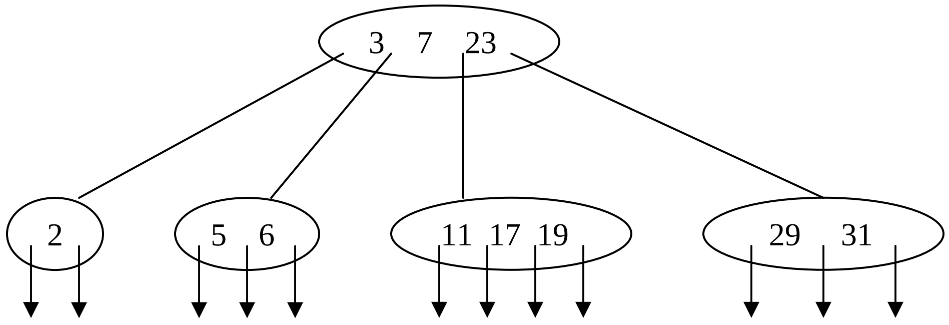
h(x) = 123 + 456 + 789 = 1566.

41. A balanced order-n multi way search tree in which each non root node contains at least $(n-1)/2$ keys is called a B-tree of order n. A key value appear only once, and can be anywhere in the tree. Since a key is present only once in the tree, it requires less space compare to a B^+ - tree of the same order.

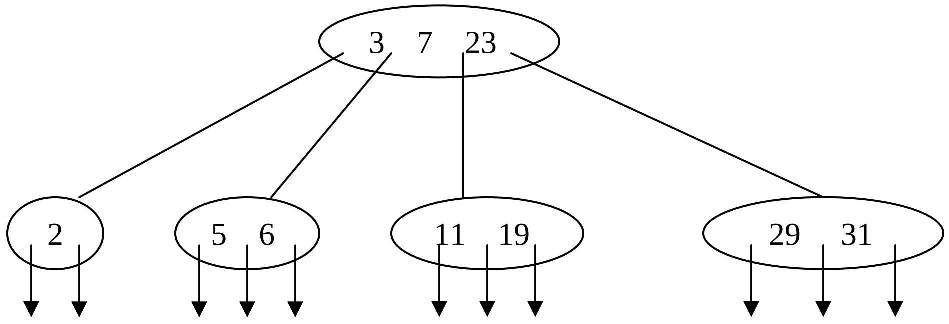
e.g., Given below is a B-tree of order 4.



If we try to insert element 6, number of keys in the first leaf node becomes 4 , which is greater the (n-1) . Hence the tree becomes unbalanced. Inorder to make the tree balanced, we split the node into two and take the middle element to the higher level. Hence the tree will look like as one given below.



Similarly deletion of element 17 will result in a tree as given below.



42. A weight of a tree is defied as the number of external nodes in the tree and it is equal to null pointers in the tree. If the ratio of the weight of the left sub tree of every node to the weight of the sub tree rooted at the node is between some fraction a and 1-a, the tree is a weight-balanced tree of ratio a or is said to be in the class *wb*[a]. when an ordinary insertion or deletion on a tree in class *wb*[a] would remove the tree from the class, rotations are used to restore the weight-balanced property.

43. If a request for a block of memory size *n* arrives, it search down the list of free blocks finding the first block of size >= *n* and allocate last *n* words out of that block. This allocation strategy is called first fit. The algorithm to perform first fit algorithm is given below.

```
function firstfit ( n int, p int)
{
        int q;
        p = link(av);  // av – start address
        q = av;
        while (p != 0)
        {
                if (size(p) >= n)
                {
                        setsize(p, size(p)-n);
                        if (size(p) == 0)
                                setlink(q, link(p));
                        else
                                p = p+size(p);
                        break;
                }
                q = p;
                p = link(p);
        }
}
```

44. If there are multiple records for a given key value, then that key is called a secondary key. Faster access to the records are provided by the use of indexes and link together by lists. The secondary key structures support access toall records that satisfy some <attribute,value> pair.
eg. {<A_i,v_{ij}> , (R_{ij1},…,R_{ijn})} where (R_{ij1},…,R_{ijn}) represents the record_list. The record_list may be maintained as a number of separate stored lists, for instance, hij, such that we have
< A_i,$v_{ij,}$ $n_{ij,}$ h_{ij} > (P_{ij1},…,P_{ijhij})
where n_{ij} is the number of records with value v_{ij} for the attribute A_i and P_{ijk} is the pointer to the kth stored list, for all k=1… h_{ij} . The average length of each stored list is n_{ij} / h_{ij} . There are two common methods of organizing the value-access file: the inverted index method and the multilist.

Inverted Index Files

The inverted index file contains the list of all records that satisfying the particular <attribute,value> pair in the index, wherein h_{ij} is equal to n_{ij} and each P_{jkm} points to a list of records of length one. In other words, a pointer for every record with the given value v_{ij} for the attribute A_i is kept in the index.

Multilist Files

The multilist file contains the list of all records that satisfying the particular <attribute,value> pair in the index, wherein h_{ij} is equal to i. There is only one stored list of length n_{ij}

Cellular Lists

Lists in a multilist file can become lengthy. The stored records may be distributed among manu physical storage units, or within the same storage unit in some manageable cluster of cylinders or some other manageable storage area, could be used to advantage by partitioning the lists along these boundaries. Thus, in a cellular list organization the lists are limited to be within a physical area of storage, referred to as a cell.

Ring Files

The last records of the lists in a multilist file points to a null record. In ring files the last record entry in each list points back to the index entry. Therefore, from any point within the list a forward traversal of the links would bring us to the index entry.

45. 1. Single level ordered Indexing

An index file is created which stores each value of the index field along with a list of pointers to all disk blocks that contain records with the field value. There are several types of ordered indexes.

i. Primary Indexes

A primary index is an ordered file whose records ate of fixed length with two fields, the primary key of the data field and a pointer to a disk block. There is one index entry in the index file for each block in the data file and it is to the first record in that block.

ii. Clustering Indexes

If records of a file are physically ordered on a non key field which does not have a distinct value for each record, that field is called the clustering field. An a index created to a clustering field is called a clustered index. This differs from a primary index, which requires that the ordering field of the data file have a distinct value for each record. A clustered index has two fields, one the clustering field and the second is a pointer to the first block in the data file that has a record with that value for its clustering field.

iii. Secondary Indexes

It is also an ordered file with two fields. The first field is of the same data type as some non ordering field of the data file that is an indexing field. The second field is either a block pointer or a record pointer. There are different types of secondary indexes

a. Secondary index structure on a key field

It has a distinct value for every record and is called a secondary key. In this case there is one index entry for each record in the data file, which contains the value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself. Such an index is called a dense index.

b. Secondary index structure on a non key field

In this case, numerous records in the data file can have the same value for the indexing field. It is commonly implemented by keeping index entries at a fixed length and have a single entry for each index field value but to create an extra level of indirection to handle the multiple pointers. Hence it is a non dense index.

2**. <u>Multilevel Indexing</u>**

Single level ordered indexing creates an ordered index file. A binary search can be a applied to the index file to locate pointers to a disk block or to a record in the file having a specific index field value and it requires approximately ($\log_2 b_i$) block access for an index with b_i blocks. The idea behind a multilevel index is to reduce the part of the index file that we continue to search by bfr_i, the blocking factor for the index, which is larger that 2. Hence, the search space is reduces much faster.

A multilevel index is maintained as a hierarchy of indexes. First level of the multilevel index is an ordered file with a distinct value for each key. Hence we create a primary index for the first level; this index to the first level is called the second level of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for each block of the first level. The blocking factor bfr_i for the second level and for all subsequent levels is the same as that for the first level index, because all index entries are ths same size; each has one field value and one block address. The third level is a primary index for the second level, has an entry for each second level block and the process repeats until all the entries of some level it in a single block.

46. <u>Mid-Square</u>

Square the identifier and then use appropriate number of bits from the middle of the square to obtain the bucket address.

<u>Division</u>

The identifier X is divided by some number M, and the remainder is used as the hash address for x.

<u>Folding</u>

Folding is an example of a uniform hash function (a function that does not result in a biased use of hash table for random inputs. i.e., probability that $h(x) = i$ to be $1/b$ for all buckets i) . In this method the identifier is partitioned into several parts, all but possibly the last being of the same length. These partitions are then added together to obtain the hash address for x. Folding itself is divided into two, depending on how we are performing the addition.

In the first method called *shift folding*, all but the last partition are shifted so that the least significant bit of each lines up with the corresponding bit of the last partition. The different partitions are added together to get $h(x)$.
e.g.

x = 123456789. The partitions are P_1 = 123, P_2 = 456, P_3 = 789
h(x) = 123 + 456 + 789 = 1368.

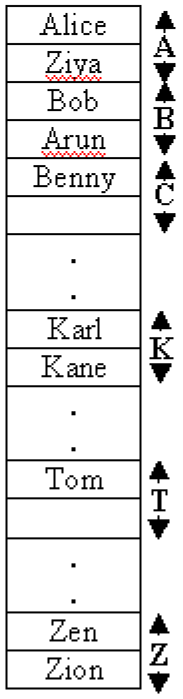
In the second method called folding at the boundaries, the identifier is folded at the partition boundaries, and digits falling into the same partition are added together to obtain *h*(*x*). This is equivalent to reversing alternate partition and then adding.
e.g.
x = 123456789. The partitions are P_1 = 123, P_2 = 654, P_3 = 789
h(x) = 123 + 456 + 789 = 1566.

Digit Analysis

Each identifier x is interpreted as a number using some radix r. The same radix is used for all the identifiers in the table. Using this radix, the digit of each identifier are examined. Digits having the most skewed distribution are deleted. Enough digits are deleted so that the number of remaining digits is small enough to give an address in the range of the hash table.

47. Linear Probing

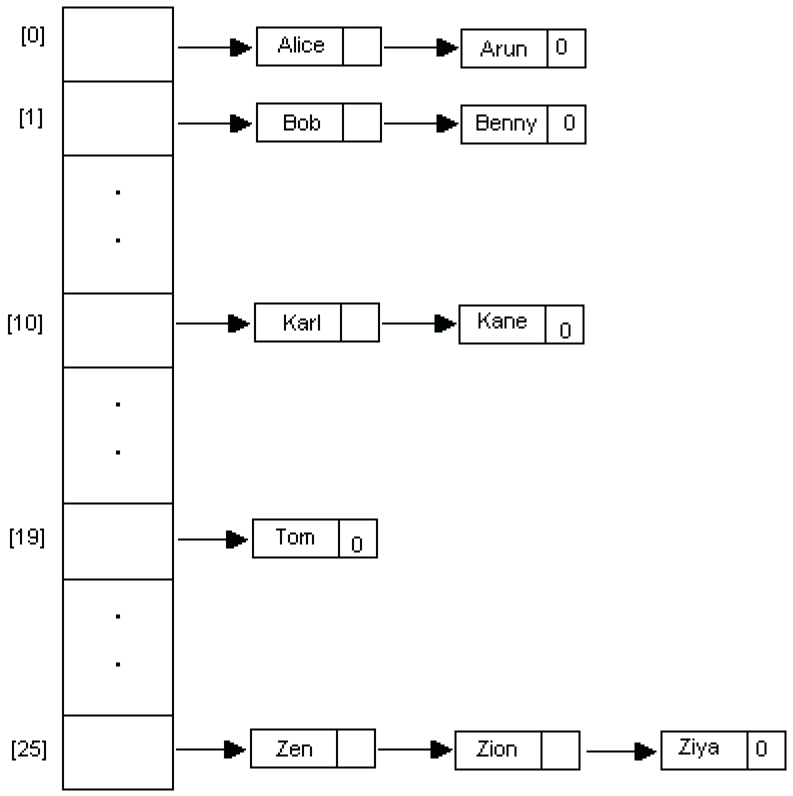
Linear probing is an overflow handing technique used while using static hashing method. In this method, we assume the hash function as an array. To detect collision and overflows, the hast table ht is initialized so that each slot contains the null identifier. When a new identifier is hashed into a full bucket, we find another bucket for this identifier as the closest unfilled bucket.
e.g. consider a hash function *h*(*x*) = first character of *x*. There are 26 buckets with 2 slots per bucket. The elements to be inserted be the names of students in a class. Let the names be Tom, Zen, Bob, Alice, Zion, Ziya, Arun, Karl, Benny, Kane. The hash table after inserting the identifiers will be as given below.



Chaining

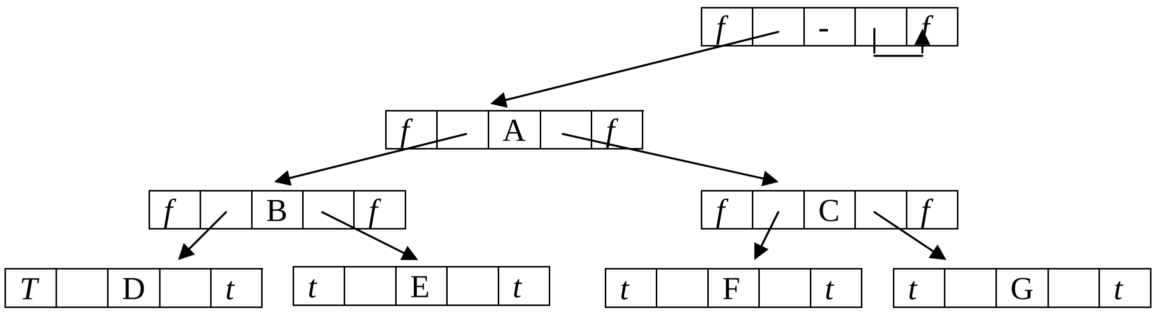
- Chaining is an overflow handling technique while using a static hashing method. An overflow is said to occur when a new identifier is hashed into a full bucket.
- Maintains a list of identifiers, one list per bucket organized as link list. Each list contains all the synonyms for that bucket.
- A search involves computing the hash function *f(x)* and examining the identifiers in the list.
- Each chain has a head node, usually smaller than other nodes (because it contains only the link field) and head nodes are arranged sequentially.

  Same elements considered above cane be placed using chaining as given below:



48. In a threaded binary tree proposed by A.J.Perlis and C. Thornton, trhe 0-links of a binary tree are replaced by pointers, called threads, to other nodes of the tree. These threads are constructed using the following rules.
    1. A 0 right child field in node p is replaced by a pointer to inorder successor of p.
    2. A 0 left child field in node p is replaced by a pointer to inorder predecessor of p.

    To distinguish between threads and normal pointers, two Boolean filelds, LeftThread and RightThread are added. If these values are true, the node contains a left thread and a right thread respectively; otherwise it contains a pointer to the left child and right child. By using the threads, we can perform an inorder traversal without making use of a stack. The algorithm for the same is given below.



```
Void Inorder()
{
    int * ch;
    for (ch = Next(); ch; ch=Next())
            print(ch->data);
}

int * Next()
{
    int * temp = CurrentNode->RightChild;
    if (!CurrentNode->RightThread)
            while(!temp->LeftThread)
                    temp = temp->LeftChild;
    CurerntNode = temp;
    if (CurrentNode = = root)
            return Null;
    else
            return CurrentNode;
}
```

49. Buddy system is a method of handling the storage management problem without frequent traversal through the list of free blocks. This is done by keeping separate free lists for blocks of different sizes. Each list contains free blocks of only one specific size. Initially, the entire memory of size 2^m is viewed as a single free block. For each power of two between 1 and 2^m ,a free list containing blocks of that size is maintained. A block of size 2^i is called *i-block* and the free list containing i-blocks is called the *i-list*. Initially, all the list except the m-list are empty. If a request for a block of size n is made, an I-block is reserved where I is the smallest integer such that $n<=2^i$. If no I-block is available an (i+1)-block is removed from the (i+1)-list and is split into two equal size buddies. Each of these buddies is an I-block. One of the buddies is allocated, and the other remains free and is placed on the I-list. If the (i+1)-block is also unavailable, an (i+2)-block is split into (i+1)-block buddies, one of which is placed on the (i+1)-list and the other is split into two I-blocks. One of these is allocated and the other is placed onto the i-list. This is continued. An outline of the above function is as follows:

```
if (the I-list is not empty)
{
        p = the address of the first block in the i-list;
        remove the first block from the i-list;
        return (p);
}
else
{
        if (i= =m)
                return (null);
        else
        {
                p = getblock($2^i$ +1);
                if ( p == null)
                        return (null);
                else
                {
                        put the i-block starting at location p on the i-list;
                        return ( p+$2^i$);
                }
        }
}
```

The liberation algorithm can be outlined as a recursive routine that frees an i-block at location alloc. The algorithm is given below.

```
if ((i = = m) or (the i-buddy of alloc is not free))
        add the i-block at alloc to the i-list;
else
{
        remove the i-buddy of alloc from the i-list;
        combine the i-block at alloc with its i-buddy;
        p = the address of the newly formed (i + 1)-block;
        liberate(p , i +1);
}
```

50. Garbage collection is the process of collecting all unused nodes and returning them to available space. This process is carried out in essentially two phases. In the first phases, known as the marking phase, all nodes in use are marked. In the second phase all unmarked nodes are returned to the available list. In Compaction we compact memory so that all free nodes form a contiguous block of memory. Compaction reduces the average retrieval time.

Marking

Marking algorithm mark all directly accessible as well as the indirectly accessible nodes. The structure of a node is as below.

mark	tag	dlink	rlink

Each node regardless of its usage will have two Boolean value, *mark* and *tag*. In order to carry out the marking, a mark field is kept in each node. If *mark* is true, it is a used node else an unused one. If the *tag* value is false, then the node contains only atomic information in a field called *data*. If *tag* = true, then the node has two link values *dlink* and *rlink*.

An algorithm for marking the nodes is given below:

```
Function mark ( int *x)
{
    int *p, *q;, top;
    top = 0;
    add(x);
    while (top > 0)
    {
        delete(p);
        do
        {
            q = p->rlink;
            if (q != null)
            {
                if ( q->tag and !q->mark)
                    add(q);
                q->mark = true;
            }
            p = p->dlink;
            if (p = = null)
                break;
            if (p->mark || !p>tag)
                break;
            p->mark = true;
        }
        p->mark = true;
    }
}
```

Storage Compaction

During this phase, all free space form one contiguous block. Since there will, in general, be links from one node to another, storage compaction must update these links to point to the relocated address of the respective nodes.

Storage compaction is done in as three phases. In the first phase, we find the new address for the used blocks. In the second phase, we change the link value to the new address. In the third phase, we relocate the node to the new address. Algorithm for storage compaction is given below.

```
Function compact()
{
    int i,j,k;

    // Phase I
    av = 0; i=0;
    while ( i <= m)
    {
```

```
                if (mark(i))
                {
                        newaddr(i) = av;
                        av = av + size(i);
                }
                i = i + size(i);
        }
        // Phase II
        i=0;
        while ( i <= m)
        {
                if (mark(i))
                {
                        link1(i) = newaddr(link1(i));
                        link2(i) = newaddr(link2(i));
                }
                i = i + size(i);
        }


        // Phase III
        i=0;
        while ( i <= m)
        {
                if (mark(i))
                {
                        k = i – newaddr(i);
                        l = newaddr(i);
                        for (j=i; j<I+size(i) – 1; j++)
                                memory[j-k] = memory[j];
                        i = i + size(l);
                }
                i = i + size(i);
        }
}
```