DATA STRUCTURES

Linked List

Prof. Khalil Al-Shqeerat

CS Dept. – Qassim University

OUTLINE

1. Array limitations
2. Linked list definition
3. Operations on Linked List
4. Implementation of various linked list operations

1. قيود المصفوفة (Array Limitations)
المصفوفة هي هيكل بيانات أساسي، لكن لها بعض القيود:
الحجم ثابت:
يجب تحديد حجم المصفوفة عند إنشائها ولا يمكن تغييره لاحقًا.
مثال: int arr[5]; لا يمكن أن يخزن أكثر من 5 عناصر.
هدر الذاكرة:
إذا كانت المصفوفة كبيرة ولم تُستخدم بالكامل، فالذاكرة غير المستخدمة تُهدر.
إذا كانت صغيرة جدًا، لن تكفي لتخزين كل العناصر بدون إنشاء مصفوفة جديدة.
الإدخال والحذف مكلفان:
الإدخال: لإضافة عنصر في منتصف المصفوفة، يجب تحريك جميع العناصر التالية.
الحذف: لإزالة عنصر، يجب تحريك العناصر للخلف.
كل هذه العمليات زمنها O(n).
تخزين نوع واحد فقط:
المصفوفة تخزن عناصر من نفس نوع البيانات فقط.

ARRAY LIMITATIONS

- Arrays have a fixed dimension.
  - Once the size of an array is decided it cannot be increased or decreased during execution.
- Insertion of a new element in an array is tedious because during insertion each element after the specified position has to be shifted one position to the right.
- Deletion of an existing element in an array is inefficient because during deletion each element after the specified position has to be shifted one position to the left.

SOLUTION!

- Linked list overcomes all these disadvantages.
- A linked list can grow and shrink in size during its lifetime.
- No maximum size of a linked list.
- Unlike arrays, while inserting or deleting elements in a linked list shifting of existing elements is not required.

الـLinked List جاءت لتعالج هذة المشكلة :

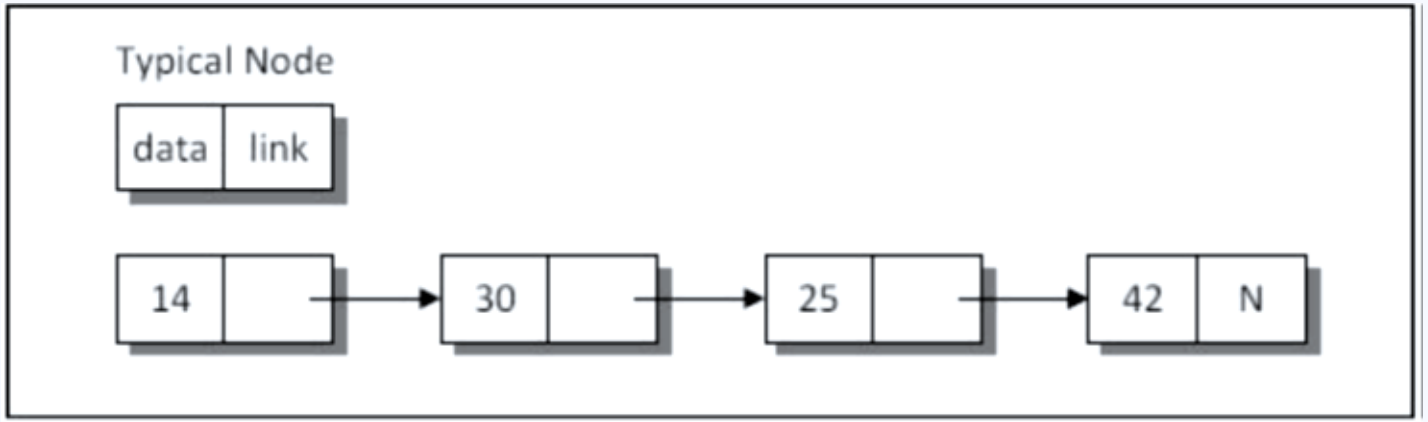
- الـLinked List تقدر تكبر وتصغر في وقت التنفيذ
- مافي حجم محدود للـ Linked List
- عند الحذف والإضافة، ما تتطلب تحريك للعناصر

هي هيكل بيانات خطي، ولكن العناصر (تسمى العقد Nodes) ليست في مواقع متجاورة بالذاكرة.
كل عقدة تحتوي على:
البيانات (Data): القيمة المخزنة.
المؤشر (Next): يشير إلى العقدة التالية في القائمة.
مميزات القائمة المرتبطة مقارنة بالمصفوفة:
٢- الحجم ديناميكي (يمكن إضافة عناصر أو حذفها بدون مشكلة حجم ثابت).
٣- الإدخال والحذف أسهل، لا حاجة لتحريك العناصر.

LINKED LIST DEFINITION

- Unlike array, linked list elements are not constrained to be stored in adjacent locations.
  - The order of the elements is maintained by explicit links between them.
- **Linked list** is a collection of elements called nodes, each of which stores two items of information—an element of the list and a link.
- The link part is a pointer to the next node.
  - It is shown using an arrow.
  - The NULL (N) in the last node indicates that it is the last node in the list.

LINKED LIST DEFINITION – CONT.




```
struct Node{
int data
Node * next
}
```

OPERATIONS ON LINKED LIST

- Several operations can be performed on linked lists:
  1. building a linked list by adding new node;
     - At the beginning,
     - At the end or
     - In the middle of the linked list),
  2. Deleting a node.
  3. Display contents of all nodes.

عمليات على القائمة المرتبطة
التنقل Traversal: الوصول لكل عقدة بالتتابع.
الإدخال Insertion: إضافة عقدة جديدة:
في البداية
في النهاية
في الوسط
الحذف Deletion: إزالة عقدة:
من البداية
من النهاية
حسب القيمة أو الموقع
طباعة Display

IMPLEMENTATION OF VARIOUS LINKED LIST OPERATIONS

```
#include
using namespace std;
class linklist
{
private:
// structure containing a data part
//and link part
    struct node
    {
        int data;
        node* link;
    } *p;
```

غالبًا راح يشير إلى العنصر الأول في الليست، تسمى بالـ head

```
public:
    linklist();
    void append (int num);
    void addatbeg (int num);
    void addafter (int loc, int num);
    void display();
    int count();
    void del (int num);
    ~linklist();
};
```

نشئ القائمة عند بداية البرنامج (غالبًا يخلي p = NULL)

يضيف عنصر في نهاية القائمة

يضيف عنصر في بداية القائمة

يضيف عنصر بعد موقع معين

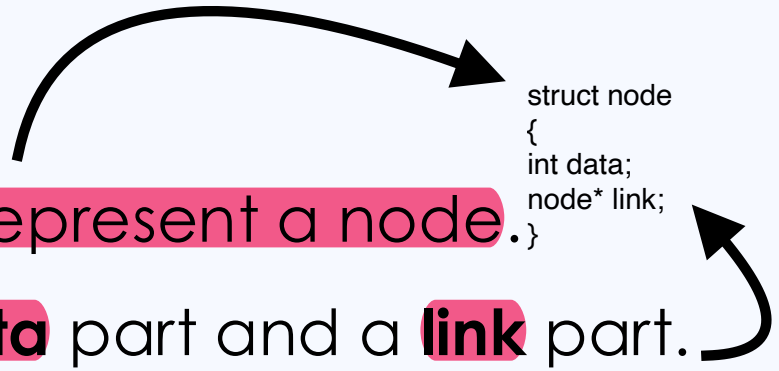
يعرض عناصر القائمة

يرجع عدد العقد

يحذف عقدة بقيمة معينة

يحذف كل العقد عند انتهاء البرنامج (تنظيف الذاكرة)

CLASS DESIGN..



- The class contains a structure to represent a node.
- The structure node contains a **data** part and a **link** part.
  - The variable **p** has been declared as pointer to a node.
  - The pointer is pointing to the first node in the linked list.
    - No matter how many nodes get added to the linked list, **p** would continue to point to the first node in the list.
  - When no node has been added to the list, **p** has been set to NULL to indicate that the list is empty.

عند عدم اضافة عنصر للـ Linked List، راح تضل p فارغة، هذا يعني ان الليست فارغة

أنشأنا struct node
data → يخزن الرقم
link → يشير للعقدة التالية
*p → هذا مؤشر لأول عقدة (Head)

CONT. – INITIALIZING DATA MEMBER

```
linklist:: linklist()
{
// The pointer to node has been set to NULL to indicate that the list is empty.
p = NULL;
}
```

This a constructor initializing p = null as an the list is empty

The following **append()** function has to deal with two situations:
(a) The node is being added to an empty list.
(b) The node is being added at the end of an existing list.

CONT. – ADDING A NODE AT THE END OF A LINKED LIST

```
void linklist:: append (int num)
{
      node *temp, *r;
      // if the list is empty, create first node
      if (p == NULL)
      {
            temp = new node;
            temp -> data = num;
            temp -> link = NULL;
            p = temp;
      }
      else
      {
            // go to last node
            temp = p;
            while (temp -> link != NULL)
                  temp = temp -> link;
            // add node at the end
            r = new node;
            r -> data = num;
            r -> link = NULL;
            temp -> link = r;
      }
}
```

FIRST CASE!

- if (p == NULL)
  - Memory is allocated for the node using **new** operator.
  - The **data** and the **link** part of this node are setup using the statements:

```
temp -> data = num;
temp -> link = NULL;
```

  - Lastly, **p** is made to point to this node, since the first node has been added to the list and **p** must always point to the first node.

SECOND CASE!

- In the other case, when the linked list is not empty.
  - **temp** is made to point to the first node in the list:

    temp = p;
  - Using **temp** we have traversed through the entire linked list:

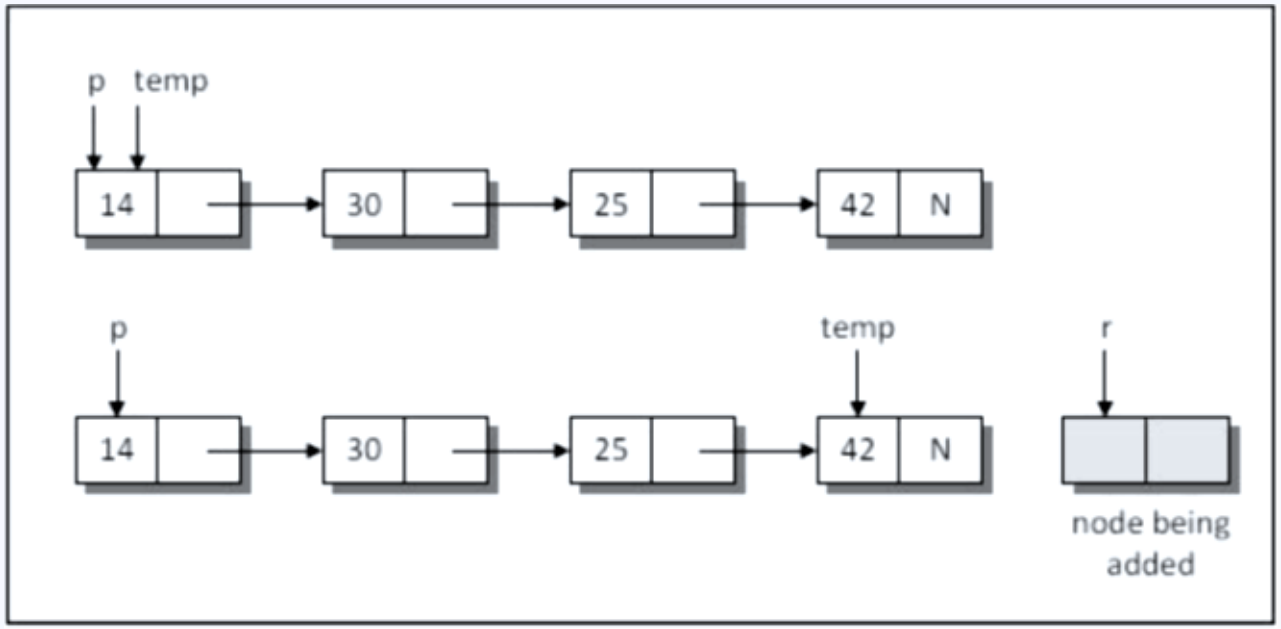
    while (temp -> link != NULL)

        temp = temp -> link;
    - Each time through the loop the statement **temp = temp -> link** makes **temp** point to the next node in the list.
    - When **temp** reaches the last node, memory for the new node is allocated.

      r =new node;
    - The previous last node is being pointed to by **temp** and the new last node is being pointed to by **r**, using the statement (temp -> link = r;).

WORKING OF APPEND() FUNCTION

- The position of the pointers before and after traversing the list:



CONT. – ADDING A NEW NODE TO THE LIST

```
void linklist:: addatbeg (int num)
{
// It adds a new node at the
//beginning of the linked list
      node *temp;
      temp = new node;
      temp -> data = num;
      temp -> link = p;
      p = temp;
}
```

For adding a new node at the beginning:

1. Memory is allocated for this node and data is stored in it.
2. The link part of this node points to the existing first node.
3. Lastly, this new node must be made the first node in the list.

CONT. – ADDING A NODE AFTER NODES

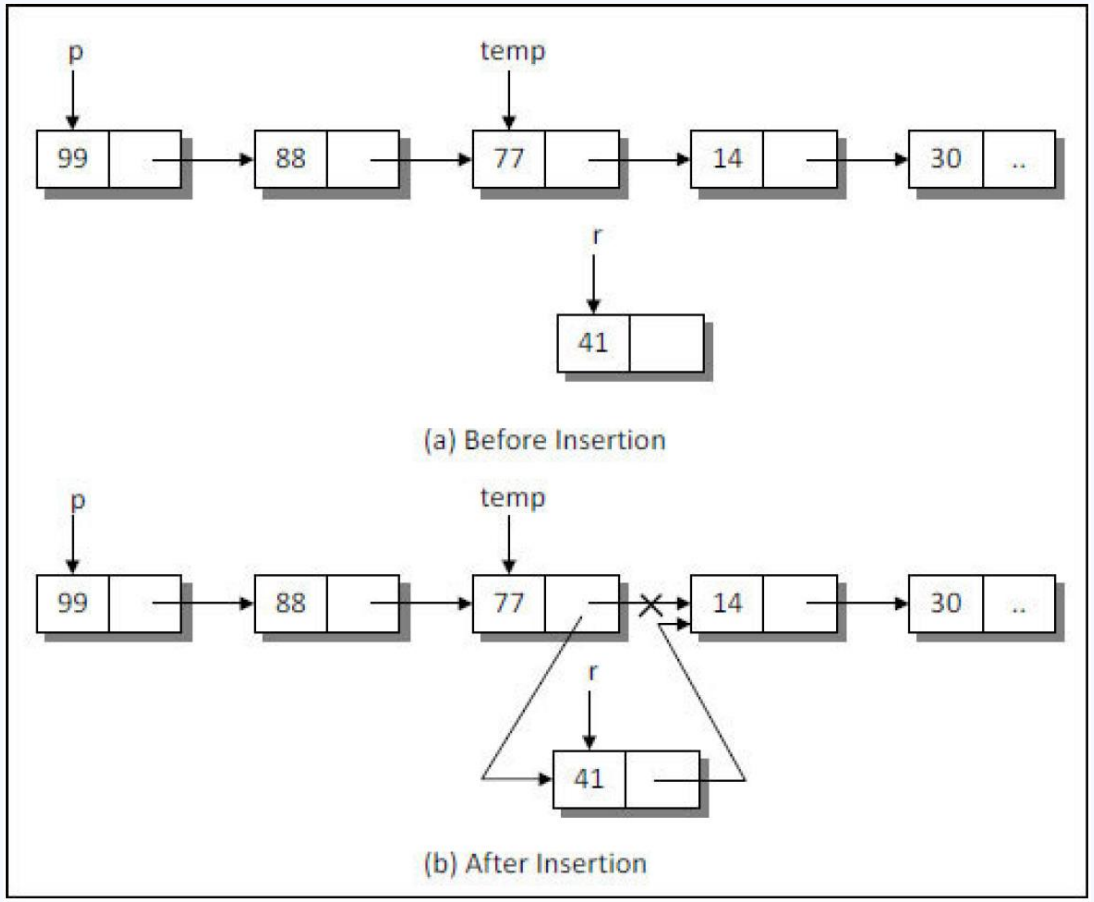
```
void linklist:: addafter (int loc, int num)
{
// It adds a new node after the specified
//number of nodes in the linked list.
      node *temp, *m;
      temp = p;
      // skip to desired portion
      for (int i = 0; i < loc - 1; i++)
      {
            temp = temp -> link;
            // if end of linked list is
            //encountered
            if (temp == NULL)
            {
                  cout << "\n There are less
                  than" << loc
                  << "elements in list" << endl;
                  return;
            }
      }
      // insert new node
      m = new node;
      m -> data = num;
      m -> link = temp -> link;
      temp -> link = m;
}
```

CONT.

- The **addafter()** function adds a new node after a specified number of nodes in the linked list.
- The **for** loop is used to skip the desired number of nodes after which a new node is to be added.
- The memory is allocated for the node to be inserted and the new number is stored in the data part of it.

WORKING OF ADDAFTER() FUNCTION

- The position of pointers once the control reaches outside the **for** loop is shown in Figure.



CONT. – DISPLAYING THE CONTENTS OF THE LINKED LIST

```
void linklist:: display()
{
      node *temp = p;
      // traverse the entire linked list
      while (temp != NULL)
      {
            cout << temp -> data << “”;
            temp = temp -> link;
      }
      cout << endl;
}
```

CONT. – COUNTING THE NUMBER OF NODES

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

CONT. - DELETEING THE SPECIFIED NODE FROM THE LIST

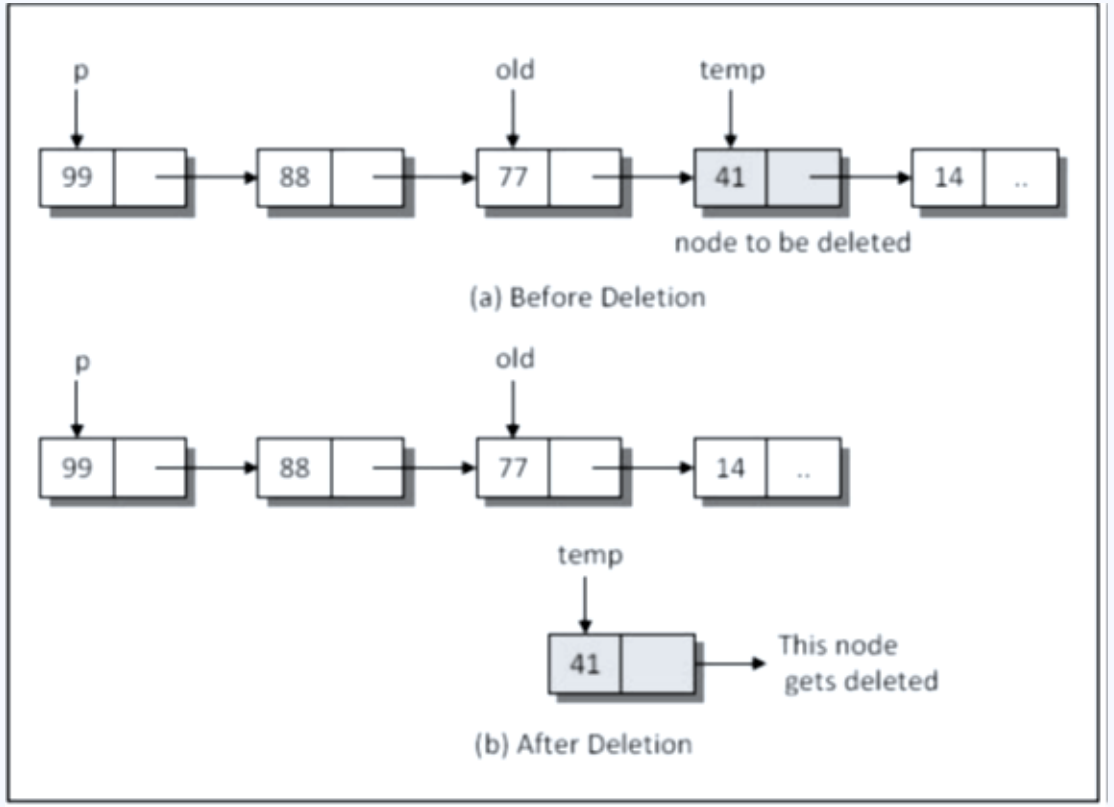
```
void linklist:: del (int num)
{
        node *old, *temp;
        temp = p;
        while (temp != NULL)
        {
                if (temp -> data == num)
                {
                // if node to be deleted is the
                // first node in the linked list
                        if (temp == p)
                                p = temp -> link;
                // delete the intermediate nodes in
                // the linked list
                        else
                                old -> link = 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.

- The **while** loop traverses through the entire linked list, checking at each node, whether it is the node to be deleted.
- If so, we have checked if the node being deleted is the first node in the linked list.
  - If it is so, we have simply shifted **p** to the next node and then deleted the earlier node.

WORKING OF DEL() FUNCTION

- The position of various pointers and links before and after the deletion is shown in Figure.



CONT. – DEALLOCATING MEMORY

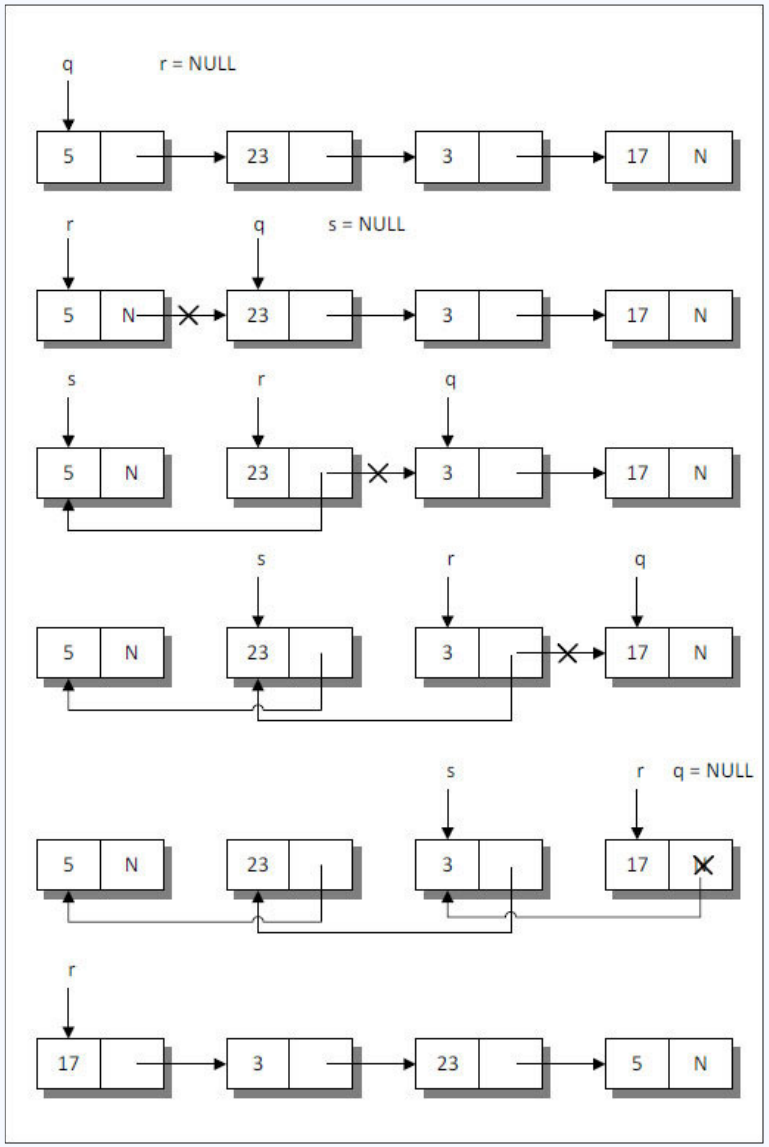
```
linklist:: ~linklist()
{
      node *q;
      while (p != NULL)
      {
            q = p -> link;
            delete p;
            p = q;
       }
 }
```

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 LIST

```
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 OPERATIONS ON LINKED LIST

- A function is called 'recursive' if a statement within the body of a function calls the same function.
- Some of the operations that are carried out on linked lists can be easily implemented using recursion.
- Recursive functions in the following code;
  - to count nodes in a linked list,
  - comparing two linked lists, and
  - cloning a linked list

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();
};
```

COUNTING THE NUMBER OF NODES IN A LINKED LIST

```
int linklist:: getlength (node *ptr)
{
      int l;
// if list is empty or if NULL is encountered
      if (ptr == NULL)
            return (0);
      else
      {
      // go to next node
            l = 1 + getlength (ptr -> link);
            return (l);
      }
}
```

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);
      }
}
```

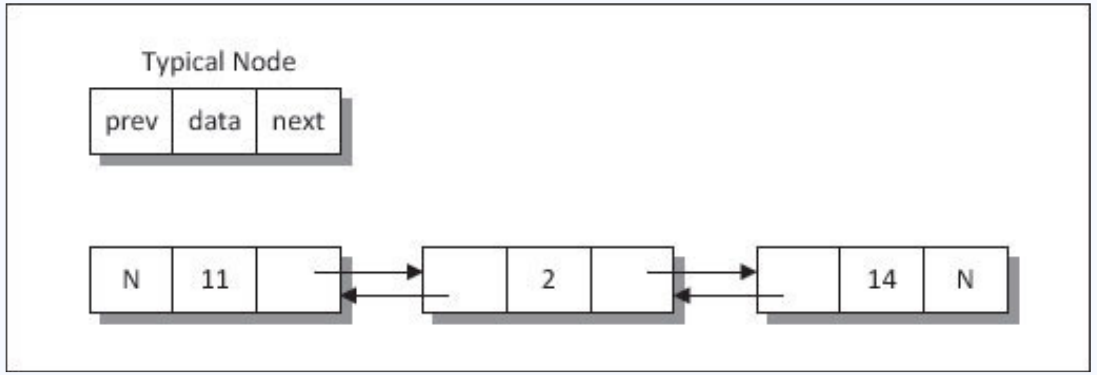
CLONING A LINKED LIST

```
// 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 LISTS

- So far. each node in the linked list provides information about where is the next node in the list.
  - It has no knowledge about where the previous node lies in memory.
  - To reach any node in the list we have to traverse the list right from the first node.

- To avoid this, we store in each node not only the address of next node but also the address of the previous node in the linked list.



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;
}
```

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;
}
```