Model Solution for QP CODE : 24788 (3 Hours)

All answers are for reference only. Any alternate answer that solves the problem should be considered eqully valid.

1. (a) Define data structure ? Give its classification.

Data structure is a speicalized format for organizing and storing of the data. Data structure is broadly classified into linear and non-linear data structure as follows:

Linear data structures- the data elements are organized in some sequence is called linear data structure. Here the various operations on a data structure are possible only in a sequence i.e. we cannot insert the element into any location of our choice. Examples of linear data structures are array, stacks, queue, and linked list.

Non-Linear data structures -When the data elements are organized in some arbitrary function without any sequence, such data structures are called non-linear data structures. Examples of such type are trees, graphs.

(b) What are the advantages of using dynamic memory allocation over static memory allocation ?

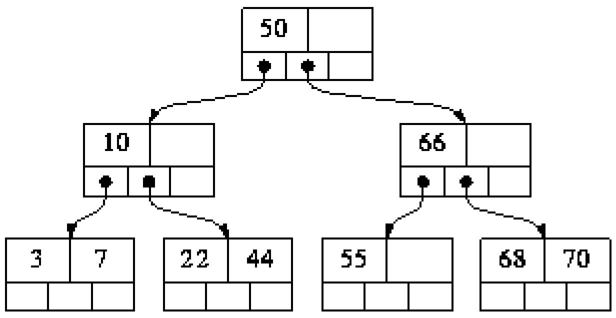
There exist multiple advantages of dynamic memory allocation mainly focusing on
i) better utilization of memory space
ii) no upper limit on the size of data structure

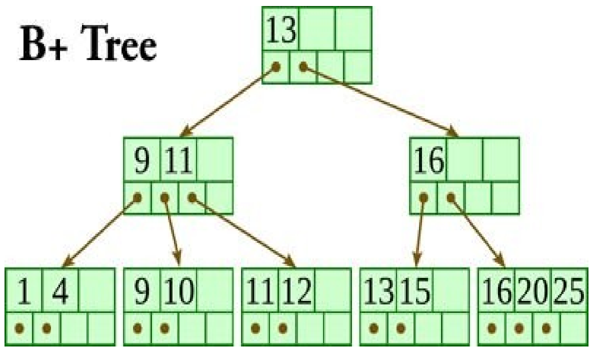
(c) Describe Multiway Search Tree with an example.

A tree where node can have more than two child nodes is called multiway search trees. B-tree and B+ trees are examples of Multiway Search Trees.

A **B-tree** is a generalized binary search tree, in which any node can have more than two children. Each internal node in a B-tree contains a number of keys. These keys separate the values, and further forms the sub-trees.

A **B+ tree** is an n-array tree with a node, which consists of a large number of children per node. The root may be a leaf or a node that contains more than two children. A B+ tree consists of a root, internal nodes and leaves.





(d) Write a function in C to implement Shell Sort.

```
* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
// Start with a big gap, then reduce the gap
(int gap = n/2; gap > 0; gap /= 2)
{
// Do a gapped insertion sort for this gap size.
// The first gap elements a[0..gap-1] are already in gapped order
// keep adding one more element until the entire array is
// gap sorted
(int i = gap; i < n; i += 1)
{
// add a[i] to the elements that have been gap sorted
// save a[i] in temp and make a hole at position i
temp = arr[i];

// shift earlier gap-sorted elements up until the correct
// location for a[i] is found
j;
(j = i; j >= gap && arr[j - gap] > temp; j -= gap)
[j] = arr[j - gap];

//put temp (the original a[i]) in its correct location
[j] = temp;
}
}
return 0;
}
```

2. (a) Discuss file I/O operations in C programming language.

Major file I/O operations are as follows:

1. Creation of a new file (fopen with attributes as “a” or “a+” or “w” or “w++”)
2. Opening an existing file (fopen)
3. Reading from file (fscanf or fgetc)
4. Writing to a file (fprintf or fputs)
5. Moving to a specific location in a file (fseek, rewind)
6. Closing a file (fclose)

for opening the file, various access modes are as follows:

- “r” – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.

•“w” – Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.

•“a” – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

•"r+" – Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.

•"w+" – Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open file.

•"a+" – Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

(b) Write C program to perform polynomial addition using Linked List.

```
#include<stdio.h>
// Node structure containing power and coefficient of variable
struct Node
{
	int coeff;
	int pow;
	struct Node *next;
};

// Function to create new node
void create_node(int x, int y, struct Node **temp)
{
	struct Node *r, *z;
	z = *temp;
	if(z == NULL)
	{
		r =(struct Node*)malloc(sizeof(struct Node));
		r->coeff = x;
		r->pow = y;
		*temp = r;
		r->next = (struct Node*)malloc(sizeof(struct Node));
		r = r->next;
		r->next = NULL;
	}
	else
	{
		r->coeff = x;
		r->pow = y;
		r->next = (struct Node*)malloc(sizeof(struct Node));
		r = r->next;
		r->next = NULL;
	}
}
```

```
// Function Adding two polynomial numbers
void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1->next && poly2->next)
        {
                // If power of 1st polynomial is greater then 2nd, then store 1st as it is
                // and move its pointer
                if(poly1->pow > poly2->pow)
                {
                        poly->pow = poly1->pow;
                        poly->coeff = poly1->coeff;
                        poly1 = poly1->next;
                }

                // If power of 2nd polynomial is greater then 1st, then store 2nd as it is
                // and move its pointer
                else if(poly1->pow < poly2->pow)
                {
                        poly->pow = poly2->pow;
                        poly->coeff = poly2->coeff;
                        poly2 = poly2->next;
                }

                // If power of both polynomial numbers is same then add their coefficients
                else
                {
                        poly->pow = poly1->pow;
                        poly->coeff = poly1->coeff+poly2->coeff;
                        poly1 = poly1->next;
                        poly2 = poly2->next;
                }

                // Dynamically create new node
                poly->next = (struct Node *)malloc(sizeof(struct Node));
                poly = poly->next;
                poly->next = NULL;
        }
while(poly1->next || poly2->next)
        {
                if(poly1->next)
                {
                        poly->pow = poly1->pow;
                        poly->coeff = poly1->coeff;
                        poly1 = poly1->next;
                }
                if(poly2->next)
                {
                        poly->pow = poly2->pow;
                        poly->coeff = poly2->coeff;
                        poly2 = poly2->next;
                }
                poly->next = (struct Node *)malloc(sizeof(struct Node));
                poly = poly->next;
                poly->next = NULL;
        }
}

// Display Linked list
void show(struct Node *node)
{
while(node->next != NULL)
        {
        printf("%dx^%d", node->coeff, node->pow);
        node = node->next;
        if(node->next != NULL)
                printf(" + ");
        }
}

// Main program
int main()
{
        struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;

        // Create first list of 5x^2 + 4x^1 + 2x^0
        create_node(5,2,&poly1);
        create_node(4,1,&poly1);
        create_node(2,0,&poly1);

        // Create second list of 5x^1 + 5x^0
        create_node(5,1,&poly2);
        create_node(5,0,&poly2);

        printf("1st Number: ");
        show(poly1);

        printf("\n2nd Number: ");
        show(poly2);

        poly = (struct Node *)malloc(sizeof(struct Node));

        // Function add two polynomial numbers
        polyadd(poly1, poly2, poly);

        // Display resultant List
        printf("\nAdded polynomial: ");
        show(poly);

return 0;
}
```

3. (a) What are different types of queues ? How can we use the queue data structure for simulation.

Queue is a linear structure which follows a particular order in which the operations are performed.The order is First In First Out (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.

Operations on Queue:

Mainly the following four basic operations are performed on queue:

Enqueue:Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition. Dequeue:Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front:Get the front item from queue.

Rear: Get the last item from queue.

Types of queue:

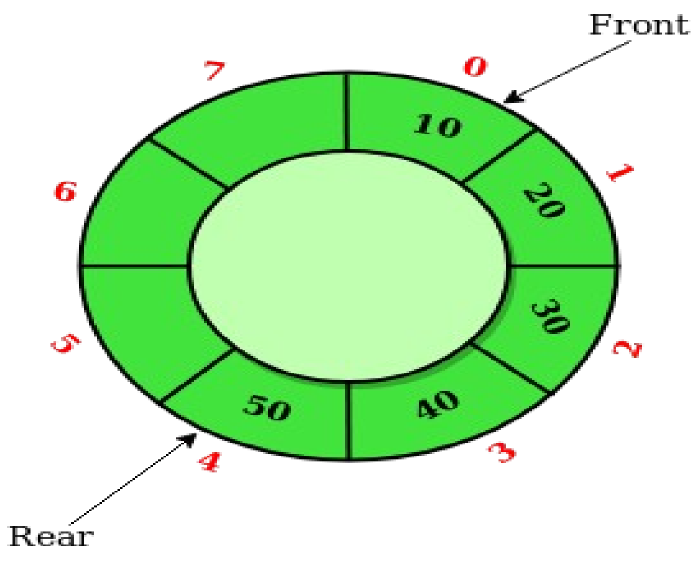
Priority Queue: Priority Queue is an extension of queue with following properties.

1) Every item has a priority associated with it.

2) An element with high priority is dequeued before an element with low priority.

3) If two elements have the same priority, they are served according to their order in the queue.

Circular Queue: Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

(b) Write a function to implement Radix Sort. Sort the following numbers using Radix Sort ; 25, 10, 68, 19, 75, 43, 22, 31, 11, 59. Show output after each pass.

Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

Initial pass	**First pass (tens place digit)**	**Second pass (unit place digit)**
25	10	10
10	31	11
68	11	19
19	22	22
75	43	25
43	25	31
22	75	43
31	68	59
11	19	68
59	59	75

4. (a) Write a C program to implement a Circular Linked List which performs the following operations : (i) Inserting element in the beginning (ii) Inserting element in the end (iii) Deleting the last element (iv) Deleting a particular element (v) Displaying the list

```
/*
* C Program to Demonstrate Circular Single Linked List
*/
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int data;
   struct node *link;
};
struct node *head = NULL, *x, *y, *z;

void create();
void ins_at_beg();
void ins_at_pos();
void del_at_beg();
void del_at_pos();
void traverse();
void search();
void sort();
void update();
void rev_traverse(struct node *p);

void main()
{
    int ch;
    printf("\n 1.Creation \n 2.Insertion at beginning \n 3.Insertion at remaining");
    printf("\n4.Deletion at beginning \n5.Deletion at remaining \n6.traverse");
    printf("\n7.Search\n8.sort\n9.update\n10.Exit\n");
    while (1)
    {
        printf("\n Enter your choice:");
        scanf("%d", &ch);
        switch(ch)
        {
        case 1:
            create();
            break;
        case 2:
            ins_at_beg();
            break;
        case 3:
            ins_at_pos();
            break;
        case 4:
            del_at_beg();
            break;
        case 5:
            del_at_pos();
            break;
        case 6:
            traverse();
            break;
        case 7:
            search();
            break;
        case 8:
            sort();
            break;
        case 9:
            update();
            break;
        case 10:
            rev_traverse(head);
            break;
        default:
            exit(0);
        }
    }
}

/*Function to create a new circular linked list*/
void create()
{
    int c;
    x = (struct node*)malloc(sizeof(struct node));
    printf("\n Enter the data:");
    scanf("%d", &x->data);
    x->link = x;
    head = x;
    printf("\n If you wish to continue press 1 otherwise 0:");
    scanf("%d", &c);
    while (c != 0)
    {
        y = (struct node*)malloc(sizeof(struct node));
        printf("\n Enter the data:");
        scanf("%d", &y->data);
        x->link = y;
        y->link = head;
        x = y;
        printf("\n If you wish to continue press 1 otherwise 0:");
        scanf("%d", &c);
    }
}
/*Function to insert an element at the begining of the list*/
void ins_at_beg()
{
    x = head;
    y = (struct node*)malloc(sizeof(struct node));
    printf("\n Enter the data:");
    scanf("%d", &y->data);
    while (x->link != head)
    {
        x = x->link;
    }
    x->link = y;
    y->link = head;
    head = y;
}
/*Function to insert an element at any position the list*/
void ins_at_pos()
{
    struct node *ptr;
    int c = 1, pos, count = 1;
    y = (struct node*)malloc(sizeof(struct node));
    if (head == NULL)
    {
        printf("cannot enter an element at this place");
    }
    printf("\n Enter the data:");
    scanf("%d", &y->data);
    printf("\n Enter the position to be inserted:");
    scanf("%d", &pos);
    x = head;
    ptr = head;
    while (ptr->link != head)
    {
        count++;
        ptr = ptr->link;
    }
    count++;
    if (pos > count)
    {
        printf("OUT OF BOUND");
        return;
    }
    while (c < pos)
    {
        z = x;
        x = x->link;
        c++;
    }
    y->link = x;
    z->link = y;
}
/*Function to delete an element at any begining of the list*/
void del_at_beg()
{
    if (head == NULL)
        printf("\n List is empty");
    else
    {
        x = head;
        y = head;
        while (x->link !=  head)
        {
            x = x->link;
```

```c
        }
        head = y->link;
        x->link = head;
        free(y);
    }
}
/*Function to delete an element at any position the list*/
void del_at_pos()
{
    if (head == NULL)
        printf("\n List is empty");
    else
    {
        int c = 1, pos;
        printf("\n Enter the position to be deleted:");
        scanf("%d", &pos);
        x = head;
        while (c < pos)
        {
            y = x;
            x = x->link;
            c++;
        }
        y->link = x->link;
        free(x);
    }
}
/*Function to display the elements in the list*/
void traverse()
{
    if (head == NULL)
        printf("\n List is empty");
    else
    {
        x = head;
        while (x->link !=  head)
        {
            printf("%d->", x->data);
            x = x->link;
        }
        printf("%d", x->data);
    }
}
/*Function to search an element in the list*/
void search()
{
    int search_val, count = 0, flag = 0;
    printf("\nenter the element to search\n");
    scanf("%d", &search_val);
    if (head == NULL)
        printf("\nList is empty nothing to search");
    else
    {
        x = head;
        while (x->link !=  head)
        {
            if (x->data == search_val)
            {
                printf("\nthe element is found at %d", count);
                flag = 1;
                break;
            }
            count++;
            x = x->link;
        }
        if (x->data == search_val)
        {
            printf("element found at postion %d", count);
        }
        if (flag == 0)
        {
            printf("\nelement not found");
        }
    }
}
/*Function to sort the list in ascending order*/
void sort()
{
    struct node *ptr, *nxt;
    int temp;
    if (head == NULL)
    {
        printf("empty linkedlist");
    }
    else
    {
        ptr = head;
        while (ptr->link !=  head)
        {
            nxt = ptr->link;
            while (nxt !=  head)
            {
                if (nxt !=  head)
                {
                    if (ptr->data > nxt->data)
                    {
                        temp = ptr->data;
                        ptr->data = nxt->data;
                        nxt->data = temp;
                    }
                }
                else
                {
                    break;
                }
                nxt = nxt->link;
            }
            ptr = ptr->link;
        }
    }
}
/*Function to update an element at any position the list*/
void update()
{
    struct node *ptr;
    int search_val;
    int replace_val;
    int flag = 0;
    if (head == NULL)
    {
        printf("\n empty list");
    }
    else
    {
        printf("enter the value to be edited\n");
        scanf("%d", &search_val);
        fflush(stdin);
        printf("enter the value to be replace\n");
        scanf("%d", &replace_val);
        ptr = head;
        while (ptr->link !=  head)
        {
            if (ptr->data == search_val)
            {
                ptr->data = replace_val;
                flag = 1;
                break;
            }
            ptr = ptr->link;
        }
        if (ptr->data == search_val)
        {
            ptr->data = replace_val;
            flag = 1;
        }
        if (flag == 1)
        {
            printf("\nUPdate sucessful");
        }
        else
        {
            printf("\n update not successful");
        }
    }
}
/*Function to display the elements of the list in reverse order*/
void rev_traverse(struct node *p)
{
    int i = 0;
    if (head == NULL)
    {
        printf("empty linked list");
    }
    else
    {
        if (p->link !=  head)
        {
            i = p->data;
            rev_traverse(p->link);
            printf(" %d", i);
        }
        if (p->link == head)
        {
            printf(" %d", p->data);
        }
    }
}
```

(b) Apply Huffman Coding for the word 'MALAYALAM'. Give the Huffman code for each symbol.

Huffman's algorithm is based on the idea that a variable length code should use the shortest code words for the most likely symbols and the longest code words for the least likely symbols. In this way, the average Code length will be reduced. The algorithm assigns code words to symbols by constructing a binary coding tree. Each symbol of the alphabet is a leaf of the coding tree. The code of a given symbol corresponds to the unique path from the root to that leaf, with 0 or 1 added to the code for each edge along the path depending on whether the left or right child of a given node occurs next along the path.

5. (a) Write a program to evaluate postfix expression.

```
// C program to evaluate value of a postfix expression
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Stack type
struct Stack
{
        int top;
        unsigned capacity;
        int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
        struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

        if (!stack) return NULL;

        stack->top = -1;
        stack->capacity = capacity;
        stack->array = (int*) malloc(stack->capacity * sizeof(int));

        if (!stack->array) return NULL;

        return stack;
}

int isEmpty(struct Stack* stack)
{
        return stack->top == -1 ;
```

```
}

char peek(struct Stack* stack)
{
	return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
	if (!isEmpty(stack))
		return stack->array[stack->top--] ;
	return '$';
}

void push(struct Stack* stack, char op)
{
	stack->array[++stack->top] = op;
}


// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
	// Create a stack of capacity equal to expression size
	struct Stack* stack = createStack(strlen(exp));
	int i;

	// See if stack was created successfully
	if (!stack) return -1;

	// Scan all characters one by one
	for (i = 0; exp[i]; ++i)
	{
		// If the scanned character is an operand (number here),
		// push it to the stack.
		if (isdigit(exp[i]))
			push(stack, exp[i] - '0');

		// If the scanned character is an operator, pop two
		// elements from stack apply the operator
		else
		{
			int val1 = pop(stack);
			int val2 = pop(stack);
			switch (exp[i])
			{
			case '+': push(stack, val2 + val1); break;
			case '-': push(stack, val2 - val1); break;
			case '*': push(stack, val2 * val1); break;
			case '/': push(stack, val2/val1); break;
			}
		}
```

```c
	}
	return pop(stack);
}

// Driver program to test above functions
int main()
{
	char exp[] = "231*+9-";
	printf ("Value of %s is %d", exp, evaluatePostfix(exp));
	return 0;
}
```

(b) Write a program in C to delete a node from a Binary Search Tree. The program should consider all the possible cases.

```c
// C program to demonstrate delete operation in binary search tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
	int key;
	struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
	struct node *temp = (struct node *)malloc(sizeof(struct node));
	temp->key = item;
	temp->left = temp->right = NULL;
	return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
	if (root != NULL)
	{
		inorder(root->left);
		printf("%d ", root->key);
		inorder(root->right);
	}
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
	/* If the tree is empty, return a new node */
	if (node == NULL) return newNode(key);

	/* Otherwise, recur down the tree */
	if (key < node->key)
		node->left = insert(node->left, key);
	else
		node->right = insert(node->right, key);

	/* return the (unchanged) node pointer */
	return node;
}

/* Given a non-empty binary search tree, return the node with minimum
key value found in that tree. Note that the entire tree does not
need to be searched. */
struct node * minValueNode(struct node* node)
{
	struct node* current = node;

	/* loop down to find the leftmost leaf */
	while (current->left != NULL)
		current = current->left;

	return current;
}

/* Given a binary search tree and a key, this function deletes the key
and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
	// base case
	if (root == NULL) return root;

	// If the key to be deleted is smaller than the root's key,
	// then it lies in left subtree
	if (key < root->key)
		root->left = deleteNode(root->left, key);

	// If the key to be deleted is greater than the root's key,
	// then it lies in right subtree
	else if (key > root->key)
		root->right = deleteNode(root->right, key);

	// if key is same as root's key, then This is the node
	// to be deleted
	else
	{
		// node with only one child or no child
		if (root->left == NULL)
		{
			struct node *temp = root->right;
			free(root);
			return temp;
		}
		else if (root->right == NULL)
```

```
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Driver Program to test above functions
int main()
{
    /* Let us create a BST

    struct node *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
```

```
}
```

6. (a) Write a program in C to implement the BFS traversal of a graph.

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

#define MAX 5

struct Vertex {
   char label;
   bool visited;
};

//queue variables

int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;

//graph variables

//array of vertices
struct Vertex* lstVertices[MAX];

//adjacency matrix
int adjMatrix[MAX][MAX];

//vertex count
int vertexCount = 0;

//queue functions

void insert(int data) {
   queue[++rear] = data;
   queueItemCount++;
}

int removeData() {
   queueItemCount--;
   return queue[front++];
}

bool isQueueEmpty() {
   return queueItemCount == 0;
}

//graph functions
```

```
//add vertex to the vertex list
void addVertex(char label) {
   struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
   vertex->label = label;
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
   printf("%c ",lstVertices[vertexIndex]->label);
}

//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
   int i;

   for(i = 0; i<vertexCount; i++) {
      if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
         return i;
   }

   return -1;
}

void breadthFirstSearch() {
   int i;

   //mark first node as visited
   lstVertices[0]->visited = true;

   //display the vertex
   displayVertex(0);

   //insert vertex index in queue
   insert(0);
   int unvisitedVertex;

   while(!isQueueEmpty()) {
      //get the unvisited vertex of vertex which is at front of the queue
      int tempVertex = removeData();

      //no adjacent vertex found
      while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
         lstVertices[unvisitedVertex]->visited = true;
         displayVertex(unvisitedVertex);
```

```
        insert(unvisitedVertex);
      }

    }

    //queue is empty, search is complete, reset the visited flag
    for(i = 0;i<vertexCount;i++) {
      lstVertices[i]->visited = false;
    }
}

int main() {
    int i, j;

    for(i = 0; i<MAX; i++) // set adjacency {
      for(j = 0; j<MAX; j++) // matrix to 0
        adjMatrix[i][j] = 0;
    }

    addVertex('S');   // 0
    addVertex('A');   // 1
    addVertex('B');   // 2
    addVertex('C');   // 3
    addVertex('D');   // 4

    addEdge(0, 1);    // S - A
    addEdge(0, 2);    // S - B
    addEdge(0, 3);    // S - C
    addEdge(1, 4);    // A - D
    addEdge(2, 4);    // B - D
    addEdge(3, 4);    // C - D

    printf("\nBreadth First Search: ");

    breadthFirstSearch();

    return 0;
}
```

(b) Hash the following elements in a table of size 11. Use any two collision resolution techniques : 23, 55, 10, 71, 67, 32, 100, 18, 10, 90, 44 .

Students may choose to implement any tow collision resolution techniques of their choice.