SORTING & HASHING

☞ Chapter at a Glance

- A **hash table** data structure is just like an array. Data is stored into this array at specific index generated by a hash function. A **hash function** hashes (converts) a number in a large range into a number in a smaller range.
- **Linear search:** Linear or Sequential Search is a method where the search begins at one end of the list, scans the elements of the list from left to right (if the search begins from left) until the desired record is found.
  This type of searching can be performed in an ordered list or an unordered list. For ordered lists that must be accessed sequentially, such as linked lists or files with variable-length records lacking an index, the average performance can be improved by giving up at the first element which is greater than the unmatched target value, rather than examining the entire list.
- **Binary Search:** In Binary Search the entire sorted list is divided into two parts. We first compare our input item with the mid element of the list and then restrict our attention to only the first or second half of the list depending on whether the input item comes left or right of the mid-element. In this way we reduce the length of the list to be searched by half.
- **Hashing:** In hash tables, there is always a possibility that two data elements will hash to the same integer value. When this happens, a **collision** occurs i.e. two data members try to occupy the same place in the hash table array. There are methods to deal with such situations like **Open Addressing and Chaining.**
- **Bubble sort:** Given an array of unsorted elements, Bubble sort performs a sorting operation on the first two adjacent elements in the array, then between the second & third, then between third & fourth & so on.
- **Insertion sort:** In insertion sort data is sorted data set by identifying an element that is out of order relative to the elements around it. It removes that element from the list, shifting all other elements up one place.
  Finally it places the removed element in its correct location.
  For example, when holding a hand of cards, players will often scan their cards from left to right, looking for the first card that is out of place. If the first three cards of a player's hand are 4, 5, 2, he will often be satisfied that the 4 and the 5 are in order relative to each other, but upon getting to the 2, desires to place it before the 4 and the 5. In that case, the player typically removes the 2 from the list, shifts the 4 and the 5 one spot to the right, and then places the 2 into the first slot on the left.
- **Quick sort:** In Quick-Sort we divide the array into two halves. We select a pivot element (normally the middle element of the array) and perform a sorting in such a manner that all the elements to the left of the pivot element is lesser than it & all the elements to it's right are greater than the pivot element. Thus we get two sub arrays.
- **Merge sort:** In this method, we divide the array or list into two sub arrays or sub lists of nearly equal as possible and then sort them separately. Then the sub-arrays are again divided into another sub arrays.
- **Heap sort:** A heap takes the form of a binary tree with the feature that the maximum or minimum element is placed in the root. Depending upon this feature the heap is called max

heap or min-heap respectively. After the heap construction the elements from the root are taken out from the tree and the heap structure is reconstructed. This process continues until the heap is empty.

Multiple Choice Type Questions

1. The ratio of the number of items in a hash table, to the table size is called the [WBUT 2007, 2009, 2016]
a) load factor b) item factor c) balanced factor d) all of these
Answer: (a)

2. Which of the following is not a requirement of good hashing function?
a) Avoid collision b) Reduce the storage space
c) Make faster retrieval d) None of these [WBUT 2008, 2015]
Answer: (b)

3. Stability of Sorting Algorithm is important for [WBUT 2007]
a) Sorting records on the basis of multiple keys
b) Worst case performance of sorting algorithm
c) Sorting alpha numeric keys as they are likely to be the same
d) None of these
Answer: (a)

4. Which of the following is the best time for an algorithm? [WBUT 2007]
a) O(n) b) O(log2 n) c) O(2n) d) O(n log2 n)
Answer: (b)

5. The Linear Probing Technique for collision resolution can lead to [WBUT 2009]
a) Primary clustering b) Secondary clustering
c) Overflow d) Efficiency storage utilization
Answer: (a)

6. The fastest sorting algorithm for an almost already sorted array is [WBUT 2009]
a) quick sort b) merge sort c) selection sort d) insertion sort
Answer: (d)

7. The time complexity of binary search is [WBUT 2009]
a) $O(n^2)$ b) O(n) c) O(log n) d) O(n log n)
Answer: (c)

8. The best case time complexity of Bubble sort technique is [WBUT 2010]
a) $O(n)$ b) $O(n^2)$ c) $O(n \log n)$ d) $O(\log n)$
Answer: (a)

9. Which of the following sorting procedures is the slowest? [WBUT 2010]
a) Quick sort b) Heap sort c) Merge sort d) Bubble sort
Answer: (d)

10. Which of the following traversal techniques lists the elements of a binary search tree in ascending order? [WBUT 2011, 2015]
a) pre-order b) Post-order c) Inorder d) None of these
Answer: (c)

11. Binary search cannot be used in linked lists. [WBUT 2011]
a) True b) False
Answer: (b)

12. Breadth-first-search algorithm uses.....data structure [WBUT 2011]
a) stack b) queue c) binary tree d) none of these
Answer: (b)

13. The best case complexity of insertion sort is – [WBUT 2011]
a) $O(n^2)$ b) $O(n \log_2 n)$ c) $O(n^3)$ d) $O(n)$
Answer: (d)

14. Which of the following is not related to hashing? [WBUT 2011]
a) Synonyms b) Collision c) Balance d) Load factor
Answer: (c)

15. A machine needs a minimum of 100sec to sort 1000 names by quick sort. The minimum time needed to sort 100 names will be approximately [WBUT 2012]
a) 72.7 sec b) 11.2 sec c) 50.2 sec d) 6.7 sec
Answer: (d)

16. What will be the time complexity for selection sort to sort an array of n elements? [WBUT 2012, 2016]
a) O(log n) b) O(n log n) c) O(n) d) $O(n^2)$
Answer: (d)

17. The best sorting technique when the data is almost sorted is [WBUT 2013]
a) Selection sort b) Bubble sort c) Quick sort d) Insertion sort
Answer: (d)

18. Which of the following is a hash function? [WBUT 2014]
a) Quadratic probing b) chaining
c) open addressing d) folding
Answer: (a)

19. The numl
ascending or
a) 11
Answer: (d)

20. Binary sea
a) divide a
c) heuristi
Answer: (b)

21. Merge sor
a) divide a
c) heuristi
Answer: (a)

22. The prereq
a) unsorte
c) descend
Answer: (d)

1. Explain the a

Answer:
A sequential sea
item, and so on
occur in the grou
A binary search
the array to dete
strategy on only
The benefit of b
100 elements. F
simple incremen
Thus for large
worthwhile for s
Binary search is
middle term).

2. What is hashi

Define Hashing.
What is collision
Explain Linear P

19. The number of swapping needed to sort numbers 8, 22, 7, 9, 31, 19, 5, 13 in entire ascending order using bubble sort is [WBUT 2014]
a) 11 b) 12 c) 13 d) 14
Answer: (d)

20. Binary search uses [WBUT 2014]
a) divide and reduce strategy b) divide and conquer strategy
c) heuristic search d) both (a) and (b)
Answer: (b)

21. Merge sort uses [WBUT 2015]
a) divide and conquer strategy b) backtracking approach
c) heuristic search d) greedy approach
Answer: (a)

22. The prerequisite condition of Binary search is [WBUT 2017]
a) unsorted array b) ascending order array
c) descending order array d) sorted array
Answer: (d)

Short Answer Type Questions

1. Explain the advantages of binary search over sequential search. [WBUT 2007, 2013]

Answer:
A sequential search of either a list, an array, or a chain looks at the first item, the second item, and so on until it either finds a particular item or determines that the item does not occur in the group. Average case of sequential search is $O(n)$
A binary search of an array requires that the array be sorted. It looks first at the middle of the array to determine in which half the desired item can occur. The search repeats this strategy on only this half of the array.
The benefit of binary search over linear search becomes significant for lists over about 100 elements. For smaller lists linear search may be faster because of the speed of the simple increment compared with the divisions needed in binary search.
Thus for large lists binary search is very much faster than linear search, but is not worthwhile for small lists.
Binary search is not appropriate for linked list structures (no random access for the middle term).

2. What is hashing? [WBUT 2007, 2012, 2014, 2017]
OR, [WBUT 2015]
Define Hashing. [WBUT 2017]
What is collision? [WBUT 2007]
Explain Linear Probing & Quadratic Probing with example.

OR,

Define 'Hashing'. Explain with a suitable example the collision resolution scheme using linear probing with open addressing. [WBUT 2010]

OR,

Define Hashing. Explain one collision resolution scheme citing one example. [WBUT 2013]

OR, [WBUT 2017]

Write two hash functions

Answer:

1st Part:

Hashing is a method for storing and retrieving records from a database. It lets you insert, delete, and search for records based on a search key value. When properly implemented, these operations can be performed in constant time. In fact, a properly tuned hash system typically looks at only one or two records for each search, insert, or delete operation. This is far better than the O(log *n*) average cost required to do a binary search on a sorted array of *n* records, or the O(log *n*) average cost required to do an operation on a binary search tree. However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly. Designers need to pay careful attention to all of the details involved with implementing a hash system.

A hash system stores records in an array called a **hash table**, which we will call HT. Hashing works by performing a computation on a search key *K* in a way that is intended to identify the position in **HT** that contains the record with key *K*. The function that does this calculation is called the **hash function**, and is usually denoted by the letter **h**. Since hashing schemes place records in the table in whatever order satisfies the needs of the address calculation, records are not ordered by value. A position in the hash table is also known as a **slot**. The number of slots in hash table **HT** will be denoted by the variable *M* with slots numbered from 0 to *M* - 1.

2nd Part:

A **collision** between two keys K & K' occurs when both have to be stored in the table & both hash to the same address in the table.

3rd Part:

Open addressing: It is a general collision resolution scheme for a hash table. In case of collision, other positions of the hash table are checked **(a probe sequence)** until an empty position is found.

The different types of Open addressing scheme includes

a) Linear Probing (Sequential Probing)
b) Quadratic Probing
c) Double Hashing (Re Hashing)

Linear probing is a used for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location. This is accomplished using two values - one as a starting value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the

stepsize, is repeatedly added to the starting value until a free space is found, or the entire table is traversed.

The function for the rehashing is the following:

rehash(key)=(n+1) % k;

For example, we have a hash table that could accommodate 9 information, and the data to be stored were integers. To input 27, we use hash(key)= 27 % 9=0. Therefore, 27 is stored at 0. If another input 18 occurs, and we know that 18 % 9= 0, then a collision would occur. In this event, the need to rehash is needed. Using linear probing, we have the rehash(key)= (18+1) % 9= 1. Since 1 is empty, 18 can be stored in it.

Quadratic probing:

In order to prevent collision we use quadratic probing scheme.

- In quadratic probing,
- We start from the original hash location i
- If a location is occupied, we check the locations $i+1^2$, $i+2^2$, $i+3^2$, $i+4^2$ ...
- We wrap around from the last table location to the first table location if necessary.

Let us take the following example:

Table Size is 11 (0..10)

- Hash Function: h(x) = x mod 11
- Insert keys (20, 30,2,13,25,24,10,9):
  - 20 mod 11 = 9
  - 30 mod 11 = 8
  - 2 mod 11 = 2
  - 13 mod 11 = 2 → $2+1^2=3$
  - 25 mod 11 = 3 → $3+1^2=4$
  - 24 mod 11 = 2 → $2+1^2$, $2+2^2=6$
  - 10 mod 11 = 10
  - 9 mod 11 = 9 → $9+1^2$, $9+2^2$ mod 11, $9+3^2$ mod 11 =7

The figure below shows the corresponding entries in the hash table.

0	
1	
2	2
3	13
4	25
5	
6	24
7	9
8	30
9	20
10	10

3. Prove that, the best case time complexity for quick sort is O(nlogn) for input size of n. [WBUT 2008]

Answer:

The analysis of the procedure QUICK_SORT is given by

$T(N) = P(N) + T(J-LB) + T(UB - J)$

where P(N), T(J-LB), and T(UB-J) denote the times to partition the given table, sort the left subtable, and sort the right subtable, respectively. Note that the time to partition a table is O(N). The worst case occurs when, at each invocation of the procedure, the current table is partitioned into two subtables with one of them being empty (that is J = LB or J = UB). Such a situation, for example, occurs when the given key set is already sorted.

The worst case time analysis, assuming J = LB, then becomes

$Tw = P(N) + Tw(0) + Tw(N-1)$
$= c*N + Tw(N-1)$
$= c*N + c*(N-1) + Tw(N-2)$
$= c*N + c*(N-1) + c*(N-2) + Tw(N-3)$

.
.
.

$= c * \{(N+1)(N)\}/2 = O(N2)$

The best case analysis occurs when the table is always partitioned in half, that is, J = [(LB+UB)/2]. The analysis becomes:

$Tb = P(N) + 2Tb(N/2)$
$= c*N + 2Tb(N/2)$
$= c*N + 2c(N/2) + 4Tb(N/4)$
$= c*N + 2c(N/2) + 4c(N/4) + 8Tb(N/8)$

.
.
.

$= (log2N)*c*N + 2log2N*Tb(1)$
$= O(Nlog2N)$

4. Give an algorithm to search for an element in an array using binary search. [WBUT 2008]

OR,

What is the precondition of performing binary search in an array? Write the Binary Search algorithm. [WBUT 2010]

Answer:

Let us consider an array A of size N = 10. The left index and right index are

left = 0 right = (N–1) = 9

```
int bin_search(int A[], int n, int item)
{
int left = 0,
right = n - 1;
int flag = 0; //a flag to indicate whether the element is found
int mid = 0;
```



```
while (left
{
mid = (left
if (item ==
{
flag = 1; /
break;
}
else if (ite
right = mid
else
left = mid +
}
return flag;
}
```

5. "Binary sea the validity of

Answer:

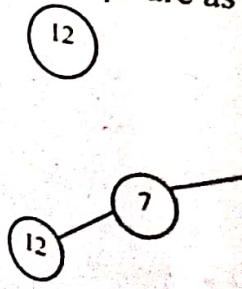
The statement i however it wou Binary search element in the saying array[mi The same canno value of the mi value of any no One solution to search is to hav the middle. Dec the linked list. C

6. Draw a minir 12, 11, Now do the he the insertion so

Answer:

1st Part:

The steps are as



```
while (left <= right) // till left and right cross each other
{
mid = (left + right)/2; // divide the array into two halves
if (item == A[mid])
{
flag = 1; //set flag = 1if the element is found
break;
}
else if (item < A[mid])
right = mid - 1; /* compute new right from the right sub array */
else
left = mid + 1; /* compute new left from the left sub array */
}
return flag;
}
```

5. "Binary search technique cannot be implemented using Linked list." ---- Justify the validity of the statement. **[WBUT 2009]**

Answer:

The statement is not true as it can Binary Search can be implemented using Linked List – however it would be less efficient than array.

Binary search on an array so fast and efficient because of its the ability to access any element in the array in constant time. Thus one can get to the middle of the array just by saying array[middle].

The same cannot be done with a linked list. One needs to write an algorithm to get the value of the middle node of a linked list. In a linked list, one loses the ability to get the value of any node in a constant time.

One solution to the inefficiency of getting the middle of the linked list during a binary search is to have the first node contain one additional pointer that points to the node in the middle. Decide at the first node if one needs to check the first or the second half of the linked list. Continue doing that with each half-list.

6. Draw a minimum heap tree from the below list: **[WBUT 2012, 2014]**

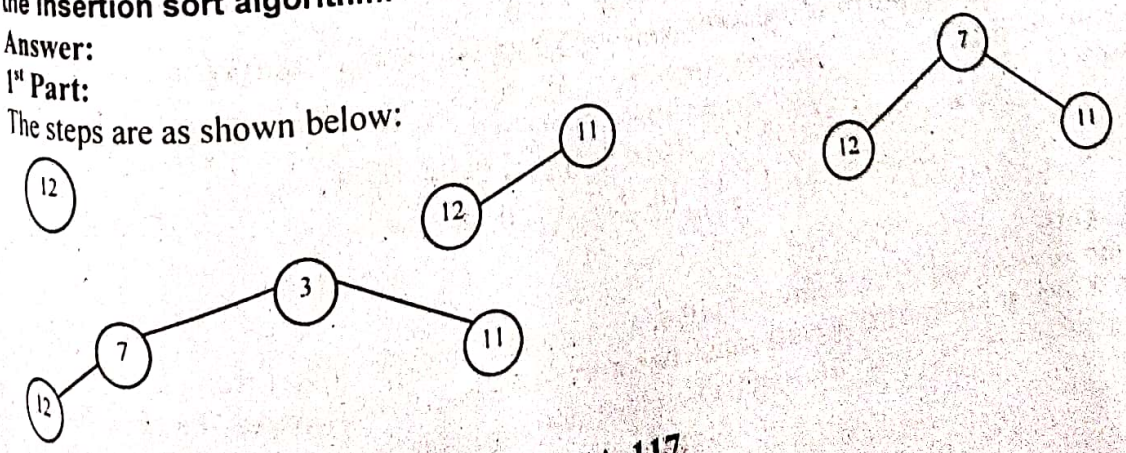
12, 11, 7, 3, 10, –5, 0, 9, 2

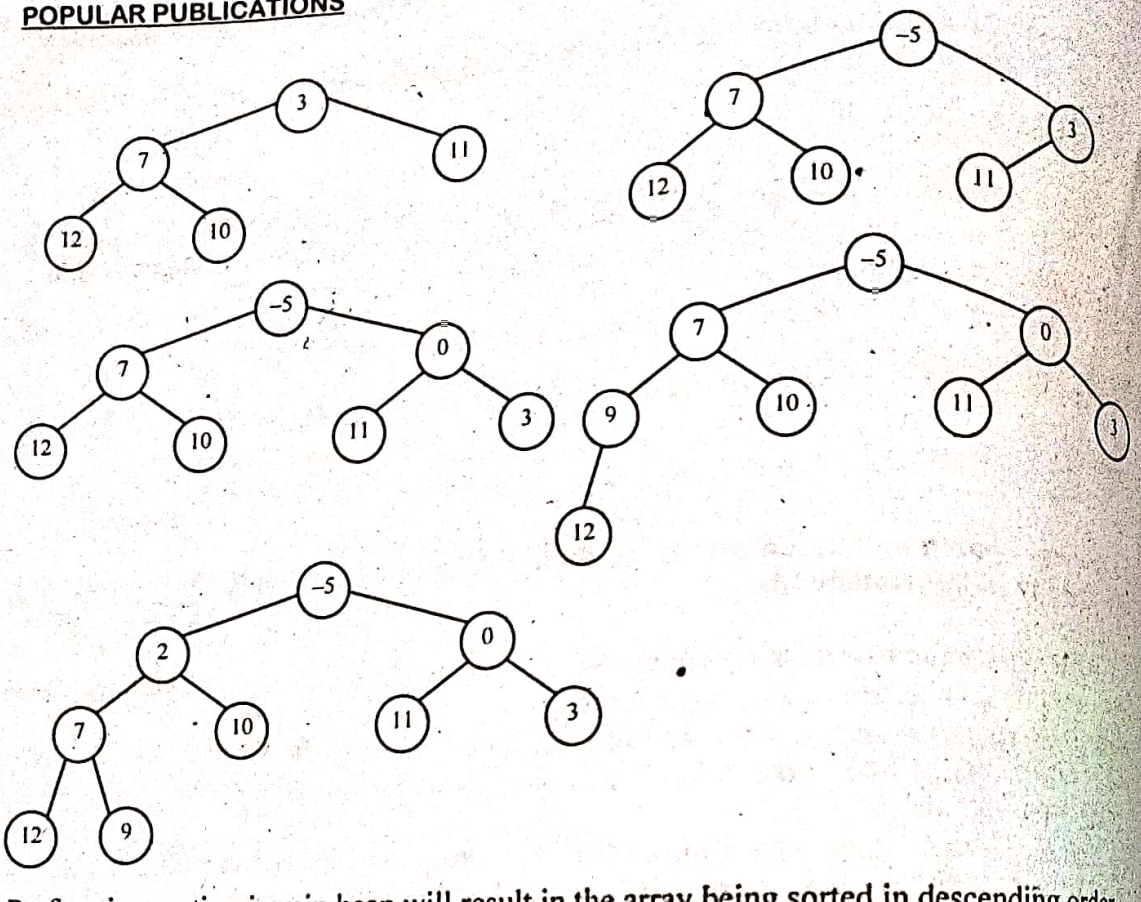
Now do the heap sort operation over the heap tree which you have formed. Write the insertion sort algorithm.

Answer:

1st Part:

The steps are as shown below:





Performing sorting in min heap will result in the array being sorted in descending order.
The array now looks like: -5, 2, 0, 7, 10, 11, 3, 12, 9

The steps for sorting are:

Tree nodes: 9, 2, 0, 7, 10, 11, 3, 12 Sorted Array: -5
Tree nodes: 0, 2, 9, 7, 10, 11, 3, 12 Sorted Array: -5
Tree nodes: 12, 2, 9, 7, 10, 11, 3 Sorted Array: 0, -5
Tree nodes: 3, 2, 9, 7, 10, 11, 12 Sorted Array: 0, -5
Tree nodes: 2, 3, 9, 7, 10, 11, 12 Sorted Array: 0, -5
Tree nodes: 3, 9, 7, 10, 11, 12 Sorted Array: 2, 0, -5
Tree nodes: 9, 7, 10, 11, 12 Sorted Array: 3, 2, 0, -5
Tree nodes: 7,9, 10, 11, 12 Sorted Array: 3,2, 0, -5
Tree nodes: 9, 10, 11, 12 Sorted Array: 7, 3,2, 0, -5
Tree nodes: 10, 11, 12 Sorted Array: 9, 7, 3,2, 0, -5
Tree nodes: 11, 12 Sorted Array: 10, 9, 7, 3,2, 0, -5
Tree nodes: 12 Sorted Array: 11, 10, 9, 7, 3,2, 0, -5
Tree nodes: Sorted Array: 12, 11, 10, 9, 7, 3,2, 0, -5

2nd Part:

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. This is called insertion sort. An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping

them sorted).
sorted sequenc

Pseudocode

We use a proce
length *n* of the
the array, with
INSERTION_
1. FOR j ←
2. DO
3. {
4. i
5. V
6.
7.
8. A

7. Write the pse

Answer:

The Algorithm f

```
#include <st
#include <co
#define N 6
void buildhe
void heapsor
int main(voi
{
  int heapAr
  int i;
  printf("\n
  for (i = 0
  printf("%d
  buildheap(
  heapsort(he
  printf("\nA
  for (i = 0;
    printf("%
  getch();
  return 0;
}
void buildhea
{
  int i, val,
  for (i = 1;
  {
    val = x[i
    s = i;
```

them sorted). Insertion sort is an example of an **incremental** algorithm; it builds the sorted sequence one number at a time.

Pseudocode

We use a procedure INSERTION_SORT. It takes as parameters an array $A[1..n]$ and the length n of the array. The array A is sorted in place: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

```
INSERTION_SORT (A)
1.  FOR j ← 2 TO length[A]
2.      DO key ← A[j]
3.         {Put A[j] into the sorted sequence A[1 . . j − 1]}
4.         i ← j − 1
5.         WHILE i > 0 and A[i] > key
6.               DO A[i +1] ← A[i]
7.                  i ← i − 1
8.         A[i + 1] ← key
```

7. Write the pseudo code for Heap sort. **[WBUT 2013]**

Answer:

The Algorithm for heap-sort is as follows.

```
#include <stdio.h>
#include <conio.h>
#define N 6
void buildheap(int[], int);
void heapsort(int[], int);
int main(void)
{
 int heapArr[N] ={15, 19, 10, 7, 17, 6};
 int i;
 printf("\nBefore Sorting:\n");
 for (i = 0; i < N; i++)
 printf("%d\t", heapArr[i]);
 buildheap(heapArr, N);
 heapsort(heapArr, N);
 printf("\nAfter Sorting:\n");
 for (i = 0; i < N; i++)
  printf("%d\t", heapArr[i]);
 getch();
 return 0;
}
void buildheap(int x[], int n)
{
 int i, val, s, f;
 for (i = 1; i < n; i++)
 {
  val = x[i];
  s = i;
```

```
    f = (s - 1) / 2;
    while (s > 0 && x[f] < val)
    {
      x[s] = x[f];
      s = f;
      f = (s - 1) / 2;
    }
    x[s] = val;
  }
}
void heapsort(int x[], int n)
{
  int i, s, f, ivalue;
  for (i = n - 1; i > 0; i--)
  {
    ivalue = x[i];
    x[i] = x[0];
    f = 0;
    if (i == 1)
      s =  - 1;
    else
      s = 1;
    if (i > 2 && x[2] > x[1])
      s = 2;
    while (s >= 0 && ivalue < x[s])
    {
      x[f] = x[s];
      f = s;
      s = 2 * f + 1;

      if (s + 1 <= i - 1 && x[s] < x[s + 1])
        s++;
      if (s > i - 1)
        s =  - 1;
    }
    x[f] = ivalue;
  }
}
```

8. Deduce the average time complexity of Quicksort algorithm. [WBUT 2015]

Answer:

The basic idea of Quicksort is as given below:

1. Pick one element in the array, which will be the *pivot*.
2. Make one pass through the array, called a *partition* step, re-arranging the entries so that:
   - the pivot is in its proper place.
   - entries smaller than the pivot are to the left of the pivot.
   - entries larger than the pivot are to its right.

3. Recu
and t

Analysis

T(N) = T(i) +
The time to so
o
o
o

The average v
1/N S T(j), j =
T(N) = 2/N (S
Multiply by N
NT(N) = 2(S
To remove the
(N-1)T(N-1) =
and subtract:
NT(N) - (N-1)
Prepare for tel
NT(N) = (N+1
Divide by N(N
T(N)/(N+1) =
Telescope:
T(N)/(N+1) =
T(N-1)/(N) = T
T(N-2)/(N-1) =
....
T(2)/3 = T(1)/2
Add the equati
T(N)/(N+1) =
T(N) = (N+1)(
The sum S (1/j)
Thus **T(N) = O**

9. What is the data?

Answer:

A binary searc
determine rows
set of data in h
supplied value
than the value
need to compare

3. Recursively apply quicksort to the part of the array that is to the left of the pivot, and to the right part of the array.

Analysis

$T(N) = T(i) + T(N - i - 1) + cN$

The time to sort the file is equal to

- the time to sort the left partition with **i** elements, plus
- the time to sort the right partition with **N-i-1** elements, plus
- the time to build the partitions

The average value of T(i) is 1/N times the sum of T(0) through T(N-1)

$1/N \; S\, T(j), \; j = 0 \text{ thru } N-1$

$T(N) = 2/N \; (S\, T(j)) + cN$

Multiply by N

$NT(N) = 2(S\, T(j)) + cN*N$

To remove the summation, we rewrite the equation for N-1:

$(N-1)T(N-1) = 2(S\, T(j)) + c(N-1)^2, \; j = 0 \text{ thru } N-2$

and subtract:

$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN - c$

Prepare for telescoping. Rearrange terms, drop the insignificant c:

$NT(N) = (N+1)T(N-1) + 2cN$

Divide by N(N+1):

$T(N)/(N+1) = T(N-1)/N + 2c/(N+1)$

Telescope:

$T(N)/(N+1) = T(N-1)/N + 2c/(N+1)$

$T(N-1)/(N) = T(N-2)/(N-1) + 2c/(N)$

$T(N-2)/(N-1) = T(N-3)/(N-2) + 2c/(N-1)$

....

$T(2)/3 = T(1)/2 + 2c/3$

Add the equations and cross equal terms:

$T(N)/(N+1) = T(1)/2 + 2c \; S\, (1/j), \; j = 3 \text{ to } N+1$

$T(N) = (N+1)(1/2 + 2c \; S(1/j))$

The sum $S\, (1/j), \; j = 3$ to N-1, is about LogN

Thus **$T(N) = O(N \log N)$**

9. What is the primary criterion of performing binary search technique on a list of data? [WBUT 2015]

Answer:

A binary search algorithm is one method of efficiently processing a sorted list to determine rows that match a given value of the sorted criteria. It does so by "cutting" the set of data in half (thus the term binary) repeatedly, with each iteration comparing the supplied value with the value where the cut was made. If the supplied value is greater than the value at the cut, the lower half of the data set is ignored, thus eliminating the need to compare those values. The reverse happens when the skipped to value is less than

the supplied search criteria. This comparison repeats until there are no more values to compare.
The binary search algorithm was able to eliminate the need to do a comparison on each of the records, and in doing so reduced the overall computational complexity of our request for the database server. Using the smaller set of sorted weight data, we are able to avoid needing to load all the record data into memory in order to compare the product weights to our search criteria.

10. What is Load Factor? Why do we need hashing? How does a hash table allow O(1) searching? Why is prime number chosen for computing a hash function? **[WBUT 2018]**

Answer:
1st Part:
Loads factor refers to the ratio of the number of records to the number of addresses within a data structure.
2nd Part:
Hashing is used to inbox and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.
3rd Part:
A hash table is an array containing all of the keys to search on. The position of each key in the array is determined by the hash function, while can be any function which always maps the same input to the same output. So, we shall assume the hash function as O(1).
4th Part:
The prime numbers are used to minimize collisions when the data exhibits some particular pattern. The reason prime numbers are used to neutralize the effect of patterns in the keys in the distribution of collisions of a hash function.

Long Answer Type Questions

1. a) Explain with an example the Merge sort algorithm. **[WBUT 2007]**
b) Write an algorithm for Merge sort. **[WBUT 2007]**

OR,

Show the operation of merge sort with an example. **[WBUT 2013]**
c) Compare the best case time complexity of selection sort and insertion sort. **[WBUT 2007]**

Answer:
a) Merge-sort is based on the divide-and-conquer paradigm. The Merge-sort algorithm can be described in general terms as consisting of the following three steps:
1. Divide Step
If given array A has zero or one element, return S; it is already sorted. Otherwise, divide A into two arrays, A_1 and A_2, each containing about half of the elements of A.
2. Recursion Step
Recursively sort array A_1 and A_2.

3. Conquer Step

Combine the elements back in A by merging the sorted arrays A_1 and A_2 into a sorted sequence.

b)
```
/*Recursive Merge Sort algorithm*/
mergesort(int a[], int low, int high)
{
    int mid;
    if(low<high)
    {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,high,mid);
    }
    return(0);
}
/* Merge function*/
merge(int a[], int low, int high, int mid)
{
    int i, j, k, c[50];  /* assume size of array is 50*/
    i=low;
    j=mid+1;
    k=low;
    while((i<=mid)&&(j<=high))
    {
        if(a[i]<a[j])
        {
            c[k]=a[i];
            k++;
            i++;
        }
        else
        {
            c[k]=a[j];
            k++;
            j++;
        }
    }
    while(i<=mid)
        {
            c[k]=a[i];
            k++;
            i++;
        }
    while(j<=high)
        {
            c[k]=a[j];
            k++;
```

```
                j++;
            }
        for(i=low;i<k;i++)
            {
                a[i]=c[i];
            }
}
```

Let the original unsorted array is: 8 4 5 2 5
In the first step it gets split into
8 4 ||| 5 2 5 (||| denotes the divider)
Further splitting and merging is shown in the next steps:
8 ||| **4** ||| 5 2 5 (split)
8 ||| 4 ||| 5 2 5 (can't split, merges a single-element array and returns)
8 ||| **4** ||| 5 2 5 (can't split, merges a single-element array and returns)
4 8 ||| 5 2 5 (merges the two segments into the array)
4 8 ||| **5** ||| **2 5** (split)
4 8 ||| **5** ||| 2 5 (can't split, merges a single-element array and returns)
4 8 ||| 5 ||| **2** ||| **5** (split)
4 8 ||| 5 ||| **2** ||| 5 (can't split, merges a single-element array and returns)
4 8 ||| 5 ||| 2 ||| **5** (can't split, merges a single-element array and returns)
4 8 ||| 5 ||| **2 5** (merges the two segments into the array)
4 8 ||| **2 5 5** (merges the two segments into the array)
2 4 5 5 8 (merges the two segments into the array)

c) Selection sort Performance: Best Case is $O(n^2)$ because even if the list is sorted, the same number of selections must still be performed.
Insertion Sort Performance: The *best-case* time complexity is when the array is already sorted, and is O(n).

2. a) Explain with a suitable example, the principle of operation of Quick sort. **[WBUT 2007, 2009, 2010]**

Answer:
In Quick-Sort we divide the array into two halves. We select a pivot element (normally the middle element of the array) & perform a sorting in such a manner that all the elements to the left of the pivot element is lesser than it & all the elements to it's right is greater than the pivot element. Thus we get two sub arrays. Then we recursively call the quick sort function on these two sub arrays to perform the necessary sorting.
Let us consider the following unsorted array:
a [ ] = 45 26 77 14 **68** 61 97 39 99 90

Step 1:
We choose two indices as left = 0 and right = 9. We find the pivot element by the formula a[left + right] / 2 = a [0 + 9] / 2 = a [4]. So the **pivot element is a [4] = 68.** We also store with a [left] = a [0] = 45 and a [right] = a [9] = 90

Step 2:
We compa
by 1 each
We record
Step 3:
We compa
right by 1
element.
right=7.
Step 4:
Since left
The array
Step 5:
We further
i.e., **left =**
We repeat
We will se
will find th
pivot eleme
of quick so

b) Find the
Answer:
Refer to Q

3. a) Why is
Answer:
A **heuristic**
problem in
there is no
be proven t
typically us
given constr
placed into a
We use Hash
1) Perform
2) It increa
data and
That's why h

b) What is th

Step 2:
We compare all the elements to the left of the pivot element. We increase the value of left by 1 each time, when an element is less than 68. We stop when there is no such element. We record the latest value of left. In our example the left value is 2 i.e., **left=2.**

Step 3:
We compare all the elements to the right of the pivot element. We decrease the value of right by 1 each time when an element is greater than 68. We stop when there is no such element. We record the latest value of right. In our example the right value is 7 i.e., **right=7.**

Step 4:
Since left < = right we swap between a [left] and a [right]. That is between 77 and 39.
The array now looks like

45 26 39 14 **68** 61 97 77 99 90.

Step 5:
We further increment the value of left and decrement the value of right by 1 respectively i.e., **left = 2 + 1 = 3 and right = 7 - 1 = 6.**
We repeat the above steps until left < = right.
We will see that at one stage that the left and right indices will cross each other and we will find that our array has been subdivided. That all the elements lying to the left of the pivot element is less than it and all to it's right are greater. We then make recursive calls of quick sort on this two sub arrays.

b) Find the complexity of Quick sort algorithm. **[WBUT 2007, 2013]**

Answer:

Refer to Question No. 8 of Short Answer Type Questions.

3. a) Why is hashing referred as a heuristic search method? **[WBUT 2008]**

Answer:

A **heuristic algorithm**, is an algorithm that is able to produce an acceptable solution to a problem in many practical scenarios, in the fashion of a general heuristic, but for which there is no formal proof of its correctness. Alternatively, it may be correct, but may not be proven to produce an optimal solution, or to use reasonable resources. Heuristics are typically used when there is no known method to find an optimal solution, under the given constraints (of time, space *etc.*) or at all. It is a technique whereby or items are placed into a structure based on a key to-address transformation.

We use Hashing for

1) Performing optimal searches & retrieval of data at a constant time i.e. O(1)
2) It increases speed, betters ease of transfer, improves retrieval, optimizes searching of data and reduces overhead.

That's why hashing are sometimes referred as heuristic search method.

b) What is the primary advantage of hashing over deterministic search algorithms? **[WBUT 2008]**

Answer:
In deterministic search method we try to explore all the data items sequentially, one by one which gives different polynomial time complexity. But hashing is the searching method where the searching time is always O(1). Is this sense later is always advantageous.

c) Define collision. Discuss two collision resolution techniques and compare their performances. [WBUT 2008]

OR,

Discuss different collision resolution techniques. [WBUT 2015, 2017]

Answer:
A collision between two keys K & K′ occurs when both have to be stored in the table & both hash to the same address in the table.
The two collision resolution techniques are:

- **Open addressing:** It is a general collision resolution scheme for a hash table. In case of collision, other positions of the hash table are checked (a probe sequence) until an empty position is found.
  The different types of Open addressing scheme include:
  a) Linear Probing (Sequential Probing)
  b) Quadratic Probing
  c) Double Hashing (Re Hashing)
- **Chaining:** It is a collision resolution scheme to avoid collisions in a hash table by making use of an external data structure. A linked list is often used.

Linear Probing:
Refer to Question No. 2 of Short Answer Type Questions.

Chaining: In open addressing, collisions are resolved by looking for an open cell in the hash table. A different approach is to create a linked list at each index in the hash table. A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index. Other items that hashes to the same index are simply added to the linked list at that index. There is no need to search for empty cells in the primary hash table array. This is the **chaining** method.
Let us consider the following elements

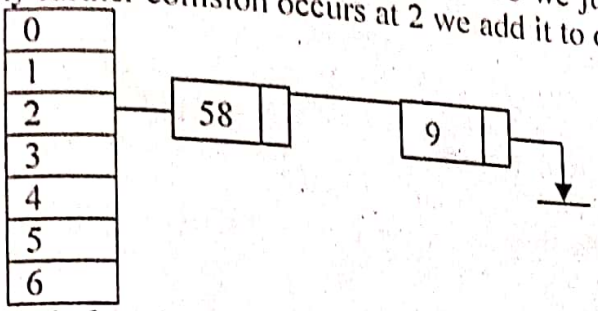
89, 18, 49, 58, 9, 7,

H(89) = 5 (Using Division – Remainder Method)
H(18) = 4 (Using Division – Remainder Method)
H(49) = 0 (Using Division – Remainder Method)
H(58) = 2 (Using Division – Remainder Method)
H (9) = 2 (Using Division – Remainder Method)

Now here, already there is one element in the position 2, which is 58 in our example. But 9 is also hashed to position 2, which is occupied by 58 in our example. When this kind of situation occurs we say that a collision has taken place.

the collision avoided by chaining method is an adjacency list representation. Whenever a collision takes place we just add to the adjacency list to the corresponding header where the collision occurred.

In our example collision occurred at header node 2. So we just add 9 and 58 to it as an adjacency list. If any further collision occurs at 2 we add it to our existing list.



A hash table uses hash functions to compute an integer value for searching a data. This integer value can then be used as an index into an array, giving us a constant time complexity to find the requested data. However, using chaining, we cannot always achieve the average time complexity of O (1) . If the hash table is very small or the hash function is not good enough then the elements can start to build in one index in the array. And hence all n elements could end up in the same linked list associated to that index. Therefore, to do a search in such a data arrangement is equivalent to looking up a data element in a linked list, something we already know to be O (n) time.

Q) Why the hash functions need to be simple? **[WBUT 2008]**

Answer:

The reason is that, the computation itself will consume less amount of time.

Q) What do you mean by hashing? What are the applications where you will prefer hash tables to other data structures? What do you mean by collision? How is it handled? **[WBUT 2011]**

Answer:

Hashing is a method for storing and retrieving records from a database. It lets you insert, delete, and search for records based on a search key value. When properly implemented, these operations can be performed in constant time. In fact, a properly tuned hash system typically looks at only one or two records for each search, insert, or delete operation. This is far better than the O(log n) average cost required to do a binary search on a sorted array of n records, or the O(log n) average cost required to do an operation on a binary search tree. However, even though hashing is based on a very simple idea, it is surprisingly difficult to implement properly. Designers need to pay careful attention to all of the details involved with implementing a hash system.

A hash system stores records in an array called a **hash table (HT)**. Hashing works by performing a computation on a search key K in a way that is intended to identify the position in HT that contains the record with key K. The function that does this calculation is called the **hash function**, and is usually denoted by the letter **h**. Since hashing schemes place records in the table in whatever order satisfies the needs of the address calculation, records are not ordered by value. A position in the hash table is also known as a **slot**. The

number of slots in hash table **HT** will be denoted by the variable M with slots numbered from 0 to M - 1.

The goal for a hashing system is to arrange things such that, for any key value K and some hash function **h**, i = **h**(K) is a slot in the table such that $0 <= i < M$, and we have the key of the record stored at **HT**[i] equal to K.

Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects), especially in interpreted programming languages like AWK, Perl, and PHP. Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually erasing the old item that is currently stored in the table and overwriting it with the new item, so every item in the table has a unique hash value.

A **collision** between two keys K & K' occurs when both have to be stored in the table & both hash to the same address in the table.

Linear probing is a used for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location. This is accomplished using two values - one as a starting value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the *stepsize*, is repeatedly added to the starting value until a free space is found, or the entire table is traversed.

The function for the rehashing is the following:

rehash(key)=(n+1) % k;

For example, we have a hash table that could accommodate 9 information, and the data to be stored were integers. To input 27, we use hash(key)= 27 % 9=0. Therefore, 27 is stored at 0. If another input 18 occurs, and we know that 18 % 9= 0, then a collision would occur. In this event, the need to rehash is needed. Using linear probing, we have the rehash(key)= (18+1) % 9= 1. Since 1 is empty, 18 can be stored in it.

5. a) What do you mean by external sorting? How does it differ from internal sorting? **[WBUT 2008, 2017, 2018]**

Answer:

External sorting is the method when the sorting takes place with the secondary memory. The time required for read and write operations are considered to be significant e.g. sorting with disks, sorting with tapes.

Internal sorting on the other hand is defined as a sorting mechanism where the sorting takes place within the main memory. The time required for read and write operations are considered to be insignificant e.g. Bubble Sort, Selection sort, Insertion sort etc.

Internal sorting is faster than external sorting because in external sorting we have to consider the external disk rotation speed, the latency time etc. Such operations are costly than sorting an array or a linked list or a hash table.

b) Write an algorith
sort technique.

Write a C function
selection sort.

Answer:

In selection sorting
The minimum valu
During each pass e
second then third ar
are sorted.

```
//n denotes the
void selecsort(
  int loc, min,
  for (i = 0; i
  {
    min = a[i];
    loc = i;
    for (j = i
    {
      if (a[j]
      {
        min =
        loc =
      }
    }
    if (loc !
    {
      temp = a
      a[i] = a
      a[loc] =
    }
  }
  printf("\nTh
  for (i = 0;
    printf("%3
}
```

Time Complexity

For first pass the i
iterates n - 2 times

$(n-1) + (n-2) + ($
$= [(n-1) * (n-1+1)$
$= [n(n-1)]/2$
$= O(n^2)$

The selection sor
consuming compar
the amount of data

b) Write an algorithm for sorting a list numbers in ascending order using selection sort technique.

OR, **[WBUT 2008, 2018]**

Write a C function for selection sort and also calculate the time complexity for selection sort. **[WBUT 2010]**

Answer:

In selection sorting we select the first element and compare it with rest of the elements. The minimum value in each comparison is swapped in the first position of the array. During each pass elements with minimum value are placed in the first position, then second then third and so on. We continue this process until all the elements of the array are sorted.

```
//n denotes the no. of elements in the array a
void selecsort(int a[], int n) {
 int loc, min, temp, i, j;
 for (i = 0; i < n; i++)
 {
  min = a[i];
  loc = i;
  for (j = i + 1; j < n; j++)
  {
    if (a[j] < min)
    {
      min = a[j];
      loc = j;
    }
  }
  if (loc ! = i)
  {
    temp = a[i];
    a[i] = a[loc];
    a[loc] = temp;
  }
 }
 printf("\nThe sorted array is:\n");
 for (i = 0; i < n; i++)
  printf("%3d", a[i]);
}
```

Time Complexity of Selection Sort

For first pass the inner for loop iterates n - 1 times. For second pass the inner for loop iterates n - 2 times and so on. Hence the time complexity of selection sort is

$(n-1)+(n-2)+(n-3)+\ldots\ldots\ldots\ldots+2+1$

$=[(n-1)*(n-1+1)]/2$

$=[n(n-1)]/2$

$=O(n^2)$

The selection sort minimizes the number of swaps. Swapping data items is time consuming compared with comparing them. This sorting technique might be useful when the amount of data is small.

6. a) In which cases, QuickSort becomes 'SlowSort'? What is the remedy in those cases?
b) Compare the performance and operation of BubbleSort and SelectionSort. [WBUT 2009]

Answer:
a) A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort becomes a "slow sort".

b) The basic idea of **Bubble Sort** is to:
1. Data elements are grouped into two sections: a sorted section and an un-sorted section.
2. Go through every element in the un-sorted section and re-arrange its position with its neighbour to put the element with higher order on the higher position. At the end, the element with the highest order will be on top of the un-sorted section, and moved to the bottom of the sorted section.
3. Repeat step 2 until no more elements left in the un-sorted section.

Performance: Best Case: $O(n)$, Worst Case: $O(n^2)$

The basic idea of **Selection** Sort:
1. Data elements are grouped into two sections: a sorted section and an un-sorted section.
2. Assuming the sorting order is from low to high, find the element with the lowest comparable order from the un-sorted section.
3. Place the found element to the end of the sorted section.
4. Repeat step 2 and 3 until no more elements left in the un-sorted section.

Performance: Best Case: $O(n^2)$, Worst Case: $O(n^2)$

7. Explain the merge sort algorithm. Why does it run faster than bubble sort in most of the cases? Show how the merge sort algorithm will sort the following array in increasing order:

100, 90, 80, 70, 60, 50, 40, 30, 20

Analyze the time complexity of the merge sort algorithm. [WBUT 2011]

Answer:
The Mergesort algorithm is based on a divide and conquer strategy. First, the sequence to be sorted is decomposed into two halves (*Divide*). Each half is sorted independently (*Conquer*). Then the two sorted halves are merged to a sorted sequence (*Combine*) as shown in Figure below.

The following procedure *mergesort* sorts a sequence *a* from index *lo* to index *hi*.

```
void mergesort(int lo, int hi)
{
    if (lo<hi)
    {
```



```
        int m=(l
        mergesor
        mergesor
        merge(lo
    }
}
```

First, index *m* in the
sequence (from *lo* to
of *mergesort*. Then
ends when *lo* = *hi*, i.e
The main work of th
merge is usually impl
an auxiliary array *b*
respective next-greate

```
void merge(int l
{
    int i, j, k;

    // copy both
    for (i=lo; i
        b[i]=a[i

    i=lo; j=m+1;
    // copy back
    while (i<=m
        if (b[i]
            a[k+
        else
            a[k+

    // copy back
    while (i<=m)
        a[k++]=b
}
```

MergeSort has the ave
and there is no worst-
as $O(n^2)$. Hence Merg
Sorting of array 100,

```
    int m=(lo+hi)/2;
    mergesort(lo, m);
    mergesort(m+1, hi);
    merge(lo, m, hi);
}
```

First, index m in the middle between lo and hi is determined. Then the first part of the sequence (from lo to m) and the second part (from $m+1$ to hi) are sorted by recursive calls of *mergesort*. Then the two sorted halves are merged by procedure *merge*. Recursion ends when $lo = hi$, i.e. when a subsequence consists of only one element.

The main work of the Mergesort algorithm is performed by function *merge*. Function *merge* is usually implemented in the following way: The two halves are first copied into an auxiliary array b. Then the two halves are scanned by pointers i and j and the respective next-greatest element at each time is copied back to array a.

```
void merge(int lo, int m, int hi)
{
  int i, j, k;

  // copy both halves of a to auxiliary array b
  for (i=lo; i<=hi; i++)
     b[i]=a[i];

  i=lo; j=m+1; k=lo;
  // copy back next-greatest element at each time
  while (i<=m && j<=hi)
     if (b[i]<=b[j])
         a[k++]=b[i++];
     else
         a[k++]=b[j++];

  // copy back remaining elements of first half (if any)
  while (i<=m)
     a[k++]=b[i++];
}
```

MergeSort has the average case time of O(nlog(n)). Moreover, Mergesort is a stable sort, and there is no worst-case scenario. Bubble sort as the average and worst case complexity as $O(n^2)$. Hence MergeSort is better than Bubble sort in most of the cases.

Sorting of array 100, 90, 80, 70, 60, 50, 40, 30, 20

Step 1: the array is divided into 2 parts recursively

100, 90;	80, 70	60, 50,	40, 30, 20	
100,90	80,70	60,50	40, 30,20	
100 90	80 70	60 50	40	30,20
100 90	80 70	60 50	40	30 20

Step 2: the arrays are now merged

100 90	80 70	60 50	40	20,30
90,100	70, 80	50, 60	20,30,40	
70,80,90,100		20,30,40,50, 60,		

20,30,40,50,60,70,80,90,100 – final sorted array

Complexity of MergeSort

If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + cn$ where c is a constant. This follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists).

$$T(n) = 2.T\left(\frac{n}{2}\right) + cn = 2.\left[2.T\left(\frac{n}{4}\right) + c.\frac{n}{2}\right] + cn = 4.T\left(\frac{n}{4}\right) + 2.cn = 4.\left[2.T\left(\frac{n}{8}\right) + c.\frac{n}{4}\right] + 2cn$$

$$= 8.T\left(\frac{n}{8}\right) + 3.cn$$

$$= \ldots$$

$$= 2^k.T\left(\frac{n}{2^k}\right) + k.cn$$

$$= 2^{\lg n}.T\left(\frac{n}{2^{\lg n}}\right) + \lg n.cn \left(2^k = n \Rightarrow k = \log_2 n\right)$$

$$= \ldots$$

$$= n.T(1) + cn.\lg(n)$$

$$= \Theta(n.\lg n)$$

8. a) Radix Sort the following list:
189, 205, 986, 421, 97, 192, 535, 839, 562, 674
Write the Radix sort algorithm. [WBUT 20
b) Find the time complexity of Binary Search Algorithm.

Answer:

a) 1st Part:

INPUT	1st pass	2nd pass	3rd pass (sorted)
189	421	205	097
205	192	421	189
986	562	535	192
421	674	939	205
097	205	562	421
192	535	674	535
535	986	986	562
839	097	189	674
562	189	192	939
674	939	097	986

2nd Part:

Suppose we have a list of n records each with a key that's a number from 1 to k (we generalize the problem a little so k is not necessarily equal to n).

We can solve this by making an array of linked lists. We move each input record into the list in the appropriate position of the array then concatenate all the lists together in order.

```
bucket sort(L)
{
list Y[k+1]
for (i = 0; i <= k; i++) Y[i] = empty
while L nonempty
{
    let X = first record in L
    move X to Y[key(X)]
}
for (i = 0; i <= k; i++)
concatenate Y[i] onto end of L
}
```

What to do when k is large? Think about the decimal representation of a number

```
x = a + 10 b + 100 c + 1000 d + ...
```

where a,b,c etc all in range 0..9. These digits are easily small enough to do bucket sort.

```
radix sort(L):
{
bucket sort by a
bucket sort by b
bucket sort by c
...
}
```

or more simply

```
radix sort(L):
{
while (some key is nonzero)
{
    bucket sort(keys mod 10)
```

```
        keys = keys / 10
    }
    }
```

In radix sorting, the last pass of bucket sorting is the one with the most effect on the overall order. So we want it to be the one using the most important digits. The previous bucket sorting passes are used only to take care of the case in which two items have the same key (mod 10) on the last pass.

b) ***The time complexity of Binary Search is as follows:***
In each iteration, the array is split into two halves. Thereby we can say that the binary search takes the form of a binary tree. The time complexity is thus $O(\log_2 n)$ in worst case also.

9. When will interpolation search be more appropriate than binary search? How does an interpolation search work? Write an algorithm for interpolation search. Show with an appropriate example that worst case time complexity of interpolation search is $O(n)$. What is the average case time complexity of interpolation search?
[WBUT 2013]

Answer:
Interpolation search works better than Binary Search for a sorted and uniformly distributed array.
Interpolation search is an algorithm for searching for a given key value in an indexed array that has been ordered by the values of the key. It parallels how humans search through a telephone book for a particular name, the key value by which the book's entries are ordered. In each search step it calculates where in the remaining search space the sought item might be, based on the key values at the bounds of the search space and the value of the sought key, usually via a linear interpolation. The key value actually found at this estimated position is then compared to the key value being sought. If it is not equal, then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position. This method will only work if calculations on the size of differences between key values are sensible.

Algorithm: INTERPOLATON SEARCH
/* Given: x, and $a_1 < ... < a_n$
Return: i such that $x = a_i$
0 if no such i exists */
i := 1
j := n
LO := a_i
HI := a_j
if x < LO **then return** 0
if x ≥ HI **then** i := j
loop invariant: x ≥ LO and x ≤ HI
while (i<j) **do**
m := floor(i + (j-i)*(x-LO) / (HI-LO))

```
MID := am
if (x > MID) then
    i := m+1
    LO := MID
else if (x < MID) then
    j := m-1
    HI := MID

else
    return MID
if (x ≠ ai) then i := 0
return i
```

On average the interpolation search makes about log(log(n)) comparisons (if the elements are uniformly distributed), where n is the number of elements to be searched. In the worst case (for instance where the numerical values of the keys increase exponentially) it can make up to O(n) comparisons (e.g., searching for 1000 in 1,2,...,999,1000,10^9 will take 1000 accesses).

10. a) Define sorting.
b) What is a stable sorting? What is In-Place sorting? **[WBUT 2014]**
c) Write the Pseudocode for Merge sort implementation. What is its time complexity?
d) If the existing array is sorted and you want to insert a new element in the list without disrupting the sortedness then which sorting technique you should use? Why?

Answer:

a) Sorting is a process by which a collection of items is placed into which is typically based on a data field called a key. Sorting refers to ordering data in an increasing or decreasing fashion according to some linear relationship among the data items.

b) ***Stable sort:***

When sorting some kind of data, only part of the data is examined when determining the sort order. The data being sorted can be represented as a record or tuple of values, and the part of the data that is used for sorting is called the *key*. A sorting algorithm is stable if whenever there are two records R and S with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list.

When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

In-place sorting:

A sort algorithm in which the sorted items occupy the same storage as the original ones. These algorithms may use $O(n)$ additional memory for bookkeeping, but at most a constant number of items are kept in auxiliary memory at any time.

c) 1st Part: ***Refer to Question No. 1(a) of Long Answer Type Questions.***

2nd Part:

Time complexity of merge sort:

Here at every step, the merge-sort considers only one array. In the next step, the algorithm splits the array into halves and then sorts and merges them. In the k^{th} iteration the algorithm splits the arrays into sub-arrays, which are $2^k - 1$ in number. In worst case the number of steps required to break the array into sub-arrays of single elements is $\log n$. At each iteration the maximum number of comparisons is O(n). So, the time complexity is $O(n \log_2 n)$ (in best, average as well as in worst case also). A drawback of mergesort is that it needs an additional space of $\Theta(n)$ for the temporary array *b*. There are different possibilities to implement function *merge*. The most efficient of these is variant b. It requires only half as much additional space, it is faster than the other variants, and it is stable.

d) Insertion Sort as its *best-case* time complexity is when the array is already sorted which is O(n).

11. What is hashing? Describe any three methods of defining a hash function. **[WBUT 2015, 2017]**

Answer:

1st part: ***Refer to Question No. 2 of Short Answer Type Questions.***

2nd part:

A hash function maps keys to small integers (buckets). An ideal hash function maps the keys to the integers in a random-like manner, so that bucket values are evenly distributed even if there are regularities in the input data.

This process can be divided into two steps:

- Map the key to an integer.
- Map the integer to a bucket.

The following functions map a single integer key (k) to a small integer bucket value h(k). m is the size of the hash table (number of buckets).

Division method (Cormen) Choose a prime that isn't close to a power of 2. h(k) = k mod m. Works badly for many types of patterns in the input data.

Knuth Variant on Division h(k) = k(k+3) mod m. Supposedly works much better than the raw division method.

Multiplication Method (Cormen). Choose m to be a power of 2. Let A be some random looking real number. Knuth suggests M = 0.5*(sqrt(5) - 1). Then do the following:

s = k*A

x = fractional part of s

h(k) = floor(m*x)

12. Write a C func
elements.
Answer:

```
#include<stdio
#include<stdli
/* structure f
struct Node
{
    int data;
    struct Nod
};
/* Function to
void insertAtT
/* Function to
void bubbleSor
/* Function to
void swap(stru
/* Function to
void printList

int main()
{
    int arr[]
    int list_s

    /* start w
    struct Nod

    /* Create
      Created
    for (i = 0
        inser

    /* print
    printf("\
    printList

    /* sort t
    bubbleSor

    /* print
    printf("\
    printList

    getchar()
    return 0;
}

/* Function t
void insertAt
```

12. Write a C function to sort positive integers that does not compose the array elements. **[WBUT 2016]**

Answer:

```
#include<stdio.h>
#include<stdlib.h>
/* structure for a node */
struct Node
{
    int data;
    struct Node *next;
};
/* Function to insert a node at the begining of a linked lsit */
void insertAtTheBegin(struct Node **start_ref, int data);
/* Function to bubble sort the given linked lsit */
void bubbleSort(struct Node *start);
/* Function to swap data of two nodes a and b*/
void swap(struct Node *a, struct Node *b);
/* Function to print nodes in a given linked list */
void printList(struct Node *start);

int main()
{
    int arr[] = {12, 56, 2, 11, 1, 90};
    int list_size, i;

    /* start with empty linked list */
    struct Node *start = NULL;

    /* Create linked list from the array arr[].
      Created linked list will be 1->11->2->56->12 */
    for (i = 0; i< 6; i++)
        insertAtTheBegin(&start, arr[i]);

    /* print list before sorting */
    printf("\n Linked list before sorting ");
    printList(start);

    /* sort the linked list */
    bubbleSort(start);

    /* print list after sorting */
    printf("\n Linked list after sorting ");
    printList(start);

    getchar();
    return 0;
}
/* Function to insert a node at the begining of a linked lsit */
Void insertAtTheBegin(struct Node **start_ref, int data)
```

```
{
    struct Node *ptr1 = (struct Node*)malloc(sizeof(struct Node));
    ptr1->data = data;
    ptr1->next = *start_ref;
    *start_ref = ptr1;
}
/* Function to print nodes in a given linked list */
void printList(struct Node *start)
{
    struct Node *temp = start;
    printf("\n");
    while (temp!=NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
/* Bubble sort the given linked lsit */
void bubbleSort(struct Node *start)
{
    int swapped, i;
    struct Node *ptr1;
    struct Node *lptr = NULL;

    /* Checking for empty list */
    if (ptr1 == NULL)
        return;

    do
    {
        swapped = 0;
        ptr1 = start;

        while (ptr1->next != lptr)
        {
            if (ptr1->data > ptr1->next->data)
            {
                swap(ptr1, ptr1->next);
                swapped = 1;
            }
            ptr1 = ptr1->next;
        }
        lptr = ptr1;
    }
    while (swapped);
}
/* function to swap data of two nodes a and b*/
void swap(struct Node *a, struct Node *b)
{
```



```
    int te
    a->dat
    b->dat
}
```

13. a) Write
b) Give an o
Answer:
a)

```
Linear Sea
Step 1: Se
Step 2: if
Step 3: if
Step 4: Se
Step 5: Go
Step 6: Pr
Step 7: Pr
Step 8: Ex
```

b) Linear sea
Obviously, t
array. In this
Likewise, th
equal to the
made.

14. Find the
assuming th
Answer:
Refer to Qu

15. Write sh
a) Radix so
b) Merge So
c) Interpolat
d) Heap
Answer:
a) Radix sor
Radix sorting
successively
copying the
of the digit
sorted.
The followin
steps involve
pass 1 the or

```
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}
```

13. a) Write an algorithm for linear search.
b) Give an outline of the complexity of your algorithm.

Answer:

a)

```
Linear Search (Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit
```

b) Linear search executes in O(n) time where n is the number of elements in the array. Obviously, the best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.
Likewise, the worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made.

14. Find the time complexity of merge sort technique using the recurrence relation assuming the size of the list $n = 2^k$. **[WBUT 2017]**

Answer:

Refer to Question No. 7 of Long Answer Type Questions.

15. Write short notes on the following: **[WBUT 2010, 2014]**
a) Radix sort **[WBUT 2012]**
b) Merge Sort **[WBUT 2014]**
c) Interpolation search **[WBUT 2015, 2018]**
d) Heap

Answer:

a) Radix sort:

Radix sorting is a technique for ordering a list of positive integer values. The values are successively ordered on digit positions, from right to left. This is accomplished by copying the values into "buckets," where the index for the bucket is given by the position of the digit being sorted. Once all digit positions have been examined, the list must be sorted.

The following table shows the sequences of values found in each bucket during the four steps involved in sorting the list 624 852 426 987 269 146 415 301 730 78 593. During pass 1 the ones place digits are ordered. During pass 2 the tens place digits are ordered,

etaining the relative positions of values set by the earlier pass. On pass 3 the hundreds place digits are ordered, again retaining the previous relative ordering. After three passes the result is an ordered list.

bucket	pass 1	pass 2	pass 3
0	73[0]	3[0]1	[0]78
1	30[1]	4[1]5	[1]46
2	85[2]	6[2]4, 4[2]6	[2]69
3	59[3]	7[3]0	[3]01
4	62[4]	1[4]6	[4]15,[4]26
5	41[5]	8[5]2	[5]93
6	42[6], 14[6]	2[6]9	[6]24
7	98[7]	[7]8	[7]30
8	7[8]	9[8]7	[8]52
9	26[9]	5[9]3	[9]87

The C-program of the radix-sort is as shown below:

```
#include <stdio.h>
#include <conio.h>
#define N 11
#define SHOWPASS

void radixsort(int[], int);
void print(int[], int);

int main(void)
{
  int unsortedArr[N] ={624, 852, 426, 987, 269, 146, 415, 301,
  730, 78, 593};
  int i;
  printf("\nBefore Sorting:\n\n");
  for (i = 0; i < N; i++)
    printf("%5d", unsortedArr[i]);
  radixsort(unsortedArr, N);
  printf("\n\nAfter Sorting:\n");
  for (i = 0; i < N; i++)
    printf("%5d", unsortedArr[i]);
  getch();
  return 0;
}
void radixsort(int a[], int n)
{
  int i, b[N], m = 0, exp = 1;
  int count = 0;
  for (i = 0; i < n; i++)
  {
    if (a[i] > m)
      m = a[i];
  }
  while (m / exp > 0)
```

```
{
 count++;
 int bucket[10] ={0};
 for (i = 0; i < n; i++)
   bucket[a[i] / exp % 10]++;
 for (i = 1; i < 10; i++)
   bucket[i] += bucket[i - 1];
 for (i = n - 1; i >= 0; i--)
   b[--bucket[a[i] / exp % 10]] = a[i];
 for (i = 0; i < n; i++)
   a[i] = b[i];
 exp *= 10;

 #ifdef SHOWPASS
   printf("\nPASS %d : ", count);
   print(a, n);
 #endif

}
}
void print(int a[], int n)
{
 int i;
 printf("\n");
 for (i = 0; i < n; i++)
   printf("%5d", a[i]);
}

/* Output of the above program

Before Sorting:
 624 852 426 987 269 146 415 301 730  78 593
PASS 1 :
 730 301 852 593 624 415 426 146 987  78 269
PASS 2 :
 301 415 624 426 730 146 852 269  78 987 593
PASS 3 :
  78 146 269 301 415 426 593 624 730 852 987

After Sorting:
  78 146 269 301 415 426 593 624 730 852 987
*/
```

Time complexity of radix sort:

The time complexity of the algorithm is as follows: Suppose that the n input numbers have maximum k digits. Then the Counting Sort procedure is called a total of k times. Counting Sort is a linear, or O(n) algorithm. So the entire Radix Sort procedure takes O(n) time.

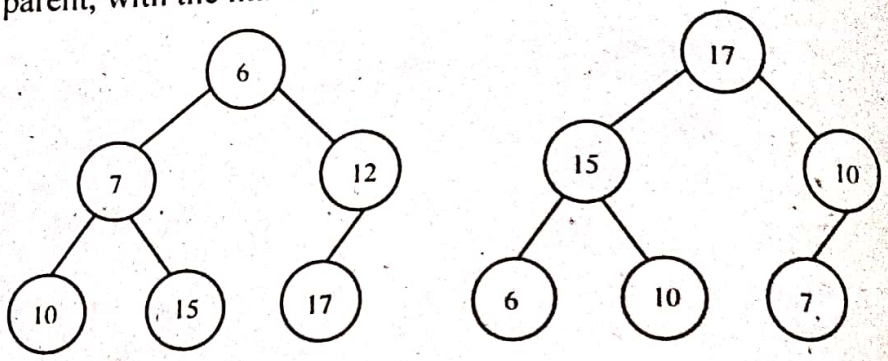
b) Merge Sort: *Refer to Question No. 1 of Long Answer Type Questions.*

c) Interpolation search: *Refer to Question No. 9 of Long Answer Type Questions.*

d) Heap Sort:

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

- the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.



In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is not a sorted structure and can be regarded as partially ordered. As it can be seen from the picture, there is no particular relationship among nodes on any given level, even among the siblings.

Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has O(log N) height.

A heap is useful data structure when we need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

1. a) Describe
b) What is dif
Answer:
a) The C code

```
void revers
    char *
    char t

    if (st
        wh

        }
        --
        wh

        }
    }
}
```

In the above c
by str and e
end is decrem

b) *The differe*
1. Union alloc
of the union b
members.
2. In union, o
each member
While structur
memory, a uni
variables. That
of one type or
occasion. Ther
or group of by
the answer.