CONT. - DELETEING THE SPECIFIED NODE FROM THE LIST

```
void linklist:: del (int num)
{
    node *old, *temp;
    temp = p;
    while (temp != NULL)
    {
        if (temp -> data == num)
        {
        // if node to be deleted is the
        // first node in the linked list
            if (temp == p)
                p = temp -> link;
        // free the memory occupied by the node
                delete temp;
                return;
            }
            // traverse the linked list till the last node //is
            reached
            else
            {
            // old points to the previous node
                old = temp;
                // go to the next node
                temp = temp -> link;
            }
    }
    cout << “Element” << num << “not found” << endl;
}
```

CONT. – DEALLOCATING MEMORY

```
linklist:: ~linklist()
{
        node *q;
        while (p != NULL)
        {
                q = p -> link;
                delete p;
                p = q;
      }
}
```

REVERSING LINKS

```
#include
using namespace std;
class linklist
{
      private:
/* structure containing a data part and link part */
      struct node
      {
            int data;
            node *link;
      } *p;
      public:
      linklist();
      void addatbeg (int num);
      void reverse();
      void display();
      ~linklist();
};
```

REVERSING THE LINKED

```
void linklist:: reverse()
{
        node *q, *r, *s;
        q = p;
        r = NULL;
        // traverse the entire linked list
        while (q != NULL)
        {
                s = r;
                r = q;
                q = q -> link;
                r -> link = s;
        }
        p = r;
}
```

CONCATENATING LINKED LISTS

```
#include
using namespace std;
class linklist
{
      private:
/* structure containing a data part and link part */
      struct node
      {
            int data;
            node *link;
      } *p;
      public:
      linklist();
      void append (int num);
      void concat (linklist &l);
      void display();
      int count();
      ~linklist();
};
```

CONCATENATING TWO LINKED LISTS

```
void linklist:: concat (linklist &l)
{
      node *temp;
      // if the first linked list is empty
      if (p == NULL)
            p = l.p;
      else
      {
      // if both linked lists are non-empty
            if (l.p != NULL)
            {
      // points to the starting of the first list
                  temp = p;
      // traverse the entire first linked list
                  while (temp -> link != NULL)
                        temp = temp -> link;
      // concatenate the second list after
      //the first
                  temp -> link = l.p;
            }
      }
      l.p = NULL;
}
```

RECURSIVE – CODE

```
#include
using namespace std;
class linklist
{
      private:
/* structure containing a data part and link part */
      struct node
      {
            int data;
            node *link;
      } *p;
      public:
      linklist();
      void add (int num);
      void addatend (node **ptr, int num);
      void display();
      int length();
      int getlength(node *ptr);
      int operator == (linklist &l);
      int compare (node *ptr1, node *ptr2);
      void copy (linklist &l);
      void duplicate (node *ptr1, node **ptr2);
      ~linklist();
};
```

COMPARING TWO LINKED LISTS

```
// compares 2 linked lists and returns 1 if linked lists are equal and 0 if unequal
int linklist:: compare (node *q, node *r)
{
    if (q == NULL && r == NULL)
        return 1;
    else
    {
        if (q == NULL || r == NULL)
            return 0;
        else if (q -> data != r -> data)
                return 0;
            else
                compare (q -> link, r -> link);
    }
}
```

CLONING A LINKED LIST

```
// copies a linked list into another
void linklist:: duplicate (node *ptr1, node **ptr2)
{
      if (ptr1 != NULL)
      {
            *ptr2 = new node;
            (*ptr2) -> data = ptr1 -> data;
            (*ptr2) -> link = NULL;
            duplicate (ptr1 -> link, &((*ptr2) -> link));
      }
}
```

DOUBLY LINKED LIST – CODE

```
#include
using namespace std;
class linklist
{
      private:
/* structure representing a node of the doubly
linked list */
      struct dnode
      {
            dnode *prev;
            int data;
            dnode *next;
      } *p;
      public:
      linklist();
      void d_append (int num);
      void d_addatbeg (int num);
      void d_addafter (int loc, int num);
      void d_display();
      int d_count();
      void d_delete(int i);
      ~linklist();
};
```

ADDING A NEW NODE TO THE LIST

```
// add a new node at the end of the list
void linklist:: d_append (int num)
{
      dnode *r, *q;
      q = p;
      // if the linked list is empty
      if (q == NULL)
      {
      //create a new node
            q = new dnode;
            q -> prev = NULL;
            q -> data = num;
            q -> next = NULL;
            p = q;
      }
            else
            {
// traverse the linked list to  reach the last node
                  while (q -> next != NULL)
                        q = q -> next;
                  // add a new node at the end
                  r = new dnode;
                  r -> data = num;
                  r -> next = NULL;
                  r -> prev = q;
                  q -> next = r;
            }
}
```

ADDING A NEW NODE AT THE BEGINNING OF THE LIST

```
void linklist:: d_addatbeg (int num)
{
dnode *q;
// create a new node
q = new dnode;
q -> prev = NULL;
q -> data = num;
q -> next = p;
// make new node the head node
p -> prev = q;
p = q;
}
```

ADDING A NODE AFTER THE SPECIFIED NUMBER OF NODES

```
void linklist:: d_addafter (int loc, int num)
{
    dnode *q;
    q = p;
    // skip to desired portion
    for (int i = 1; i <= loc; i++)
    {
        q = q -> next;
    // if end of linked list is encountered
        if (q == NULL)
        {
        cout << "There are less than" << loc <<
"elements" << endl;
        return;
        }
    }
    // insert new node
    q = q -> prev;
    dnode *temp = new dnode;
    temp -> data = num;
    temp -> prev = q;
    temp -> next = q -> next;
    temp -> next -> prev = temp;
    q -> next = temp;
}
```

H62/

1-insert function of doubley linked list after specific location

DELETING THE SPECIFIED NODE FROM THE DOUBLY LIST

```
void linklist:: d_delete (int num)
{
      dnode *q = p;
      // traverse the entire linked list
      while (q != NULL)
      {
      // if node to be deleted is found
            if (q -> data == num)
            {
// if node to be deleted is the first node
                  if (q == p)
                  {
                        p = p -> next;
                        p -> prev = NULL;
                  }
                  else
                  {
            // if node to be deleted is the last node
                        if (q -> next == NULL)
                              q -> prev -> next = NULL;
                        else
            // if node to be deleted is any intermediate node
                        {
                              q -> prev -> next = q -> next;
                              q -> next -> prev = q -> prev;
                        }
```

CONT. – CODE

```
                    delete q;
                }
// return back after deletion
                return;
            }
            // go to next node
            q = q -> next;
        }
        cout << num << "not found" << endl;
}
```

IMPLEMENTATION OF QUEUE AS AN ARRAY

```
#include
using namespace std;
const int MAX = 10;
class queue
{
      private:
            int arr[MAX];
            int front, rear;
      public:
            queue();
            void addq (int item);
            int delq();
};
```

```
// initializing data members
queue:: queue()
{
front = -1;
rear = -1;
}
```

ADDING AN ELEMENT TO THE QUEUE

```
void queue:: addq (int item)
{
      if (rear == MAX - 1)
      {
            cout << “Queue is full” << endl;
            return;
      }
      rear++;
      arr[rear] = item;
      if (front == -1)
            front = 0;
}
```

REMOVING AN ELEMENT FROM THE QUEUE

```
int queue:: delq()
{
    int data;
    if (front == -1)
    {
        cout << "Queue is Empty" << endl;
        return NULL;
    }
    data = arr[front];
    arr[front] = 0;
    if (front == rear)
        front = rear = -1;
    else
        front++;
    return data;
}
```

IMPLEMENTATION QUEUE AS A LINKED LIST

```
#include
using namespace std;
class queue
{
    private:
        struct node
        {
            int data;
            node *link;
        } *front, *rear;
    public:
        queue();
        void addq (int item);
        int delq();
        ~queue();
};

// initializing a data member
queue:: queue()
{
front = rear = NULL;
}
```

ADDING A NEW NODE TO THE QUEUE

```
void queue:: addq (int item)
{
      node *temp;
      temp = new node;
      if (temp == NULL){
            cout << "Queue is full" << endl;
            return;}
      temp -> data = item;
      temp -> link = NULL;
      if (front == NULL)
      {
            rear = front = temp;
            return;
      }
      rear -> link = temp;
      rear = rear -> link;
}
```

REMOVING AN ELEMENT FROM THE QUEUE

```
int queue:: delq()
{
    if (front == NULL)
    {
    cout << "Queue is empty" << endl;
    return NULL;
    }
    node *temp;
    int item;
    item = front -> data;
    temp = front;
    front = front -> link;
    delete temp;
    return item;
}
```

462/

2-deqeueu operation as linked list

DEALLOCATING MEMORY

```
queue:: ~queue()
{
      if (front == NULL)
            return;
      node *temp;
      while (front != NULL)
      {
            temp = front;
            front = front -> link;
            delete temp;
      }
}
```

1-بتشابتر الكيو (ديستركتور للكيو)

IMPLEMENTATION OF CIRCULAR QUEUE

```
#include
using namespace std;
const int MAX = 8;
class queue
{
    private:
        int arr[MAX];
        int front, rear;
    public:
        queue();
        void addq (int item);
        int delq();
        void display();
        ~queue();
};
// initializing a data member
queue:: queue()
{
    front = rear = -1;
    for (int i = 0; i < MAX; i++)
        arr[i] = 0;
}
```

ADDING AN ELEMENT TO THE CIRCULAR QUEUE

```
void queue:: addq (int item)
{
    if ((rear == MAX - 1 && front == 0) || (rear + 1 == front))
    {
        cout << "Queue is full" << endl;
        return;
    }
    if (rear == MAX - 1)
        rear = 0;
    else
        rear++;
    arr[rear] = item;
    if (front == -1)
        front = 0;
}
```

REMOVING AN ELEMENT FROM THE QUEUE

```
int queue:: delq()
{
      int data;
      if (front == -1)
      {
      cout << "Queue is empty" << endl;
      return NULL;
      }
      data = arr[front];
      arr[front] = 0;
      if (front == rear)
      {
            front = -1;
            rear = -1;
      }
      else
      {
            if (front == MAX - 1)
                  front = 0;
            else
                  front++;
      }
      return data;
}
```

STACK AS AN ARRAY

```
#include
using namespace std;
const int MAX = 10;
class stack
{
      private:
            int arr[MAX];
            int top;
      public:
            stack();
            void push (int item);
            int pop();

};
```

```
// initializing a data member
stack:: stack()
{
top = -1;
}
```

ADDING AN ELEMENT TO THE STACK

```
void stack:: push (int item)
{
    if (top == MAX - 1)
    {
        cout << “Stack is full” << endl;
        return;
    }
    top++;
    arr[top] = item;
}
```

EXTRACTING AN ELEMENT FROM THE STACK

```
int stack:: pop()
{
    if (top == -1)
    {
        cout << “Stack is empty” << endl;
        return NULL;
    }
    int data = arr[top];
    Arr[top] = 0;
    top--;
    return data;
}
```

IMPLEMENTATION STACK AS A LINKED LIST

```
#include
using namespace std;
class stack
{
    private:
        struct node
        {
            int data;
            node *link;
        } *top;
    public:
        stack();
        void push (int item);
        int pop();
        ~stack();
};

// initializing a data member
stack:: stack()
{
top = NULL;
}
```

ADDING A NEW NODE TO THE STACK

```
void stack:: push (int item){
//add node at the beginning
      node *temp;
      temp = new node;
      if (temp == NULL){
            cout << "Stack is full" << endl;
            return;}
      temp -> data = item;
      temp -> link = top;
      top = temp;
}
```

452/

[4 points]

Question 4: Implement a function "**push**" using linked list to add a new node to the stack. [5 points]

```
void stack:: push (int item){

node *temp;
temp = new node;

if (temp == NULL){
cout << "Stack is full" << endl;
full.
return;
}

temp -> data = item;
temp -> link = top;.
top = temp;

}
```

POP AN ELEMENT FROM THE STACK

```
int stack:: pop()
{
//Remove the first element in list
      if (top == NULL)
      {
            cout << "Stack is empty" << endl;
            return NULL;
      }
      node *temp;
      int item;
      temp = top;
      item = temp -> data;
      top = top -> link;
      delete temp;
      return item;
}
```

DEALLOCATING MEMORY

```
stack:: ~stack()
{
      if (top == NULL)
            return;
      node *temp;
      while (top != NULL)
      {
      temp = top;
      top = top -> link;
      delete temp;
      }
}
```

IMPLEMENTATION OF BST OPERATIONS

```
#include <iostream>
using namespace std;
struct btreenode
{
    btreenode *leftchild;
    int data;
    btreenode *rightchild;
};
class btree
{
private:
    btreenode *root;
    void inorder (btreenode *sr);
    void preorder (btreenode *sr);
    void postorder (btreenode *sr);
    void insert (btreenode **sr, int);
    bool search (struct btreenode *, int);
    void locate (btreenode **sr, int num,
btreenode **par, btreenode **x, bool *found);
    void rem (btreenode **sr, int num);
    void del (btreenode *sr);
```

IMPLEMENTATION OF BST OPERATIONS – CONT.

```
public:
    btree();
    void buildtree (int num);
    void display();
    bool searchbst (int);
    void remove (int num);
    ~btree();
};
```

```
// initialises data members
btree:: btree()
{
    root = NULL;
}
// calls insert() to build tree
void btree:: buildtree (int num)
{
    insert (&root, num);
}
```

INSERTING A NEW NODE IN BST

```
void btree:: insert (btreenode **sr, int num)
{
    if (*sr == NULL)
    {
        *sr = new btreenode;
        (*sr)->leftchild = NULL;
        (*sr)->data = num;
        (*sr)->rightchild = NULL;
    }
// search the node to which new node will be attached};
else
{
// if new data is less, traverse to left
    if (num < (*sr)->data)
        insert (& ((*sr)->leftchild), num);
    else
// else traverse to right
    insert (& ((*sr)->rightchild), num);
}
}
```

TRAVERSING BST

```
// traverse BST in Left-Root-Right fashion
void btree:: inorder (btreenode *sr)
{
    if (sr != NULL)
    {
        inorder (sr->leftchild);
        cout << sr->data << "\t";
        inorder (sr->rightchild);
    }
}
```

```
// traverse BST in Root-Left-Right fashion
void btree:: preorder (btreenode *sr)
{
    if (sr != NULL)
    {
        cout << sr->data << "\t";
        preorder (sr->leftchild);
        preorder (sr->rightchild);
    }
}
```

TRAVERSING BST – CONT.

```
// traverse BST in Left-Right-Root fashion
void btree:: postorder (btreenode *sr)
{
    if (sr != NULL)
    {
        postorder (sr->leftchild);
        postorder (sr->rightchild);
        cout << sr->data << "\t";
    }
}
```

```
bool btree:: searchbst (int num)
{
    bool flag;
    flag = search (root, num);
    return flag;
}
```

SEARCHING BST

```
bool btree:: search (struct btreenode *sr, int
num)
{
      while (sr != NULL)
      {
            if (num == sr->data)
                  return true;
            else if (num < sr->data)
                  sr = sr->leftchild;
            else
                  sr = sr->rightchild;
      }
      return false;
}

// calls rem to delete node
void btree:: remove (int num)
{
      rem (&root, num);
}
```

451/

Q5) write a function to search for a number in BST

452/

[5 points]

Question 7: Write a function to search for a specific value in the binary search tree and return true if the value is found, false otherwise.

[5 points]

```
bool btree:: search (btreenode *sr, int num) {

while (sr != NULL) {
if (num == sr->data)
return true;

else if (num < sr->data)
sr = sr->leftchild;
else
sr = sr->rightchild;
}

return false;
```

DELETING A NODE FROM BST

```
// deletes a node from the BST
void btree:: rem (btreenode **sr, int num)
{
    bool found;
    btreenode *parent, *x, *xsucc;
    // if tree is empty
    if (*sr == NULL)
    {
    cout << endl << "Tree is empty";
    return;
    }
parent = x = NULL;
// call the locate function to find the node being
deleted
locate (sr, num, &parent, &x, &found);
// if the node to deleted is not found
if (found == false)
{
    cout << endl << "Data to be deleted, not
found";
    return;
}
```

461

1- بتشابتر التري (ديليت إذا كان عندنا ولدين)

DELETING A NODE FROM BST – CONT.

```
// If the node to be deleted has two children
if (x->leftchild != NULL && x->rightchild !=
NULL)
{
    parent = x;
    xsucc = x->rightchild;
    while (xsucc->leftchild != NULL)
    {
        parent = xsucc;
        xsucc = xsucc->leftchild;
    }
    x->data = xsucc->data;
    x = xsucc;
}
// if the node to be deleted has no child
if (x->leftchild == NULL && x->rightchild ==
NULL)
{
    if (parent->rightchild == x)
    parent->rightchild = NULL;
    else parent->leftchild = NULL;
    delete x;
return;
}
```

DELETING A NODE FROM BST – CONT.

```
// if the node to be deleted has only rightchild
if (x->leftchild == NULL && x->rightchild != NULL)
{
    if (parent->leftchild == x)
    parent->leftchild = x->rightchild;
    else
    parent->rightchild = x->rightchild;
    delete x;
    return;
}
```

```
// if the node to be deleted has only left child
if (x->leftchild != NULL && x->rightchild == NULL)
{
    if (parent->leftchild == x)
        parent->leftchild = x->leftchild;
    else
        parent->rightchild = x->leftchild;
    delete x;
    return;
}
}
```

DELETING A NODE FROM BST – CONT.

```
// returns the address of the node to be deleted,
address of its parent
// and whether the node is found or not
void btree:: locate (btreenode **sr, int num,
btreenode **par,
btreenode **x, bool *found)
{
      btreenode *q;
      q = *sr;
      *found = false;
      *par = NULL;
      while (q != NULL)
      {
      // if the node to be deleted is found
            if (q->data == num)
            {
                  *found = true;
                  *x = q;
                  return;
            }
            *par = q;
```

DELETING A NODE FROM BST – CONT.

```
        if (q->data > num)
                q = q->leftchild;
        else
                q = q->rightchild;
        }
}

// calls del to deallocate memory
btree:: ~btree()
{
        del (root);
}
```

```
// deletes nodes of a BST
void btree:: del (btreenode *sr)
{
        if (sr != NULL)
        {
                del (sr->leftchild);
                del (sr->rightchild);
        }
        delete sr;
}
```

452

[6 points]

Question 6: Write a recursive function in C++ that merges two sorted linked lists into a new sorted linked list and returns the head of new linked list.

[5 points]

Question 7: Write a fu...

مو متأكدة من الحل

```
struct node {
  int data;
  node* link;
};

node* mergeLists(node* L1, node* L2) {
  // Base cases
  if (L1 == NULL) return L2;
  if (L2 == NULL) return L1;

  node* result;

  // Pick smaller value and recurse
  if (L1->data < L2->data) {
    result = new node;
    result->data = L1->data;
    result->link = mergeLists(L1->link, L2);
  } else {
    result = new node;
    result->data = L2->data;
    result->link = mergeLists(L1, L2->link);
  }
  return result;
}
```

461/

الأكواد

1- بتشابتر التري (ديليت إذا كان عندنا ولدين)

1-بتشابتر الكيو (ديستركتور للكيو)

3-بتشابتر اللينكد ليست (ريكيرجون فنكشن تعد العناصر اللي باللينكد ليست)

مو متأكدة من الحل

```
int countNodes(node* head) {
  if (head == NULL)
    return 0;
  return 1 + countNodes(head->next);
}
```

462/

1-insert function of doubley linked list after specific location.

2-deqeueu operation as linked list

451/

Q4) write a function to count the number of nodes in linked list

من الميد

```
int linklist:: count()
{
    int c = 0;
    node *temp = p;
    // traverse the entire linked list
    while (temp != NULL)
    {
        c++;
        temp = temp -> link;
    }
    return c;
}
```