LINKED LIST

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

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

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 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

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
        return 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;
    return duplicate (ptr1 -> link, &((*ptr2) -> link));
    }
}
```

DOUBLY LINKED 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;
    }
}
```

```
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;
}
```

```
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;
}
```

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;
                }

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

QUEUE AS AN ARRAY

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

```
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;
}
```

STACK AS AN ARRAY

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

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

QUEUE AS A LINKED LIST

```
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;
}
```

```
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;
}
```

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

STACK AS A LINKED LIST

```
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;
}
```

```
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;
}
```

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

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;
}
```

```
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;
}
```

```
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;
}
```

BST OPERATIONS

```
void insert(btreenode*& sr, int num) {
    if (sr == NULL) {
        sr = new btreenode;
        sr->data = num;
        sr->leftchild = NULL;
        sr->rightchild = NULL;
    }
    else {
        if (num < sr->data)
            insert(sr->leftchild, num);
        else
            insert(sr->rightchild, num);
    }
}
```

```
bool search(btreenode* sr, int num)
{
    if (sr == NULL)
        return false;

    if (sr->data == num)
        return true;

    if (num < sr->data)
        return search(sr->leftchild, num);
    else
        return search(sr->rightchild, num);
}
```

```
// Inorder Traversal (Left - Root - Right)
void inorder(btreenode* sr) {
    if (sr != NULL) {
        inorder(sr->leftchild);
        cout << sr->data << "\t";
        inorder(sr->rightchild);
    }
}

// Preorder Traversal (Root - Left - Right)
void preorder(btreenode* sr) {
    if (sr != NULL) {
        cout << sr->data << "\t";
        preorder(sr->leftchild);
        preorder(sr->rightchild);
    }
}

// Postorder Traversal (Left - Right - Root)
void postorder(btreenode* sr) {
    if (sr != NULL) {
        postorder(sr->leftchild);
        postorder(sr->rightchild);
        cout << sr->data << "\t";
    }
}
```

```
void deleteBST(btreenode* &root, int num)
{
    if (root == NULL)
        return;

    if (num < root->data)
        deleteBST(root->leftchild, num);

    else if (num > root->data)
        deleteBST(root->rightchild, num);

    else
    {
// الحالة 1: بدون أبناء //
        if (root->leftchild == NULL && root->rightchild == NULL)
        {
            delete root;
            root = NULL;
        }
// الحالة 2: ابن يمين فقط //
        else if (root->leftchild == NULL)
        {
            btreenode* temp = root;
            root = root->rightchild;
            delete temp;
        }
// الحالة 3: ابن يسار فقط //
        else if (root->rightchild == NULL)
        {
            btreenode* temp = root;
            root = root->leftchild;
            delete temp;
        }
// الحالة 4: له ابنين //
        else
        {
            btreenode* temp = root->rightchild;
            while (temp->leftchild != NULL)
                temp = temp->leftchild;

            root->data = temp->data;
            deleteBST(root->rightchild, temp->data);
        }
    }
}
```