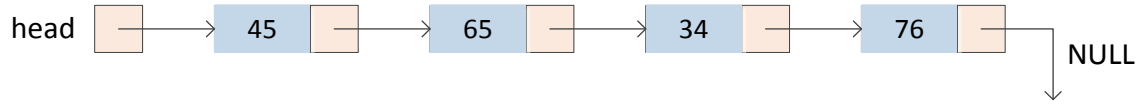
Linked List

- A linked list is a collection of components called nodes.
- In the C++ terminology, a linked list is classified as a container. In Java, it is called a collection.
- Every node (except the last node) contains the address of the next node.
- A node has two components, the data (or info) and the link (or next).

- The address of the first node in the list is stored in a separate pointer variable usually called head, first, or list.

- The null pointer (NULL, physical value 0 in C/C++) is used to denote the end of the list, or not a valid address.

Example: a linked list of 4 integers.



In typical C++ implementations, we shall define the node using struct, and the linked list is defined as a class.

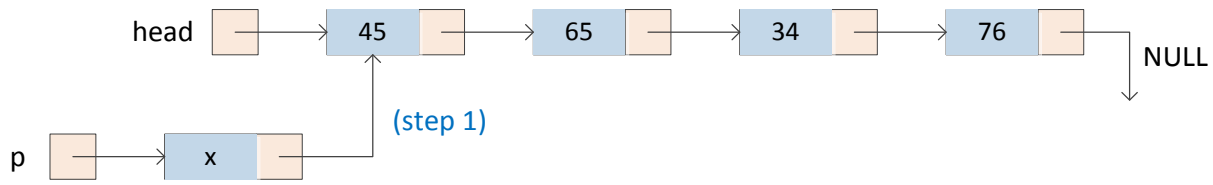
```
struct node
{
   int info;
   node *link; // another commonly used var name is "next"
};

node *head;
```

Example program codes that operate on a linked list

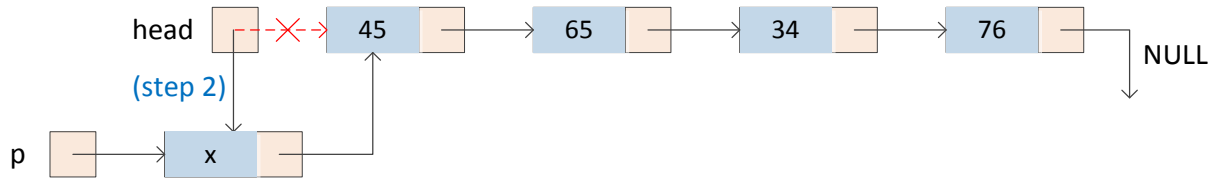
```
// To find the length of the linked list

int len = 0;
node *cur = head;  //traverse the list using cur

while (cur != NULL) // cur points to a valid node
{
   len++;
   cur = cur->link; //move to the next node
}
```

```
// Insert a new element x at the front of the list

node *p = new node;  // create the node for storing x
p->info = x;

p->link = head;  // step 1
head = p;        // step 2
```

step 1:

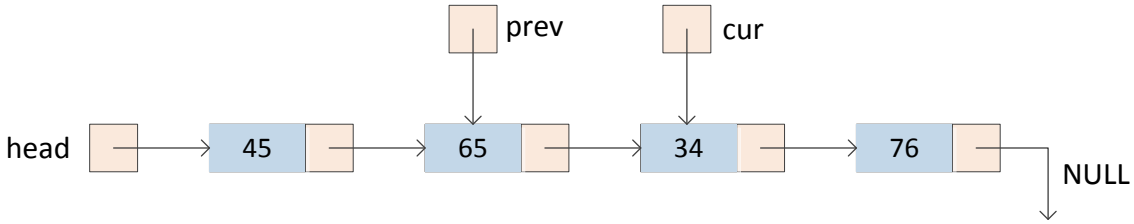


step 2:

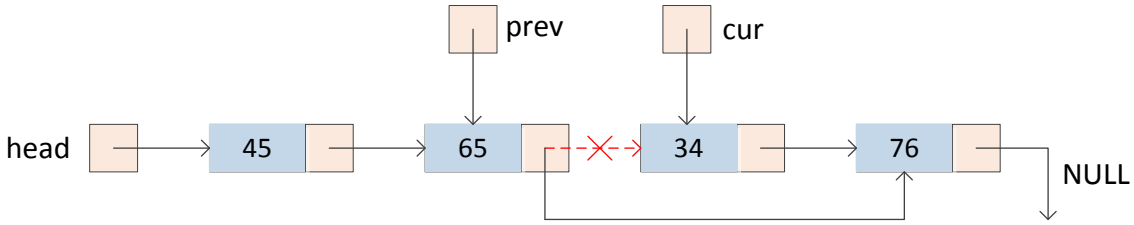


```
// Remove the element x (1st instance) from the linked list

// To remove a node from the linked list, we need to
// know the reference to its predecessor.

node *cur = head;
node *prev = NULL;
// prev points to the predecessor of cur

while (cur != NULL && cur->info != x)
{
   prev = cur;
   cur = cur->link;
}

// if cur == NULL, x is not found in the linked list

if (cur != NULL) // cur->info == x
{
   if (prev != NULL)
      prev->link = cur->link;
   else // cur is the first node in the list
      head = cur->link;

   delete cur; //free the storage of the removed node
}
```

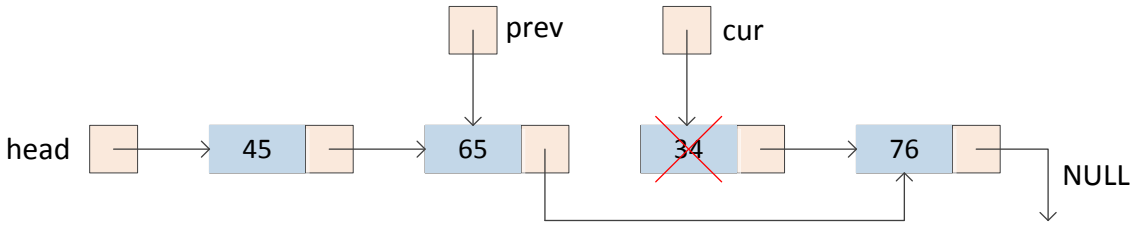
1. Locate the node storing the value x, e.g. x = 34



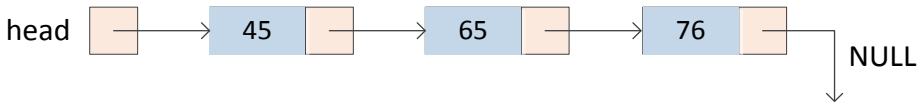
2. Update the links



3. Physically delete the node



Structure of the list after removing the element 34



```
// Insert a new element x into an ordered list

node *p = new node;
p->info = x;

if (head == NULL || x <= head->info)
{
   p->link = head;  //insert at front
   head = p;
}
else //head != NULL && x > head->info
{
   node *prev = head;
   node *cur = head->link;

   while (cur != NULL && x > cur->info)
   {
      prev = cur;
      cur = cur->link;
   }

   // insert node p after node prev
   p->link = prev->link;
   prev->link = p;
}
```

Remark:

- Null-pointer exception is a common error in programs that manipulate linked list.
- A pointer must be properly initialized or tested for not equal to NULL before you can use it to access a data member or the next node.

Implementing linked list as a class in C++

Operations that we would perform on a linked list:

- Initialize the list.
- Clear the list.
- Determine if the list is empty.
- Print the list.
- Find the length of the list.
- Make a copy of the list, e.g. assignment operator= and the copy constructor.
- Search the list for a given item.
- Insert an item to the list.
    - The requirement of the insert operation depends on the representation invariant or the intended uses of the list.
    - For ordered list, we need to maintain the ordering of list elements.
    - If it is used as a queue, insertion is performed at the rear (end of list).
    - If it is used as a stack, insertion is performed at the front.
- Remove an item from the list. Similar to the case of insertion.
- Traverse the list (in the application program that uses the linked list object), i.e. retrieve the elements one by one (in some specific order) to carry out the required computation on each node.
    - To implement the traversal, we shall make use of an iterator.
    - A linked list is a container that holds together a collection of items.
    - An iterator is an object that produces each element of a container, one at a time.
    - The two basic operations on an iterator are the dereference operator *, and the pre-increment operator ++ (advance to the next element).
- There can be other operations on the linked list, e.g. reverse the list, merge two lists, etc.

We want the linked list class to be generic such that it can be used to process different data types.

```
// file: linkedListType.h

#ifndef LINKED_LIST_H
#define LINKED_LIST_H
#include <ostream>  // use cout in the print() function

template<class Type>  //definition of node
struct nodeType
{
   Type info;
   nodeType<Type> *link;
};


template<class Type>     //The functions are simple, hence,
class linkedListIterator //they are included in the class definition
{
private:
   nodeType<Type> *current;

public:
   linkedListIterator() {  current = NULL;  }

   linkedListIterator(nodeType<Type> *ptr)
   {  current = ptr;   }

   Type operator*()
   {  return current->info;  }

   linkedListIterator<Type> operator++()  //pre-increment operator
   {
      current = current->link;
      return *this;
   }

   bool operator==(const linkedListIterator<Type>& other)
   {  return current == other.current;  }

   bool operator!=(const linkedListIterator<Type>& other)
   {  return current != other.current;  }

   //Remark: in the Java JDK, the ListIterator class has its own
   //version of insert and remove methods.
};
```

```
template<class Type>
class linkedListType
{
protected:
   int count;               // no. of elements in the list
   nodeType<Type> *first;   // pointer to the first node
   nodeType<Type> *last;    // pointer to the last node

public:
   linkedListType(); //default constructor

   linkedListType(const linkedListType<Type>& other);
   //copy constructor,
   //it is used when an object is passed to a function by value

   const linkedListType<Type>& operator=
                              (const linkedListType<Type>& other);

   ~linkedListType(); //destructor

   void initializeList();
   bool isEmpty() const;
   void print() const;
   int length() const;
   void destroyList(); //clear the list
   virtual bool search(const Type& x) const;
   virtual void insert(const Type& x);
   virtual void remove(const Type& x);

   Type remove_front(); //Remove front node and return the data value.
                        //The list must be non-empty.

   linkedListIterator<Type> begin();
   linkedListIterator<Type> end();

   //There can be additional member functions, but to simplify
   //the discussion, we will only consider the above functions.

private:
   void copyList(const linkedListType<Type>& other);
   //private function used in the copy constructor and operator=
};
```

Remark:
The constructor/mutator methods should maintain the consistency of the member variables, i.e. `count > 0`, iff `first != NULL`.

```
// Implementations of the member functions

template<class Type>
linkedListType<Type>::linkedListType()
{
   count = 0;
   first = last = NULL;
}


template<class Type>
void linkedListType<Type>::destroyList()
{  //clear the list, free the storage occupied by the nodes

   nodeType<Type> *temp;

   while (first != NULL)
   {
      temp = first;
      first = first->link;
      delete temp;
   }
   count = 0;
   last = NULL;  //first == NULL, guaranteed by the while-loop
}


template<class Type>
linkedListType<Type>::~linkedListType()
{
   destroyList();
}

template<class Type>
void linkedListType<Type>::copyList(const linkedListType<Type>& other)
{
   if (first != NULL)
      destroyList(); //clear the old contents of the list

   if (other.first == NULL) // the other list is empty
      return;

   //copy the first node
   first = new nodeType<Type>;
   first->info = other.first->info;
   first->link = NULL;
   last = first;
```

```
    //copy the remaining nodes
    nodeType<Type> *p = other.first->link;

    while (p != NULL)
    {
        last->link = new nodeType<Type>;
        last = last->link;
        last->info = p->info;
        p = p->link;
    }
    last->link = NULL;
    count = other.count;
}


template<class Type>
void linkedListType<Type>::initializeList()
{
    count = 0;
    first = last = NULL;
}


template<class Type>
linkedListType<Type>::linkedListType
                      (const linkedListType<Type>& other)
{
    initializeList();
    copyList(other);
}


template<class Type>
const linkedListType<Type>& linkedListType<Type>::operator=
                                  (const linkedListType<Type>& other)
{
    if (this != &other) //avoid self-copy
        copyList(other);

    return *this;
}

template<class Type>
bool linkedListType<Type>::isEmpty() const
{
    return count == 0; // or return first == NULL;
}
```

```
template<class Type>
void linkedListType<Type>::print() const
{
   nodeType<Type> *p;
   p = first;
   while (p != NULL)
   {
      cout << p->info << " ";
      p = p->link;
   }
   cout << endl;
}


template<class Type>
int linkedListType<Type>::length() const
{
   return count;
}


template<class Type>
bool linkedListType<Type>::search(const Type& x) const
{
   nodeType<Type> *p;

   p = first;
   while (p != NULL && p->info != x)
      p = p->link;

   return p != NULL;
}
```

```
template<class Type>
void linkedListType<Type>::insert(const Type& x)
{  //append at the end of the list

   nodeType<Type> *p = new nodeType<Type>;
   p->info = x;
   p->link = NULL;

   if (first == NULL)
      first = last = p;
   else
   {  last->link = p;
      last = p;
   }
   count++;
}


template<class Type>
void linkedListType<Type>::remove(const Type& x)
{  //remove the first instance of x in the list

   nodeType<Type> *p, *q;
   p = first;
   q = NULL;

   while (p != NULL && p->info != x)
   {
      q = p;
      p = p->link;
   }

   if (p != NULL)
   {
      if (p == first)
         first = first->link;
      else
         q->link = p->link;

      if (p == last)
         last = q;

      delete p;
      count--;
   }
}
```

```
template<class Type>
Type linkedListType<Type>::remove_front()
{  //Remove the front node and return the data value.
   //Precondition: the list must be non-empty.

   nodeType<Type> *p;
   Type x;

   p = first;
   x = first->info;  //Null-pointer exception occurs
                     //if the list is empty.

   if (first == last)
      first = last = NULL;
   else
      first = first->link;

   delete p;
   count--;

   return x;
}


template<class Type>
linkedListIterator<Type> linkedListType<Type>::begin()
{
   linkedListIterator<Type> temp(first);
   //create an iterator that references the beginning of the list

   return temp;
}


template<class Type>
linkedListIterator<Type> linkedListType<Type>::end()
{
   linkedListIterator<Type> temp(NULL);
   return temp;
}


#endif  //end of the file linkedListType.h
```

Example codes that uses a linked list

```
int main()
{
   linkedListType<int> list;
   ifstream inFile("testData.txt");

   if (!inFile.is_open())
   {
      cout << "Error: cannot open data file" << endl;
      exit(0); //terminate the program
   }

   while (!inFile.eof()) //not end of file
   {
      int i;
      inFile >> i;    //read in an integer

      if (!inFile.fail())
         list.insert(i); //insert into the linked list
      else
         break;
   }

   inFile.close();

   cout << "Contents of the list: ";
   list.print();

   //codes to compute the sum of the numbers in the list
   linkedListIterator<int> iter = list.begin();
   linkedListIterator<int> eol = list.end();  //end of list

   int sum = 0;
   while (iter != eol)
   {
      sum += *iter; //get the data value via the iterator
      ++iter;       //advance the iterator to the next item
   }

   cout << "Sum of the list = " << sum << endl;
}
```

In the above implementation of the linked list, the elements in the list are arranged in chronological order (order by the insertion time).

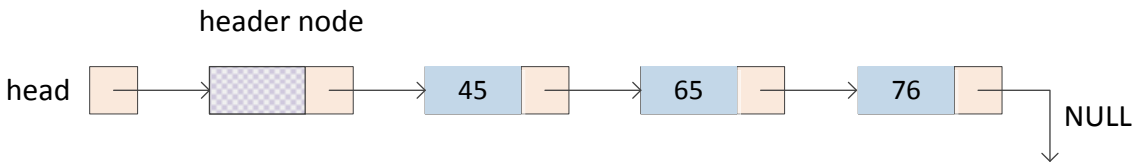
If we want the elements to be ordered by the data values, we can implement an ordered list using inheritance.

```
// file: orderedLinkedList.h

#ifndef ORDERED_LINKED_LIST_H
#define ORDERED_LINKED_LIST_H

#include "linkedListType.h"

template<class Type>
class orderedLinkedList: public linkedListType<Type>
{
   // Inherit the features linkedListType class.
   // Only need to redefine the search() and insert() functions
public:
   bool search(const Type& x) const;
   void insert(const Type& x);

};


//Implementations of the member functions

template<class Type>
bool orderedLinkedList<Type>::search(const Type& x) const
{
   nodeType<Type> *p;

   p = first;
   while (p != NULL && p->info < x)
         p = p->link;

   return (p != NULL && p->info == x);
}
```

```
template<class Type>
void orderedLinkedList<Type>::insert(const Type& x)
{  //maintain the ordering of the elements in the list

   nodeType<Type> *p = new nodeType<Type>;
   p->info = x;
   p->link = NULL;

   if (first == NULL)
      first = last = p;
   else if(x <= first->info)
   {
      p->link = first;  //insert at front
      first = p;
   }
   else //first != NULL && x > first->info
   {
      nodeType<Type> *prev = first;
      nodeType<Type> *cur = first->link;

      while (cur != NULL && x > cur->info)
      {
         prev = cur;
         cur = cur->link;
      }

      // insert node p after node prev
      p->link = prev->link;
      prev->link = p;

      if (prev == last)
         last = p;
   }
   count++;
}


#endif  //end of the file orderedLinkedList.h
```

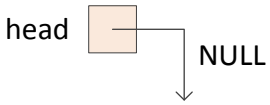
Other variants of linked list

1. Linked list with header node:

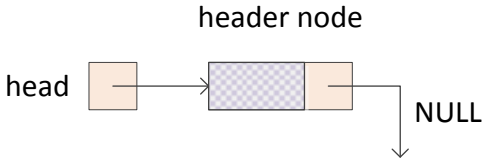


The data field of the header node is not used to store valid data. Some metadata may be stored in the header node.

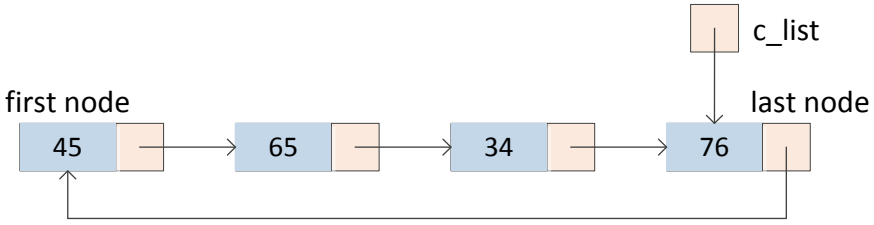
List does not exist (or not yet created):



List is empty:



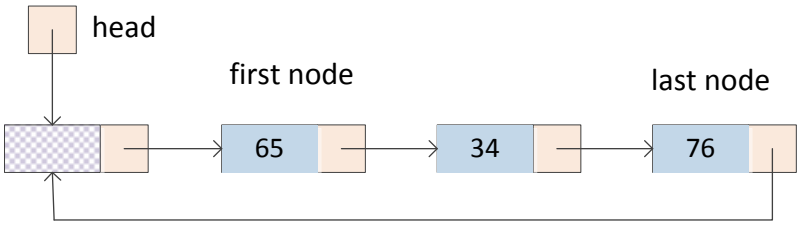
2. Circular list:
   - The `link` of the last node points back to the first node.



   - The pointer variable `c_list` points to the last node in the list.
   - We can access the first node in one step:

```
node* first = c_list->link;
```

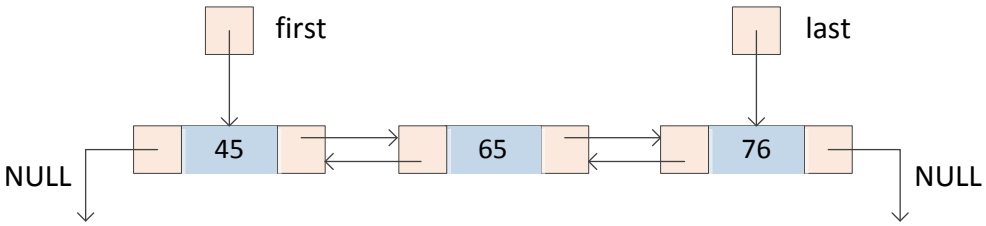
3. Circular list with header:



4. Doubly-linked list:


```
template<class Type>
struct nodeType
{
   Type info;
   nodeType<Type> *next;   //points to successor node
   nodeType<Type> *back;   //points to predecessor node
}
```

With a doubly-linked list, we can traverse the list in the forward or backward direction.



Remark: The `list` container in the C++ standard template library (STL), and the `LinkedList` class in the Java JDK are implemented as doubly-linked list.

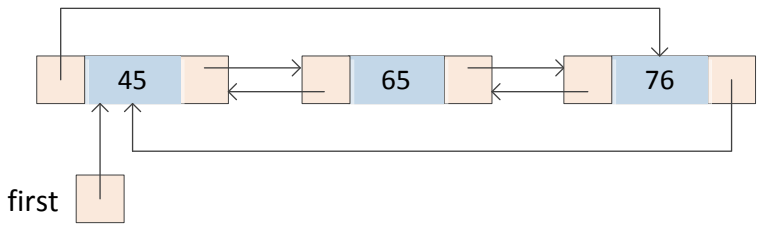
```
//Insert element x after the current node in a doubly-linked
//list

nodeType<Type> *p = new nodeType<Type>;
p->info = x;

p->next = current->next;
p->back = current;

if (current->next != NULL) //current has a successor
   current->next->back = p;

current->next = p;
```

```
//Remove node p in a doubly-linked list

if (p->next != NULL)
   p->next->back = p->back;

if (p->back != NULL)
   p->back->next = p->next;

delete p;
```

---

5. Doubly-linked list with header node

6. Circular doubly-linked list



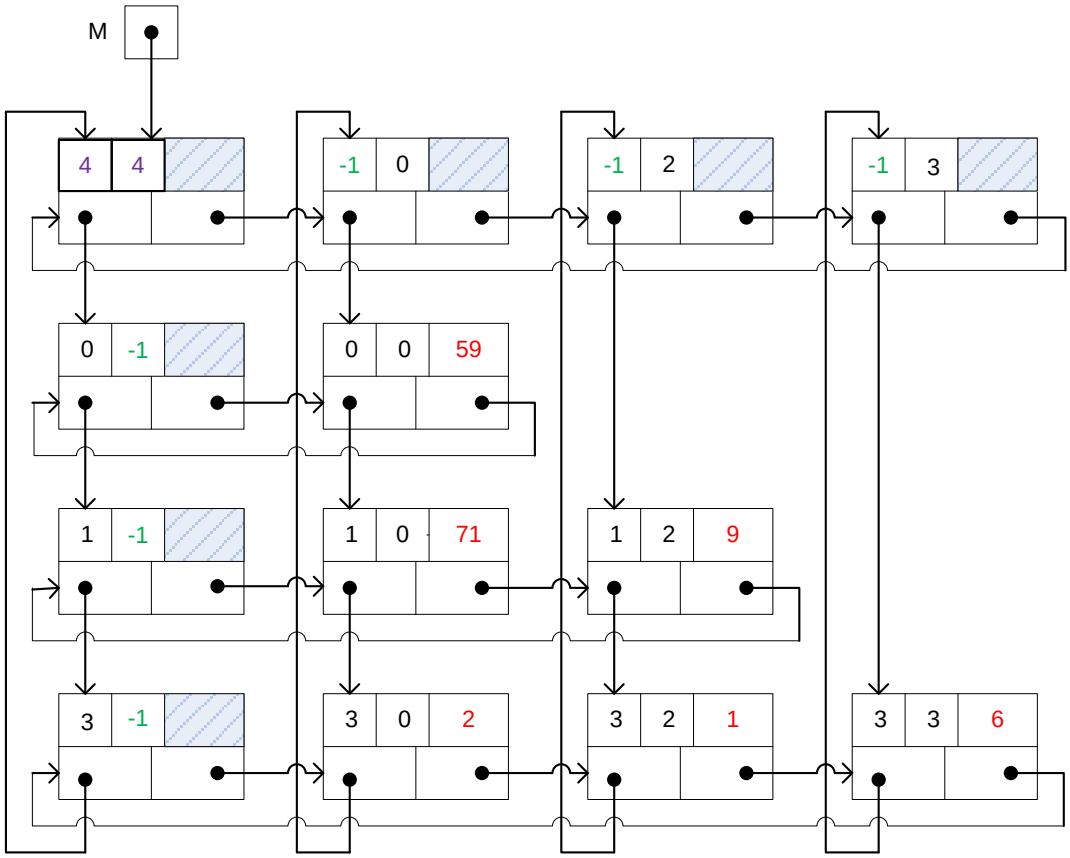
Circular doubly-linked list is used in the OS for dynamic memory allocation and de-allocation. The algorithm is called boundary-tag method.

Orthogonal list: Linked representation of sparse matrix

Non-zero elements in a row (or column) are connected in a circular linked list with header.

```
struct node {   int row, col;
                double value;
                node *down, *right;
            };

node *M;   //reference pointer to the sparse matrix
```

Example: $M = \begin{bmatrix} \mathbf{59} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{71} & \mathbf{0} & \mathbf{9} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{2} & \mathbf{0} & \mathbf{1} & \mathbf{6} \end{bmatrix}$



- In this example, values of `nRow` and `nCol` of the sparse matrix are stored in the dummy header node pointed at by `M`.
- The header of each circular list stores the respective row/column number.

```
//Return the value of matrix[i][j]
//Precondition: i < nRow and j < nCol
double getValue(node *m, int i, int j)
{
   node *p, *q;
   int found;

   found = 0;
   p = m->down;
   while (p != m && !found)
   {
      if (p->row < i)
         p = p->down;
      else if (p->row == i) //found the row
         found = 1;
      else
         p = m; //terminate the loop
   }

   if (found) //p points to header of row i
   {
      q = p->right;
      while (q != p)
      {
         if (q->col < j)
            q = q->right;
         else if (q->col == j)
            return q->value;  //value of matrix[i][j]
         else
            q = p; //terminate the loop
      }
   }

   return 0; //matrix[i][j] == 0
}
```