Q.1 a. Explain different data types used in 'C' programming Language.

Answer:

a. There are only a few basic data types in C: (4) Marks

char : a single byte, capable of holding one character in the local character set
int : an integer, typically reflecting the natural size of integers on the host machine
float : single-precision floating point
double: double-precision floating point

In addition, there are a number of qualifiers that can be applied to these basic types. Short and long apply to integers. The intent is that short and long should provide different lengths of integers where practical;

int will normally be the natural size for a particular machine. short is often 16 bits long, and int either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its own hardware, subject only to the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int, which is no longer than long.

The qualifier signed or unsigned may be applied to char or any integer. unsigned numbers are always positive or zero, and obey the laws of arithmetic modulo 2n, where n is the number of bits in the type. So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255, while signed chars have values between -128 and 127 (in a two's complement machine.) Whether plain chars are signed or unsigned is machine-dependent, but printable characters are always positive.

The type long double specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; float, double and long double could represent one, two or three distinct sizes.

b. Write a C function to traverse a circular linked list.

Answer:

a. /* Function to traverse a given Circular linked list and print nodes */ 4 marks

```
void printList(struct node *first)
{
   struct node *temp = first;

  // If linked list is not empty
 if (first != NULL)
{
   // Keep printing nodes till we reach the first node again
   do
   {
  printf("%d ", temp->data);
  temp = temp->next;
   }
 while (temp != first);
}
}
```

c. Write the algorithm for selection sort.

Answer:

Selection sort works as follows: first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.
(4 marks)

```
SELECTION_SORT (A)

for i ← 1 to n-1 do

   min j ← i;

   min x ← A[i]

   for j ← i + 1 to n do

      If A[j] < min x then

         min j ← j

         min x ← A[j]

   A[min j] ← A [i]

   A[i] ← min x
```

d. Write the recursive algorithms for in-order and post-order binary tree traversal.

Answer:

Algorithm for post-order traversal (2 + 2) Marks

Algorithm In-order (tree)

1. Traverse the left sub-tree, i.e., call In-order(left-sub-tree)
2. Visit the root.
3. Traverse the right sub-tree, i.e., call In-order(right-sub-tree)

Algorithm for in-order traversal

Algorithm Post-order (tree)

1. Traverse the left sub-tree, i.e., call Post-order(left-sub-tree)
2. Traverse the right sub-tree, i.e., call Post-order(right-sub-tree)
3. Visit the root.

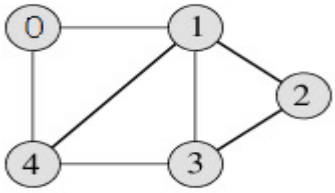
e. Explain the adjacency matrix representation for graph data structure.

Answer:

Adjacency Matrix: **(4 Marks)**

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w**.**

For a given graph



Adjacency matrix is

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

f. Define Multi way search tree.

Answer:

Multi-way search tree (4) Marks

A multi-way tree is a tree that can have more than two children. A multi-way tree of order m (or an m-way tree) is one in which a tree can have m children

A m-way search tree is a tree in which

1. The nodes hold between 1 to m-1 distinct keys

2. The keys in each node are sorted

3. A node with k values has k+1 sub-trees, where the sub-trees may be empty.

4. The i'th sub-tree of a node $[v_1, ..., v_k]$, $0 < i < k$, may hold only values v in the range $v_i < v < v_{i+1}$ (v_0 is assumed to equal - infinity , and v_{k+1} is assumed to equal infinity).

g. Convert the expression A*B+C/D–F to equivalent Prefix and Postfix notations.(7×4)

Answer:

Given expression (A*B+C/D-F) (2+ 2) Marks

Conversion to prefix Notation:

(*A B) + C/D – F

(*A B)+(/ C D) – F

(+*A B/C D) – F

- +*A B/C D F (PREFIX NOTATION)

Conversion to postfix Notation:

(A B *) + C / D – F

(A B*) + (C D /) – F

(A B * C D / +) – F

A B * C D / +F – (POSTFIX NOTATION)

Q.2 a. Write a C program to find the sum of first n natural numbers where n is entered by user. (5)

Answer:

#include <stdio.h> (5) Marks

```
int main(){
   int n, count, sum=0;
   printf("Enter the value of n.\n");
   scanf("%d", &n);
   for(count=1;count<=n;++count)  //for loop terminates if count>n
   {
     sum+=count;    /* this statement is equivalent to sum=sum+count */
   }
   printf("Sum=%d", sum);
   return 0;
```

b. What is pointer variable? Write a C program to copy one string to another, using pointers and without using library functions. (5)

Answer:

What is pointer variable **(1 + 4) marks**

Pointer variable or simply pointer are the special types of variables that holds memory address of other variables rather than data, that is, a variable that holds address value is called a pointer variable or simply a pointer.

Declaration of Pointer

Dereference operator (*) are used for defining pointer variable

data_type* pointer_variable_name;

int* p;

program to copy one string to another using pointers

```
/* strcpy: copy t to s; pointer version */

void strcpy(char *s, char *t)

{

int i;

i = 0;

while ((*s = *t) != '\0') {

s++;

t++;

}

}
```

c. Define structure. How structures are declared? Discuss with the help of program how the members of a structure can be accessed? (8)

Answer:

Structure definition

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called ``records'' in some languages, notably Pascal.) Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities

Defining & Declaration of structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program. The format of the struct statement is this:

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

For example --

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Or

```
struct date
{
    int date;
    char month[20];
    int year;
};

struct date today;
```

Program for accessing of structure members

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */
```

```
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = any value in the range of integer.

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = with in the range of integer.

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

Q.3 a. Write an algorithm or C function to insert a new node at the end of singly linked list. (5)

Answer:

```
void insert_at_end()
{
struct node *new_node,*temp;, *start = NULL

new_node=(struct node *)malloc(sizeof(struct node));

if(new_node == NULL)
 printf("nFailed to Allocate Memory");
else
 printf("nEnter the data : ");
 scanf("%d",&new_node->data);
 new_node->next=NULL;
 if(start==NULL)
 {
  start=new_node;
   }
```

```
else
{
  temp = start;
    while(temp->next!=NULL)
    {
    temp = temp->next;
    }
  temp->next = new_node;
}
}
```

b. How the sparse matrix is represented in memory? (5)

Answer:

SPARSE MATRIX (5)

If a large number of elements of the matrix are zero elements, then it is called a sparse matrix.

Representation in Memory

Representing a sparse matrix by using a two-dimensional array leads to the wastage of a substantial amount of space. Therefore, an alternative representation must be used for sparse matrices. One such representation is to store only non- zero elements along with their row positions and column positions. That means representing every non-zero element by using triples (i, j, value), where i is a row position and j is a column position, and store these triples in a linear list. It is possible to arrange these triples in the increasing order of row indices, and for the same row index in the increasing order of column indices. Each triple (i,j,value) can be represented by using a node having four fields as shown in the following:

```
Struct snode
{ Int row,col,val;
 Struct snode *next; };
```

0	val	r	c
1			
2			
3			

Using array

c. Suppose a two dimensional array A is declared as A (1: 5, 1: 4). Assume the base address to be 500 and that each element requires 2 words of storage. Calculate the address of A[4,3] if the array is stored in
(i) Row Major Order (ii) Column major order (4+4)

Answer:

Given Base (A) = 500, w =2, lbr =1, lbc =1 , ubr =5, ubc =4 (4 + 4) Marks

Number of rows m = (ubr-lbr+1) = 5 -1+1 =5
Number of columns n = (ubc-lbc+1) = 4-1+1 =4

(i) Row major Order

Add(A[i][j]) = base (A) + w [n (i – lbr) + (j –lbc)]

Add (A[4][3] = 500 + 2[4 (4 -1) + (3 -1)

Add (A[4][3] = 500 + 2 [4 *3 + 2]

Add (A[4][3] = 500 + 2 [14]

Add (A[4][3] = 500 + 28

Add (A[4][3] = 528

(i) Column major Order

Add(A[i][j]) = base (A) + w [m (j – lbc) + (i –lbr)]

Add (A[4][3] = 500 + 2[5 (3 – 1) + (4 – 1)

Add (A[4][3] = 500 + 2[5 * 2 + 3]

Add (A[4][3] = 500 + 2[10 + 3]

Add (A[4][3] = 500 + 2[13]

Add (A[4][3] = 500 + 26

Add (A[4][3] = 526

Q.4 a. What do you understand by STACK? Write a C program for array implementation of STACK data structure. (9)

Answer:

Stack Definition (2 + 7)

Marks

A stack is an abstract data type that follows Last in First Out principle (LIFO). Last element added to the stack must be the first element to be removed.

Stack supports two main operations:
push() - inserts new element to the top of stack
pop() - Remove the top element from the stack

C Program for Array implementation of Stack

```
#include <stdio.h>
 #include <stdlib.h>

  int max = 100;          *For array implementation, stack needs to be an array.
  int stack[max];
  int stackSize, top = -1;
```

```
/* push element to the top of stack */
void push(int data) {
    if (top >= stackSize - 1) {
        printf("Stack overflow\n");
        return;
    }
    stack[++top] = data;
    printf("Data added to stack:%d\n", stack[top]);
}

/* check whether stack is empty or not */
int isEmpty() {
    if (top == -1)
        return 1;
    else
        return 0;
}

/* pop/remove top element from stack */
void pop() {
    if (isEmpty()) {
        printf("Stack underflow\n");
        return;
    }
    printf("Data poped from stack:%d\n", stack[top]);
    top = top - 1;
}

/* display elements in stack */
void display() {
    int i = 0;
    if (isEmpty()) {
        printf("No data present in stack\n");
        return;
    }

    for (i = top; i >= 0; i--) {
        printf("%d\n", stack[i]);
    }
}

int main () {
    int size, data, ch;
    printf("Enter the size of the stack:");
    scanf("%d", &size);
    stackSize = size;
    stack = (int *)malloc(sizeof (int) *size);
```

```
    while (1) {
        printf("1. push\n2. Pop\n3. IsEmpty\n");
        printf("4. Display stack objects\n");
        printf("5. Object count\n6. Exit\n");
        printf("Enter ur option:");
        scanf("%d", &ch);
        switch (ch){
            case 1:
                printf("Enter the ur data:");
                scanf("%d", &data);
                push(data);
                break;
            case 2:
                pop();
                break;
            case 3:
                if (isEmpty())
                    printf("Stack is Empty\n");
                else
                    printf("Stack is not Empty\n");
                break;
            case 4:
                display();
                break;
            case 5:
                if (isEmpty())
                    printf("No objects in stack\n");
                else
                    printf("Object count:%d\n", top+1);
                break;
            case 6:
                exit(0);
            default:
                printf("U have entered wrong option\n");
                break;
        }

    }
    return 0;
}
```

b. What do you mean by Queue data structure? Write the algorithms or C function for inserting & deleting an element in linked list based implementation of queue data structure. (9)

Answer:

Queue data structure:

Queue is an abstract data type or a linear data structure, in which the elements are inserted from one end called REAR (also called tail), and the deletion of existing element takes place from the

other end called as FRONT (also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first

C function for inserting and deleting an element in linked list based queue

```
struct node
{
   int info;
   struct node *ptr;
}*front,*rear,*temp,*front1;

void enq (int data) // insertion in queue
{
   if (rear == NULL)
   {
      rear = (struct node *)malloc(1*sizeof(struct node));
      rear->ptr = NULL;
      rear->info = data;
      front = rear;
   }
   else
   {
      temp=(struct node *)malloc(1*sizeof(struct node));
      rear->ptr = temp;
      temp->info = data;
      temp->ptr = NULL;

      rear = temp;
   }
   count++;
}

/* Deletion from the queue */
void deq()
{
   front1 = front;

   if (front1 == NULL)
   {
      printf("\n Error: Trying to display elements from empty queue");
      return;
   }
   else
      if (front1->ptr != NULL)
      {
         front1 = front1->ptr;
         printf("\n Dequed value : %d", front->info);
         free(front);
```

```
        front = front1;
    }
    else
    {
        printf("\n Dequed value : %d", front->info);
        free(front);
        front = NULL;
        rear = NULL;
    }
    count--;
}
```

Q.5 a. What is threaded binary tree? Write a function in C that traverses a threaded binary tree in in-order. (9)

Answer:

Threaded Binary Tree

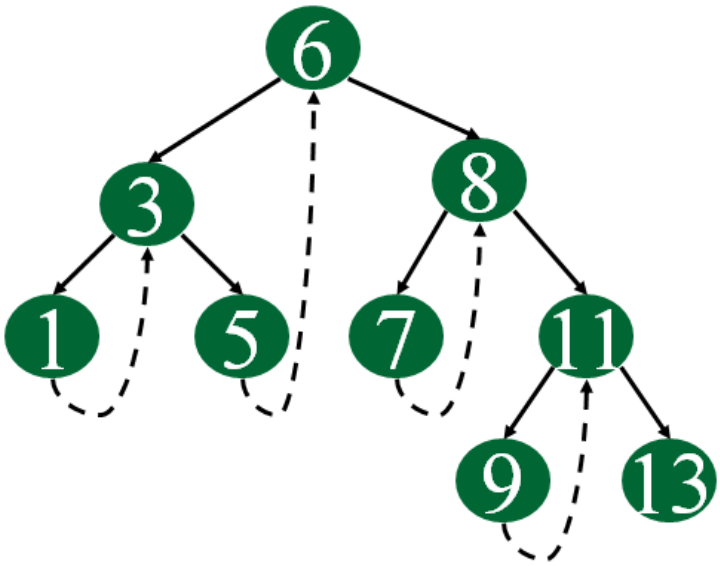
In-order traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make in-order traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the in-order successor of the node (if it exists).

There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the in-order successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to in-order predecessor and in-order successor respectively. The predecessor threads are useful for reverse in-order traversal and post-order traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



C representation of a Threaded Node
Following is C representation of a single threaded node.

```
struct Node
{
   int data;
   Node *left, *right;
   bool rightThread;
}
```

Since right pointer is used for two purposes, the Boolean variable rightThread is used to indicate whether right pointer points to right child or in-order successor. Similarly, we can add leftThread for a double threaded binary tree.

In-order Traversal using Threads
.

```
// Utility function to find leftmost node in atree rooted with n
struct Node* leftMost(struct Node *n)
{
   if (n == NULL)
     return NULL;

   while (n->left != NULL)
      n = n->left;
    return n;
}
 // C code to do inorder traversal in a threadded binary tree
void inOrder(struct Node *root)
{
   struct Node *cur = leftmost(root);
   while (cur != NULL)
   {
      printf("%d ", cur->data);

      // If this node is a thread node, then go to
      // inorder successor
      if (cur->rightThread)
         cur = cur->right;
      else // Else go to the leftmost child in right subtree
         cur = leftmost(cur->right);
   }
}
```

b. Define Binary search tree. Write an algorithm to insert a new element in a binary search tree. (9)

Answer:
Binary Search Tree

A binary search tree (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub-tree and smaller than the keys in all nodes in that node's right sub-tree.

Algorithm/ C function to insert a new element in a binary search tree

```
struct tnode {
int data;
struct tnode *left;
struct tnode *right;
};

struct tnode *insert(struct tnode *p , int val){
      if(p==NULL){
            p = (struct tnode *)malloc(sizeof(struct tnode));
            p->data = val;
            p->left = NULL;
            p->right = NULL;
      }else{
            struct tnode *q,*temp;
            q=p;
            temp=p;
            while(q != NULL ){
                  temp= q;
                  if( val >= temp->data)
                        q=temp->right;
                  else
                        q=temp->left;
            }
            struct tnode *n;
            n= (struct tnode *)malloc (sizeof(struct tnode));
            n->data =val;
            if( val >=temp->data )
                  temp->right =n;
            else
                  temp->left = n;
      }
      return p;
}
```

Q.6 a. Write a C function to implement Merge sort. Also compute the time complexity of merge sort. (9)

Answer:

C function to implement merge sort

```
void merge-sort (int list[],int low,int high)
{
```

```
    int mid;

    if(low < high)
    {
        mid = (low + high) / 2;
        merge-sort(list, low, mid);
        merge-sort (list, mid + 1, high);
        merge(list, low, mid, high);
    }
}

void merge(int list[],int low,int mid,int high)
{
    int i, mi, k, lo, temp[50];

    lo = low;
    i = low;
    mi = mid + 1;
    while ((lo <= mid) && (mi <= high))
    {
        if (list[lo] <= list[mi])
        {
            temp[i] = list[lo];
            lo++;
        }
        else
        {
            temp[i] = list[mi];
            mi++;
        }
        i++;
    }
    if (lo > mid)
    {
        for (k = mi; k <= high; k++)
        {
            temp[i] = list[k];
            i++;
        }
    }
    else
    {
        for (k = lo; k <= mid; k++)
        {
             temp[i] = list[k];
            i++;
        }
    }
```

```
   for (k = low; k <= high; k++)
   {
      list[k] = temp[k];
   }
}
```

Complexity of Merge sort

Assumption: N is a power of two.

For N = 1: time is a constant (denoted by 1)

Otherwise: time to mergesort N elements = time to mergesort N/2 elements plus time to merge two arrays each N/2 elements.

Time to merge two arrays each N/2 elements is linear, i.e. N

Thus we have:

(1) T(1) = 1

(2) T(N) = 2T(N/2) + N

Next we will solve this recurrence relation. First we divide (2) by N:

(3) T(N) / N = T(N/2) / (N/2) + 1

N is a power of two, so we can write

(4) T(N/2) / (N/2) = T(N/4) / (N/4) +1

(5) T(N/4) / (N/4) = T(N/8) / (N/8) +1
(6) T(N/8) / (N/8) = T(N/16) / (N/16) +1
(7) ……
(8) T(2) / 2 = T(1) / 1 + 1
Now we add equations (3) through (8) : the sum of their left-hand sides will be equal to the sum of their right-hand sides:

T(N) / N + T(N/2) / (N/2) + T(N/4) / (N/4) + … + T(2)/2 =

T(N/2) / (N/2) + T(N/4) / (N/4) + ….+ T(2) / 2 + T(1) / 1 + LogN

(LogN is the sum of 1s in the right-hand sides)

After crossing the equal term, we get

(9) T(N)/N = T(1)/1 + LogN

T(1) is 1, hence we obtain

(10) T(N) = N + NlogN = O(NlogN)

Hence the complexity of the MergeSort algorithm is O(NlogN).

b. What is minimum spanning tree? Write & explain Kruskal algorithm for finding minimum spanning tree. (9)

Answer:

Minimum spanning Tree

A spanning tree of a graph is just a sub-graph that contains all the vertices and is a tree. A minimum spanning tree (MST) of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.

Kruskal algorithm for finding minimum spanning tree

STEPS

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree**.**

Data Structure/functions used in algorithm

Make_SET(v): Create a new set whose only member is pointed to by v. Note that for this operation v must already be in a set.

FIND_SET(v): Returns a pointer to the set containing v.

UNION(u, v): Unites the dynamic sets that contain u and v into a new set that is union of these two sets.

Algorithm

Start with an empty set A, and select at every stage the shortest edge that has not been chosen or rejected, regardless of where this edge is situated in the graph.

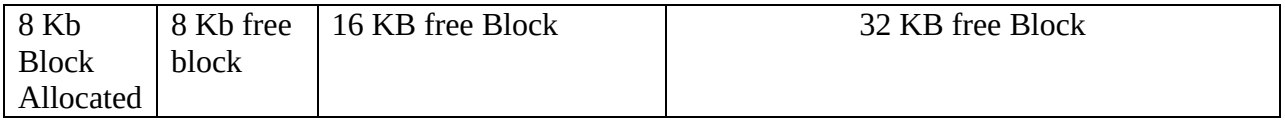
KRUSKAL(V, E, w)

```
A ← { }          ▷ Set A will ultimately contains the edges of the MST
for each vertex v in V
   do MAKE-SET(v)
sort E into non-decreasing order by weight w
for each (u, v) taken from the sorted list
   do if FIND-SET(u) = FIND-SET(v)
```

then A ← A ∪ {(u, v)}
UNION(u, v)
return A

Q.7 a. Explain the Buddy system approach used for dynamic memory management. (9)
Answer:
In a buddy system, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence, such that any block except the smallest can be divided into two smaller blocks of permitted sizes.
When the allocator receives a request for memory, it rounds the requested size up to a permitted size, and returns the first block from that size's free list. If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.
When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size (coalescence). To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the "buddy" of any free block can be calculated from its address

A binary buddy heap before allocation.



A binary buddy heap after allocating a 8 KB block.

A binary buddy heap after allocating a 10 KB block; note the 6 KB wasted because of rounding up.

For example, an allocator in a binary buddy system might have sizes of 16, 32, 64, …, 64 kB. It might start off with a single block of 64 kB. If the application requests a block of 8 kB, the allocator would check its 8 kB free list and find no free blocks of that size. It would then split the 64 kB block into two block of 32 kB, split one of them into two blocks of 16 kB, and split one of them into two blocks of 8 kB. The allocator would then return one of the 8 kB blocks to the application and keep the remaining three blocks of 8 kB, 16 kB, and 32 kB on the appropriate free lists. If the application then requested a block of 10 kB, the allocator would round this request up to 16 kB, and return the 16 kB block from its free list, wasting 6 kB in the process.

A Fibonacci buddy system might use block sizes 16, 32, 48, 80, 128, 208, … bytes, such that each size is the sum of the two preceding sizes. When splitting a block from one free list, the two parts get added to the two preceding free lists.
A buddy system can work very well or very badly, depending on how the chosen sizes interact with typical requests for memory and what the pattern of returned blocks is. The rounding typically leads to a significant amount of wasted memory, which is called internal fragmentation. This can be reduced by making the permitted block sizes closer together.

b. Define B – tree. Explain the procedure of inserting a new element in B Tree. (9)

Answer:

An extension of a multi-way search tree of order m is a B-tree of order m. This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node

.

A B-tree of order m is a multi-way search tree in which:

1. The root has at least two sub-trees unless it is the only node in the tree.
2. Each non-root and each non-leaf node have at most m nonempty children and at least m/2 nonempty children.
3. The number of keys in each non-root and each non-leaf node is one less than the number of its nonempty children.
4. All leaves are on the same level.

Insertion into a B-tree

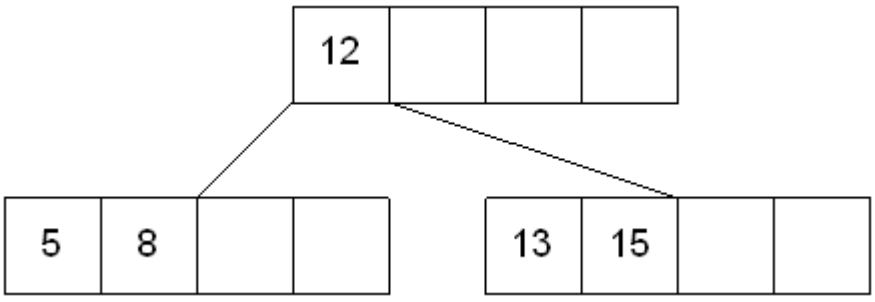
The condition that all leaves must be on the same level forces a characteristic behaviour of B-trees, namely that B-tree are not allowed to grow at their leaves; instead they are forced to grow at the root.

When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:

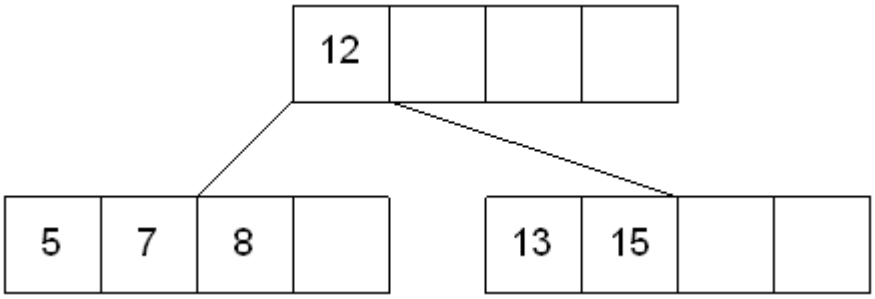
A key is placed into a leaf that still has room.
The leaf in which a key is to be placed is full.
The root of the B-tree is full.

Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



Inserting the number 7 results in:

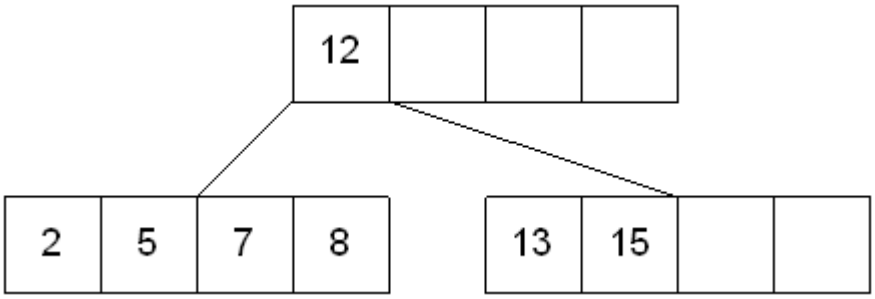


Case 2: The leaf in which a key is to be placed is full

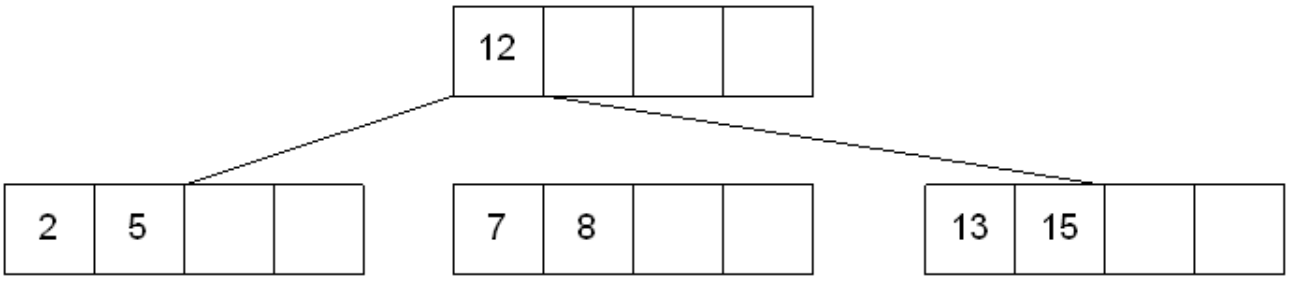
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

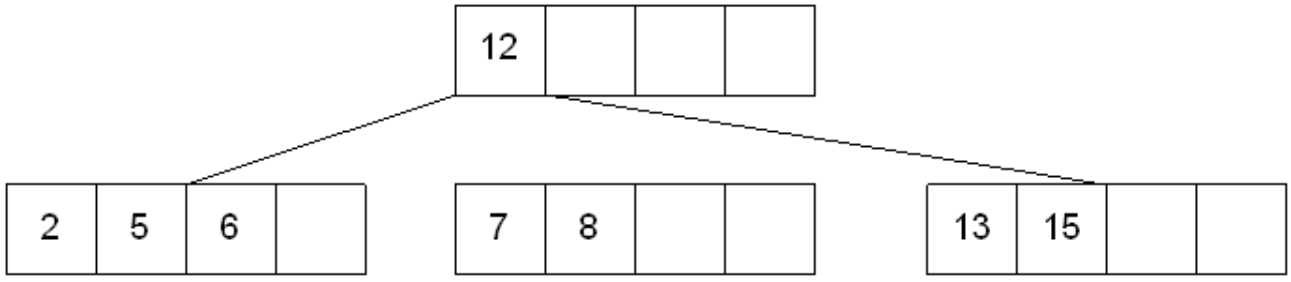
The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process is continues up the tree until all of the values have "found" a location.

Insert 6 into the following B-tree:

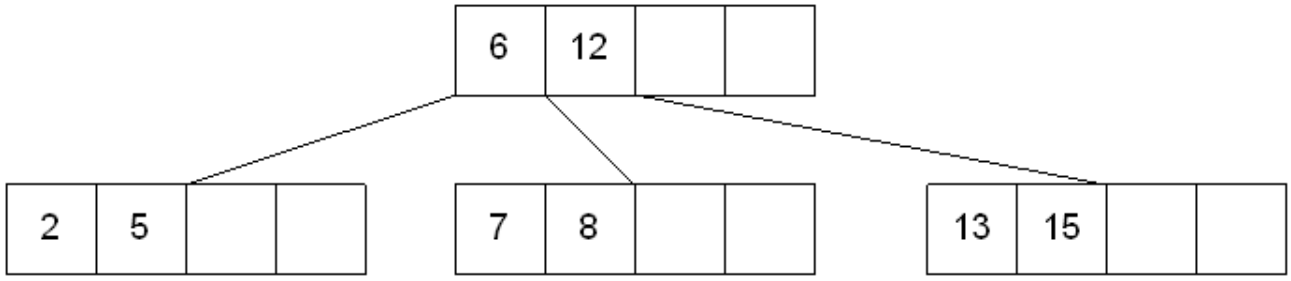


results in a split of the first leaf node:





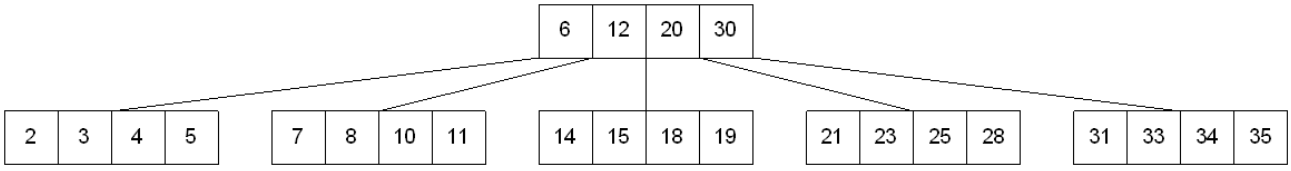
The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:



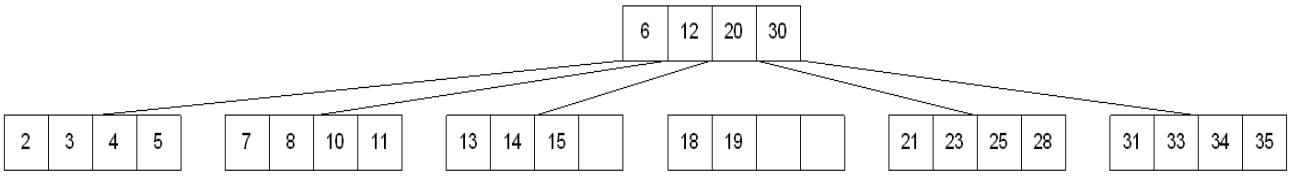
Case 3: The root of the B-tree is full

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

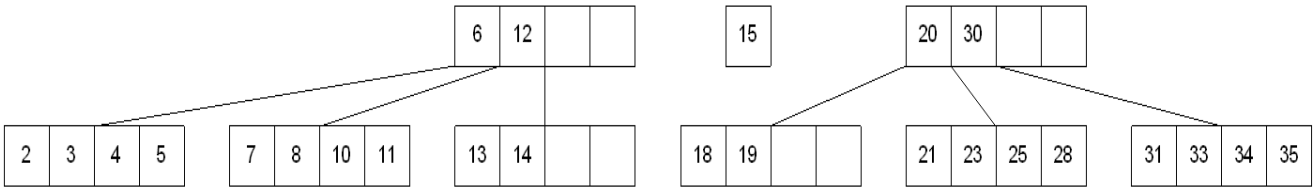
Inserting 13 into the following tree:



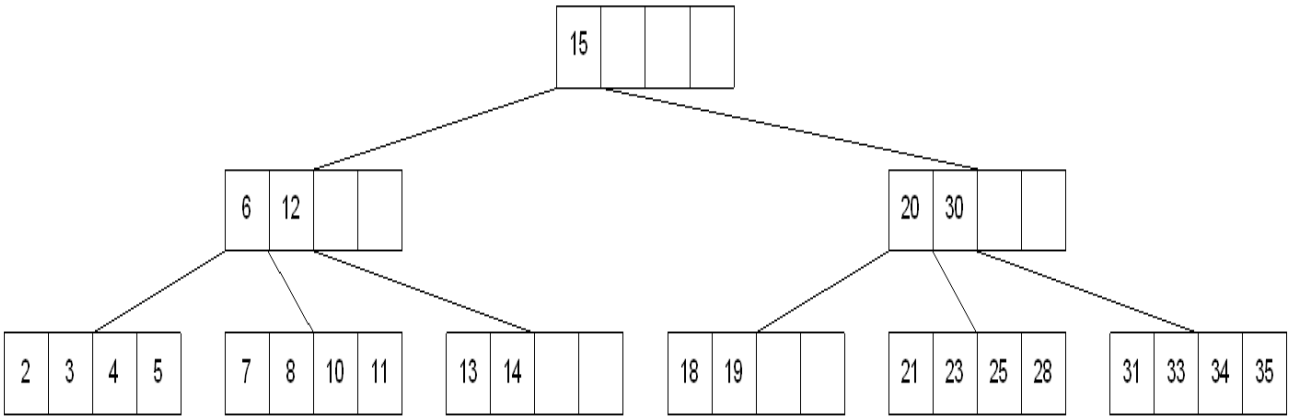
Results in:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:



TEXT BOOK

I. Kernighan and Ritchie, "The C Programming Language", Prentice Hall of India, 1989
II. Tenenbaum, Augestein and Langsam "Data Structures using C", Prentice Hall, 1992
III. Horowitz and Sahani, "Fundamentals of Data Structures ", Galgotia Book Source (GBS) Publication, reprint, 1994