CHAPTER THREE
3. DATA STRUCTURES AND APPLICATIONS : LINKED LISTS

3.1. Structures

In many occasions what we want to store are not mere sequences of elements of the same data type (as in arrays), but sets of different elements with different data types. Structures offer a way of joining these heterogeneous elements together to form complex structures.

A structure is aggregate data types built using elements of primitive data types. It is a group of data elements grouped together under one name. These data elements known as *members* (*also called fields*) can have different types, some can be *int*, some can be *float*, some can be *char* and so on, and different lengths.

The difference between array and structure is the element of an array has the same type while the elements of structure can be of different type. Structures are aggregate data types built using elements of primitive data types. Another difference is that each element of an array is referred by its position while each element of structure has a unique name.

Declaration of structures

Structures are declared in C++ using the following syntax:

```
struct   structure_name
{
Member_type 1        member_name1;
Member_type 2        member_name2;
Member_type 3        member_name3;
.
.
} object_names;
```

Where *struct* is a keyword, *structure_name* is a valid identifier name for the structure type; object_name can be a set of valid identifiers for objects that have the type of this structure. Within braces { } there is a list with the data members, each one is specified with a data type and a valid identifier as its name.

For example:

```
struct student
 {
  char name [20];
  char Idd [10];
  char Department [25];
  float GPA;
 };
```

Creating structure variables

The structure variables are created in different ways.

1. We can create structure variables at the time of declaration
   Example:

```
struct student
{
 char name [20];
 char Idd [10];
 char Department [25];
 int age;
} stud1, stud2, stud3;
```

2. We can create structure variables in such a way

```
struct student
{
 char name [20];
 char Idd [10];
 char Department [25];
 int age;
} ;
struct student stud1, stud2, stud3;
```

3. We can also create structure variables as follows

```
struct student
{
 char name [20];
 char Idd [10];
 char Department [25];
 int age;
} ;

student stud1, stud2, stud3;
```

Accessing Members of Structures

We use the *dot operator* (.) to access members of a structure. We use this format to *read (accept),* to *write* (*display)* or *assign* data to members of a structure. We can assign value to the name *Stud1* as follows:

Stud1.name = "Abrham"; i.e.
Structure_variable_name . member_name;

We use also the *Arrow operator (->)* to access data members of pointer variables pointing to the structure. If we have, for example, a pointer variable for *student* structure created above like *struct student *studptr; ;* then we can set value for *name* member of the structure for this variable using *studptr -> name = "Abrham";*. Note that *studptr->name* is the same as (**studptr).name*.

Self-Referential Structures

Structures can hold pointers to instances of themselves, and such structures are called self-referential structures. Example of self-referential structure is the following.

```
struct student{
        char name[10];
        float GPA;
        student *next;
      };
```

For printing member of GPA of next variable is done in the following manner.
cout<<next->GPA; or cout<<(*next).GPA;

3.2. Singly linked lists

3.2.1. Introduction

Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data. A linked list is made up of a series of objects, called the nodes of the list that are connected by links. Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class. An additional benefit to creating a list node class is that it can be reused by the linked implementations for the stack and queue data structures presented later in the future chapters. Objects in the link class contain an element field to store the element value, and a next field to store a pointer to the next node on the list. The list built from such nodes is called a singly linked list, or a one-way list, because each list node has a single pointer to the next node on the list, i.e. each node in a singly linked list has a link only to its successor in the sequence.

The most flexible way of implementing linked structures is pointer.

A linked list is made up of a chain of nodes. Each node contains:

- the data item, and
- a pointer to the next node

A linked list, in its simplest form, is a collection of nodes that together form a linear ordering. As in the children's game "Follow the Leader," each node stores a pointer, called next, to the next node of the list. In addition, each node stores its associated element. See the following figure.

Figure 1: Example of a singly linked list of airport codes. The next pointers are shown as arrows. The null pointer is denoted by ∅.

The next pointer inside a node is a link or pointer to the next node of the list. Moving from one node to another by following a next reference is known as link hopping or pointer hopping. The first and last nodes of a linked list are called the *head* and *tail* of the list, respectively. Thus, we can link-hop through the list, starting at the head and ending at the *tail*. We can identify the tail as the node having a null next reference.

Array vs Linked lists

Arrays are *simple* and *fast but we m*ust specify their size at construction time. This has its own drawbacks. If you construct an array with space for *n*, tomorrow you may need *n*+1. Here comes a need for a more flexible system.

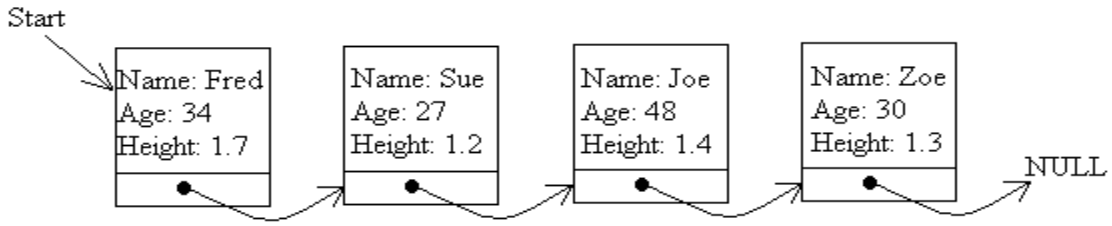
Arrays are nice and simple for storing things in a certain order. However, they are not very adaptable. For instance, we have to fix the size n of an array in advance, which makes resizing an array difficult. Insertions and deletions are difficult because elements need to be shifted around to make space for insertion or to fill empty positions after deletion.

Array has at least two limitations: (1) its size has to be known at compilation time and (2) the data in the array are separated in computer memory by the same distance, which means that inserting an item inside the array requires shifting other data in this array. This limitation can be overcome by using linked structures. Flexible space use by dynamically allocating space for each element as needed. This implies that one need not know the size of the list in advance. Memory is efficiently utilized.

Like an array, a singly linked list maintains its elements in a certain order, as determined by the chain of next links. Unlike an array, a singly linked list does not have a predetermined fixed size. It can be resized by adding or removing nodes.

3.2.2. Creating Linked Lists in C++

A linked list is a data structure that is built from structures and pointers. It forms a chain of "nodes" with pointers representing the links of the chain and holding the entire thing together. A linked list can be represented by a diagram like this one:



This linked list has four nodes in it, each with a link to the next node in the series. The last node has a link to the special value ***NULL***, which any pointer (whatever its type) can point to, to show that it is the last link in the chain. There is also another special pointer, called ***Start*** (also called ***Head***), which points to the first link in the chain so that we can keep track of it. It is important to note that ***head*** is not a node, rather the address of the first node of the list.

The key part of a linked list is a structure, which holds the data for each node (the name, address, age or whatever for the items in the list), and, most importantly, a pointer to the next node. Here we have given the structure of a typical node:

```
struct node
{
  char name[20];    // Name of up to 20 letters
  int age
  float height;     // In metres
  node *next;// Pointer to next node
 };
struct node *start_ptr = NULL;
```

The important part of the structure is the line before the closing curly brackets. This gives a pointer to the next node in the list. This is the only case in C++ where you are allowed to refer to a data type (in this case **node**) before you have even finished defining it!

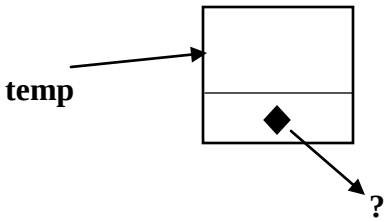
We have also declared a pointer called **start_ptr** that will permanently point to the start of the list. To start with, there are no nodes in the list, which is why **start_ptr** is set to *NULL*.

3.2.3. Adding a node to the list

The first problem that we face is how to add a node to the list. For simplicity's sake, we will assume that it has to be added to the end of the list, although it could be added anywhere in the list (a problem we will deal with later on).

Firstly, we declare the space for a pointer item and assign a temporary pointer to it. This is done using the **new** statement as follows:

```
temp = new node;
```



We can refer to the new node as ***temp**, i.e. **"the node that temp points to"**. When the fields of this structure are referred to, brackets can be put round the ***temp** part, as otherwise the compiler will think we are trying to refer to the fields of the pointer. Alternatively, we can use the arrow pointer notation. That's what we shall do here.

Having declared the node, we ask the user to fill in the details of the person, i.e. the name, age, address or whatever:

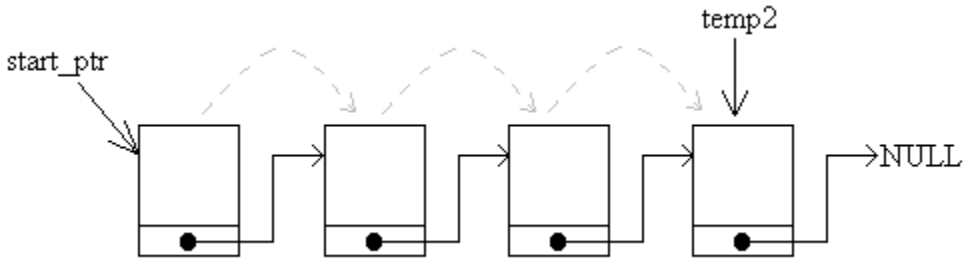
```
cout<< "Please enter the name of the person: ";
cin>> temp->name;
cout<< "Please enter the age of the person : ";
cin>> temp->age;
cout<< "Please enter the height of the person : ";
cin>> temp->height;
```

temp->next = NULL;

The last line sets the pointer from this node to the next to *NULL*, indicating that this node, when it is inserted in the list, will be the last node. Having set up the information, we have to decide what to do with the pointers. Of course, if the list is empty to start with, there's no problem - just set the Start pointer to point to this node (i.e. set it to the same value as temp):

```
if (start_ptr == NULL)
start_ptr = temp;
```

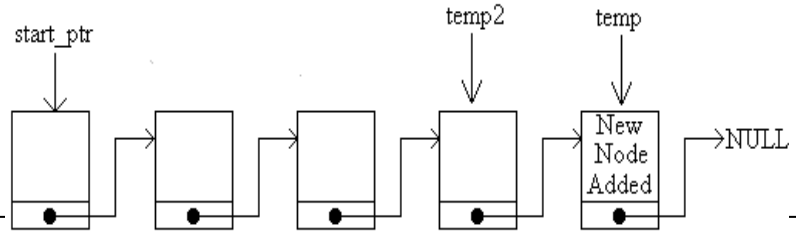
It is harder if there are already nodes in the list. In this case, the secret is to declare a second pointer, **temp2**, to step through the list until it finds the last node.




```
temp2 = start_ptr;
  // We know this is not NULL - list not empty!
while (temp2->next != NULL)
 {
   temp2 = temp2->next;   // Move to next link in chain
 }
```

The loop will terminate when **temp2** points to the last node in the chain, and it knows when this happened because the **next** pointer in that node will point to *NULL*. When it has found it, it sets the pointer from that last node to point to the node we have just declared:

temp2->next = temp;



The link **temp2->next** in this diagram is the link joining the last two nodes. The full code for adding a node at the end of the list is shown below, in its own little function:

```
void add_node_at_end ( )
{
   node *temp, *temp2;   // Temporary pointers
    // Reserve space for new node and fill it with data
   temp = new node;
   cout<< "Please enter the name of the person: ";
   cin>> temp->name;
   cout<< "Please enter the age of the person : ";
   cin>> temp->age;
   cout<< "Please enter the height of the person : ";
   cin>> temp->height;
   temp->next = NULL;
    // Set up link to this node
     if (start_ptr = = NULL)
       start_ptr = temp;
     else
      {
        temp2 = start_ptr;
        // We know this is not NULL - list not empty!
       while (temp2->next != NULL)
         {
          temp2 = temp2->next; // Move to next link in chain
         }
        temp2->next = temp;
     }
 }
```

3.2.4. Displaying the list of nodes

Having added one or more nodes, we need to display the list of nodes on the screen. This is comparatively easy to do. Here is the method:

1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to *NULL*, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the **next** pointer of the node it is currently indicating.
5. Jump back to step 2.

The temporary pointer moves along the list, displaying the details of the nodes it comes across. At each stage, it can get hold of the next node in the list by using the **next** pointer of the node it is currently pointing to. Here is the C++ code that does the job:

```
temp = start_ptr;
 do
  {
    if (temp = = NULL)
      cout << "End of list" << endl;
    else
     {
       // Display details for what temp points to
        cout << "Name : " << temp->name << endl;
        cout << "Age : " << temp->age << endl;
        cout << "Height : " << temp->height << endl;
         // Move to next node (if present)
         temp = temp->next;
       }
   } while (temp != NULL);
```

Check through this code, matching it to the method listed above. It helps if you draw a diagram on paper of a linked list and work through the code using the diagram.

3.2.5. Navigating through the list

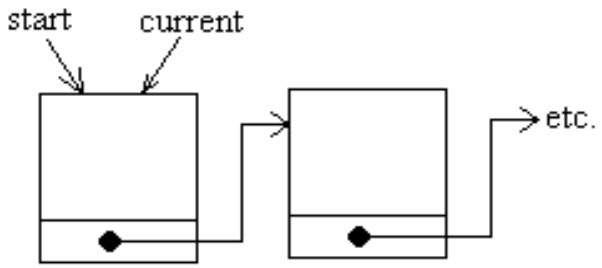
One thing you may need to do is to navigate through the list, with a pointer that moves backwards and forwards through the list, like an index pointer in an array. This is certainly necessary when you want to insert or delete a node from somewhere inside the list, as you will need to specify the position.

We will call the mobile pointer **current**. First of all, it is declared, and set to the same value as the **start_ptr** pointer:

```
node *current;

current = start_ptr;
```

Notice that you don't need to set *current* equal to the address of the *start* pointer, as they are both pointers. The statement above makes them both point to the same thing:



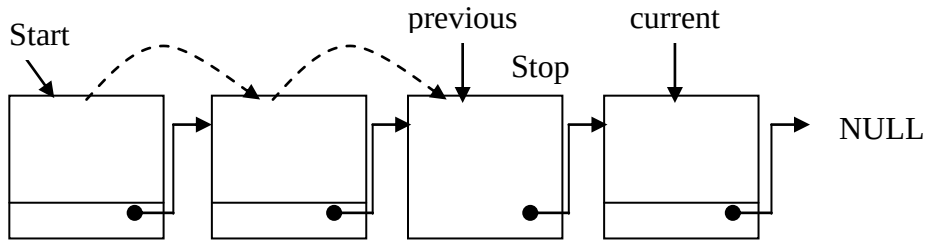
It's easy to get the current pointer to point to the next node in the list (i.e. move from left to right along the list). If you want to move current along one node, use the next field of the node that it is pointing to at the moment:

```
current = current->next;
```

In fact, we had better check that it isn't pointing to the last item in the list. If it is, then there is no next node to move to:

```
if (current->next = = NULL)
    cout << "You are at the end of the list." << endl;
else
    current = current->next;
```

Moving the current pointer back one step is a little harder. This is because we have no way of moving back a step automatically from the current node. The only way to find the node before the current one is to start at the beginning, work our way through and stop when we find the node before the one we are considering at the moment. We can tell when this happens, as the next pointer from that node will point to exactly the same place in memory as the current pointer (i.e. the current node).



First of all, we had better check to see if the current node is also the first one. If it is, then there is no "previous" node to point to. If not, check through all the nodes in turn until we detect that we are just behind the current one.

```
if (current = = start_ptr)
    cout << "You are at the start of the list" << endl;
else
  {
   node *previous;     // Declare the pointer
    previous = start_ptr;

    while (previous->next != current)
       {
          previous = previous->next;
       }
    current = previous;
  }
```

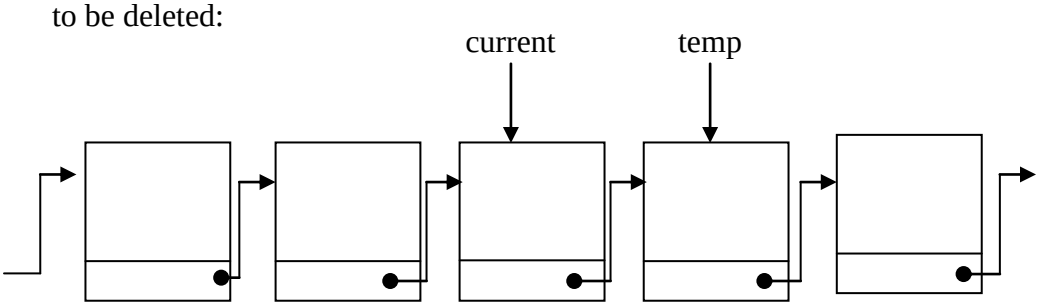
The else clause translates as follows: Declare a temporary pointer (for use in this else clause only). Set it equal to the start pointer. All the time that it is not pointing to the node before the current node, move it along the line. Once the previous node has been found, the current pointer is set to that node - i.e. it moves back along the list.

Now that you have the facility to move back and forth, you need to do something with it. Firstly, let's see if we can alter the details for that particular node in the list:

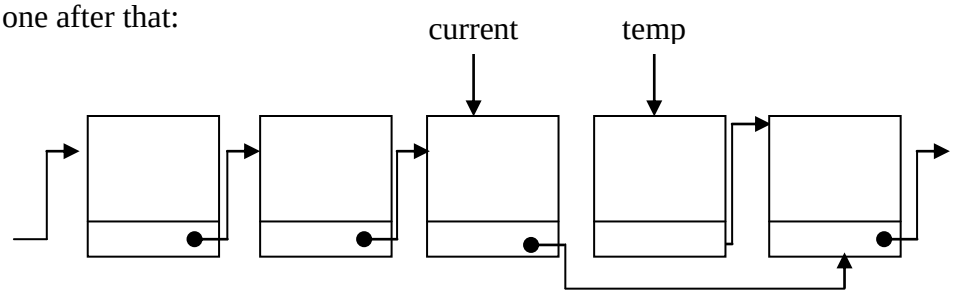
```
cout << "Please enter the new name of the person: ";
cin >> current->name;
cout << "Please enter the new age of the person : ";
cin >> current->age;
cout << "Please enter the new height of the person : ";
cin >> current->height;
```

The next easiest thing to do is to ***delete*** a node from the list directly after the current position. We have to use a temporary pointer to point to the node to be deleted. Once this node has been "anchored", the pointers to the remaining nodes can be readjusted before the node on death row is deleted. Here is the sequence of actions:

1. First, the temporary pointer is assigned to the node after the current one. This is the node to be deleted:



2. Now the pointer from the current node is made to leap-frog the next node and point to the one after that:



3. The last step is to delete the node pointed to by **temp**.

Here is the code for deleting the node. It includes a test at the start to test whether the current node is the last one in the list:

```
if (current->next = = NULL)
    cout << "There is no node after current" << endl;
else
  {
   node *temp;
   temp = current->next;
   current->next = temp->next;     // Could be NULL
   delete temp;
  }
```

Here is the code to ***add*** a node after the current one. This is done similarly, but we haven't illustrated it with diagrams:

```
if (current->next  = = NULL)
   add_node_at_end();
else
    {
     node *temp;
     new temp;
     get_details(temp);
     // Make the new node point to the same thing as the current node
     temp->next = current->next;
     // Make the current node point to the new link in the chain
     current->next = temp;
      }
```

We have assumed that the function **add_node_at_end()** is the routine for adding the node to the end of the list that we created near the top of this section. This routine is called if the current pointer is the last one in the list so the new one would be added on to the end.

Similarly, the routine **get_temp(temp)** is a routine that reads in the details for the new node similar to the one defined just above.

... and so ...

3.2.6. Deleting a node from the list

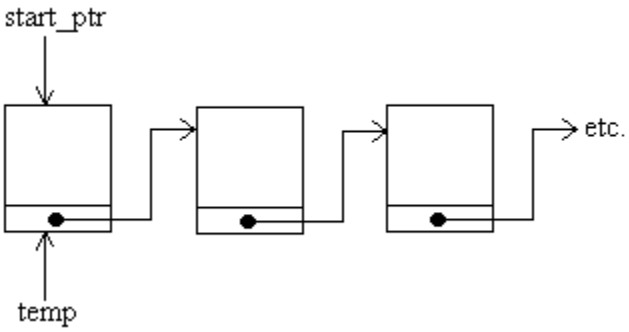
When it comes to deleting nodes, we have three choices: Delete a node from the start of the list, delete one from the end of the list, or delete one from somewhere in the middle. For simplicity, we shall just deal with deleting one from the start or from the end.

When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the **delete** instruction:

```
delete temp;     // Release the memory pointed to by temp
```

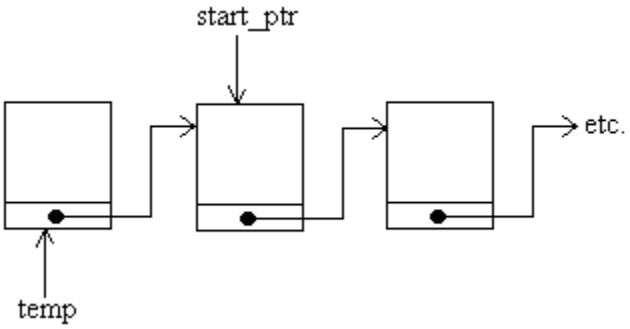
However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. Here is how we go about deleting the first node in the linked list:

temp = start_ptr; // Make the temporary pointer identical to the start pointer

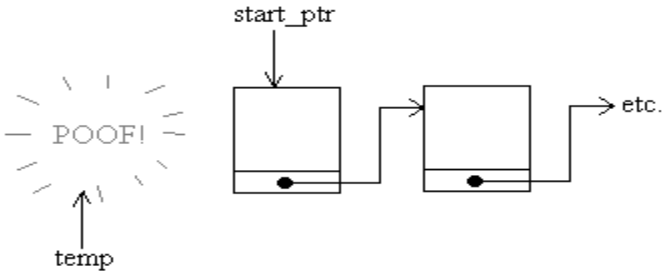


Now that the first node has been safely tagged (so that we can refer to it even when the start pointer has been reassigned), we can move the start pointer to the next node in the chain:

start_ptr = start_ptr->next; // Second node in chain.



delete temp; // Wipe out original start node



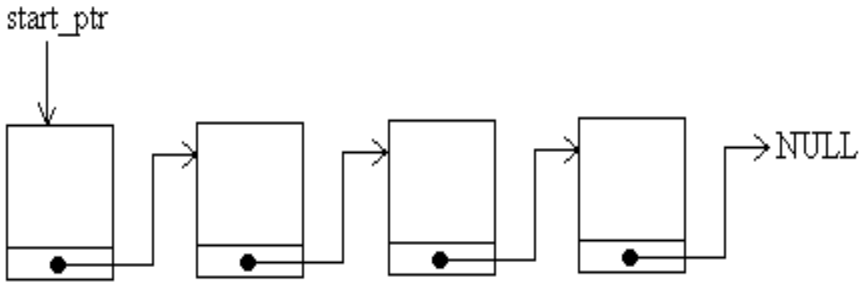
Here is the function that deletes a node from the start:

```
void delete_start_node()
  {
    node *temp;
    temp = start_ptr;
    start_ptr = start_ptr->next;
    delete temp;
  }
```

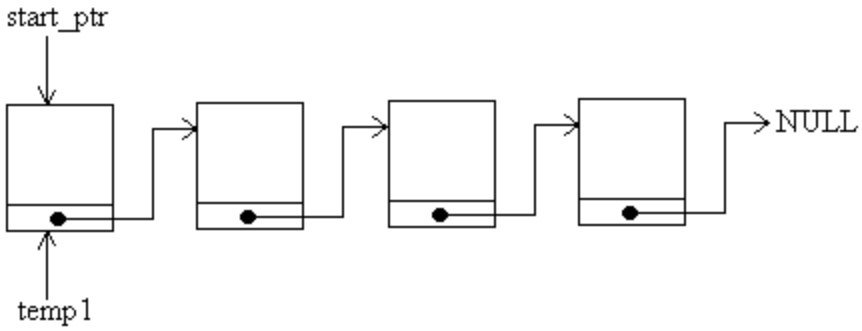
Deleting a node from the end of the list is harder, as the temporary pointer must find where the end of the list is by hopping along from the start. This is done using code that is almost identical to that used to insert a node at the end of the list. It is necessary to maintain two temporary pointers, **temp1** and **temp2**. The pointer **temp1** will point to the last node in the list and **temp2** will point to the previous node. We have to keep track of both as it is necessary to delete the last node and immediately afterwards, to set the **next** pointer of the previous node to **NULL** (it is now the new last node).

1. Look at the start pointer. If it is NULL, then the list is empty, so print out a "No nodes to delete" message.
2. Make **temp1** point to whatever the start pointer is pointing to.
3. If the **next** pointer of what **temp1** indicates is NULL, then we've found the last node of the list, so jump to step 7.
4. Make another pointer, **temp2**, point to the current node in the list.
5. Make **temp1** point to the next item in the list.
6. Go to step 3.
7. If you get this far, then the temporary pointer, **temp1**, should point to the last item in the list and the other temporary pointer, **temp2**, should point to the last-but-one item.
8. Delete the node pointed to by **temp1**.
9. Mark the **next** pointer of the node pointed to by **temp2** as NULL - it is the new last node.

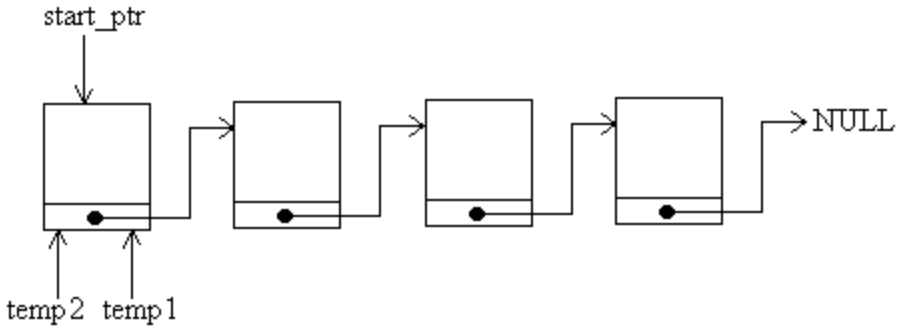
Let's try it with a rough drawing. This is always a good idea when you are trying to understand an abstract data type. Suppose we want to delete the last node from this list:



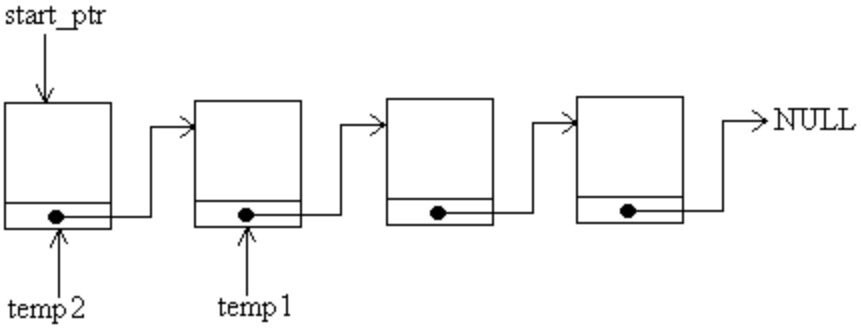
First, the start pointer doesn't point to NULL, so we don't have to display " No nodes to delete " message. Let's get straight on with step2 - set the pointer **temp1** to the same as the start pointer:



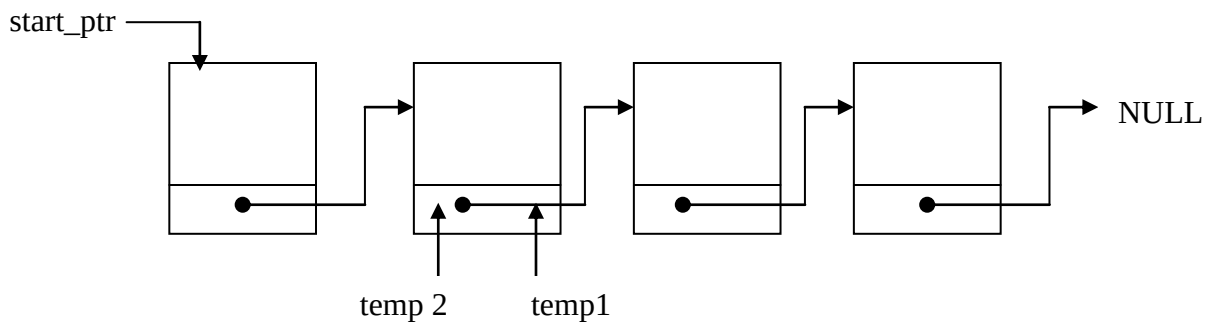
The **next** pointer from this node isn't NULL, so we haven't found the end node. Instead, we set the pointer **temp2** to the same node as **temp1** as shown below.



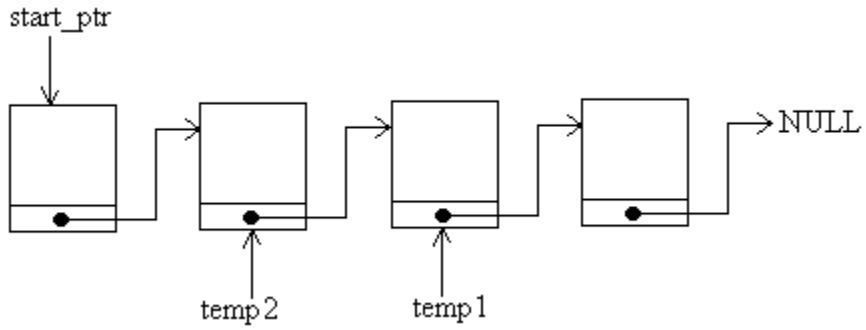
and then move temp1 to the next node in the list:



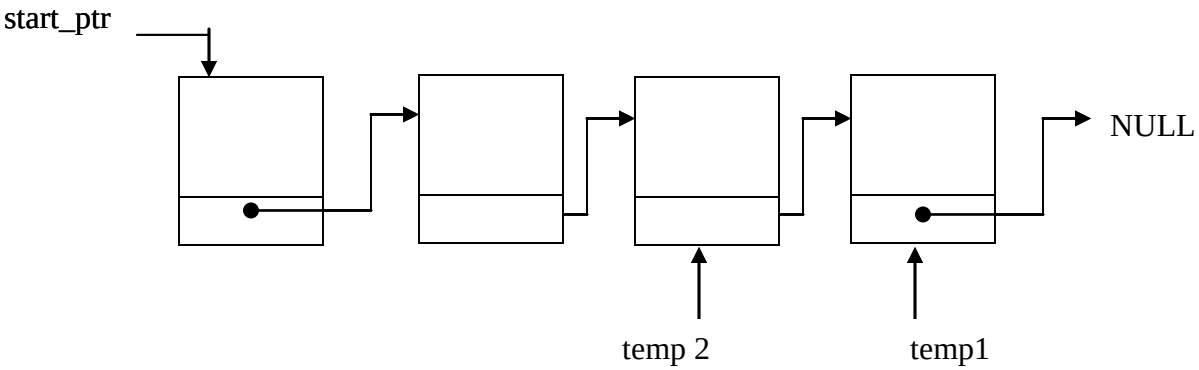
Going back to step 3, we see that **temp1** still doesn't point to the last node in the list, so we make **temp2** point to what **temp1** points to



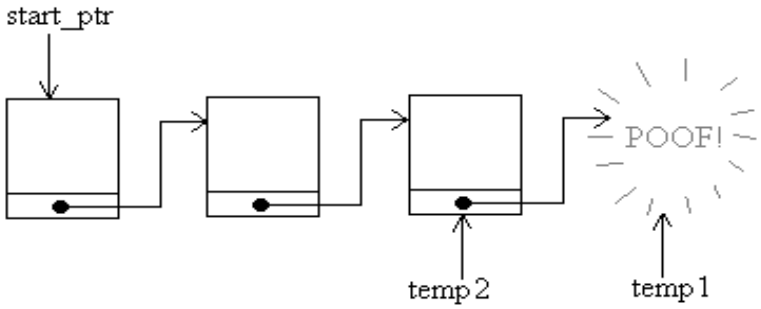
and temp1 is made to point to the next node alone:



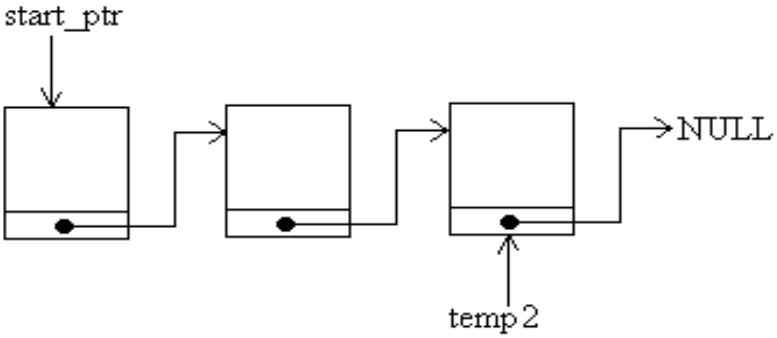
Eventually, this goes on until **temp1** really is pointing to the last node in the list, with **temp2** pointing to the second last node:



Now we have reached step 8. The next thing to do is to delete the node pointed to by **temp1**



and set the **next** pointer of what **temp2** indicates to NULL as indicated below:



We suppose you want some code for all that! All right then,

```
void delete_end_node ( )
  {
    node *temp1, *temp2;
    if (start_ptr = = NULL)
        cout << "The list is empty!" << endl;
    else
      {
        temp1 = start_ptr;
        while (temp1->next ! = NULL)
          {
            temp2 = temp1;
            temp1 = temp1->next;
          }
        delete temp1;
        temp2->next = NULL;
      }
  }
```

The code seems a lot shorter than the explanation!

Now, the sharp-witted amongst you will have spotted a problem. If the list only contains one node, the code above will malfunction. This is because the function goes as far as the **temp1 = start_ptr** statement, but never gets as far as setting up **temp2**. The code above has to be adapted so that if the first node is also the last (has a NULL **next** pointer), then it is deleted and the **start_ptr** pointer is assigned to NULL. In this case, there is no need for the pointer **temp2**:

```
void delete_end_nod e( )
  {
    node *temp1, *temp2;
    if (start_ptr = = NULL)
        cout << "The list is empty!" << endl;
    else
      { temp1 = start_ptr;
        if (temp1->next = = NULL)     // This part is new!
          { delete temp1;
            start_ptr = NULL;
          }
        else
      { while (temp1->next ! = NULL)
              { temp2 = temp1;
                temp1 = temp1->next;
              }
            delete temp1;
            temp2 -> next = NULL;
          }
      }
  }
```

3.3. Doubly Linked Lists

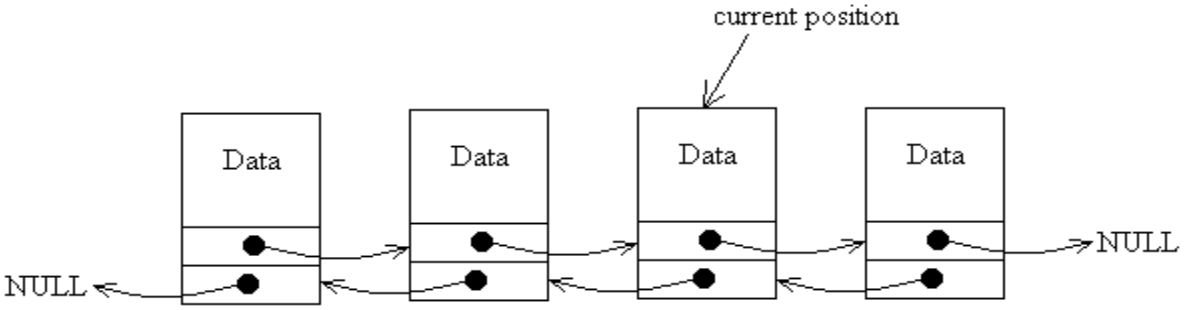
3.3.1. Introduction

As we saw in the previous section, removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node immediately preceding the one we want to remove. There are many appli-cations where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the doubly linked list. In addition to its element member, A node in a doubly linked list stores two references—a ***next*** link which points to the next node in the list, and a ***prev*** link, which points to the previous node in the list. Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

If you've mastered how to do singly linked lists, then it shouldn't be much of a leap to doubly linked lists. A doubly linked list is one where there are links from each node in both directions. We can traverse in both directions in a doubly linked list as each and every node of a double linked list contains address of next node along with address of previous node also. Thus a node in a doubly linked list contains three fields.

i. Data field
ii. Address of next node
iii. Address of previous node



You will notice that each node in the list has two pointers, one to the next node and one to the previous one - again, the ends of the list are defined by NULL pointers. Also there is no pointer to the start of the list. Instead, there is simply a pointer to some position in the list that can be moved left or right, called **current** pointer.

The reason we needed a start pointer in the ordinary linked list is because, having moved on from one node to another, we can't easily move back, so without the start pointer, we would lose track

of all the nodes in the list that we have already passed. With the doubly linked list, we can move the current pointer backwards and forwards at will.

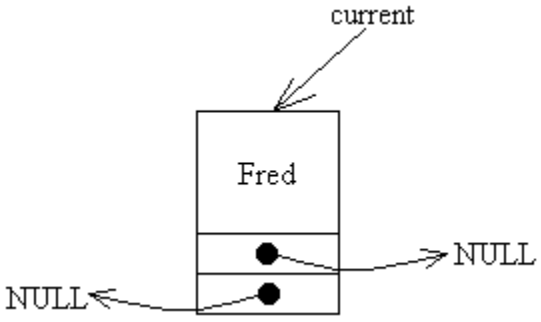
The most common reason to use a doubly linked list is because it is easier to implement than a singly linked list. While the code for the doubly linked implementation is a little longer than for the singly linked version, it tends to be a bit more "obvious" in its intention, and so easier to implement and debug. Member functions for processing doubly linked lists are slightly more complicated than their singly linked counterparts because there is one more pointer member to be maintained. Removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node in front of the one we want to remove.

3.3.2. Creating Doubly Linked Lists

The nodes for a doubly linked list would be defined as follows:

```
struct node{
            char name[20];
            node *next;    // Pointer to next node
            node *prev;    // Pointer to previous node
             };
   node *current;
current = new node;
current -> name = "Fred";
current -> next = NULL;
current -> prev = NULL;
```

We have also included some code to declare the first node and set its pointers to NULL. It gives the following situation:



We still need to consider the directions 'forward' and 'backward', so in this case, we will need to define functions to add a node to the start of the list (left-most position) and the end of the list (right-most position).

3.3.3. Inserting Node to a Doubly Linked List

Because of its double link structure, it is possible to insert a node at any position within a doubly linked list. To add a node to a list, the node has to be created, its data members properly initialized, and then the node needs to be incorporated into the list.

Move the current pointer to the *start* (*head*) node in the doubly linked list, and then inserting a node at the start of a doubly linked list involves the following steps.

1. A new node is created, and make the data field of the node with a value being inserted.
2. Make the *previous* address of the new node to *NULL*.
3. Make the *next* address of the node to the value of *head* so that this member points to the first node in the list. But now, the new node should become the first node; therefore,
4. Head is set to point to the new node. But the new node is not yet accessible from its sucessor; to rectify this,
5. The *previous* member of the successor of the new node is set to point to the new node

The algorithm for inserting a node to the start of a doubly linked list is the following.

```
void Add_Node_At_Start ( String new_name)
 {
  // Declare a temporary pointer and move it to the start
   node *temp = current;
   while (temp -> prev ! = NULL)
      temp = temp -> prev;
  // Declare a new node and link it in
    node *temp2;
    temp2 = new node;
    temp2 -> name = new_name;  // Store the new name in the node
    temp2 -> prev = NULL;       // This is the new start of the list
    temp2 -> next = temp;       // Links to current list
    temp -> prev = temp2;
 }
```

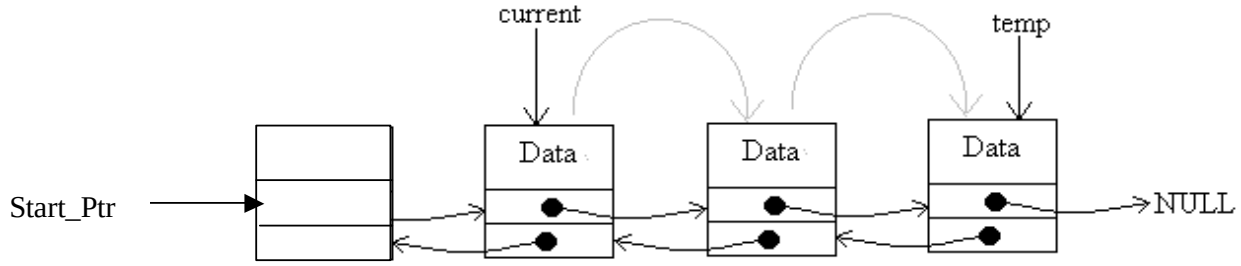
Move the current pointer of the list to the last node (*tail*), then inserting a node at the end of a doubly linked list consists of the following actions:

1. A new node is created, and then initialize its data filed with value being inserted,
2. Set the *next* member of the node to *NULL*,
3. Set the *previous* member to the value of *tail* so that this member points to the last node in the list. But now, the new node should become the last node; therefore,
4. Make *tail* to point to the new node. But the new node is not yet accessible from its predecessor; to rectify this,
5. Set the *next* member of the predecessor is set to point to the new node.

The algorithm for inserting a new node at the end of a doubly linked list is the following.

```
void Add_Node_At_End (String  new_name)
 {
   // Declare a temporary pointer and move it to the end
   node *temp = current;
   while (temp -> next ! = NULL)
      temp = temp -> next;
    // Declare a new node and link it in
     node *temp2;
     temp2 = new node;
     temp2 -> name = new_name;  // Store the new name in the node
     temp2 -> next = NULL;       // This is the new start of the list
     temp2 -> prev = temp;       // Links to current list
     temp -> next = temp2;
  }
```

Here, the new name is passed to the appropriate function as a parameter. We'll go through the function for adding a node to the right-most end of the list. The method is similar for adding a node at the other end. Firstly, a temporary pointer is set up and is made to march along the list until it points to last node in the list.



After that, a new node is declared, and the name is copied into it. The next pointer of this new node is set to NULL to indicate that this node will be the new end of the list. The prev pointer of the new node is linked into the last node of the existing list. The next pointer of the current end of the list is set to the new node.

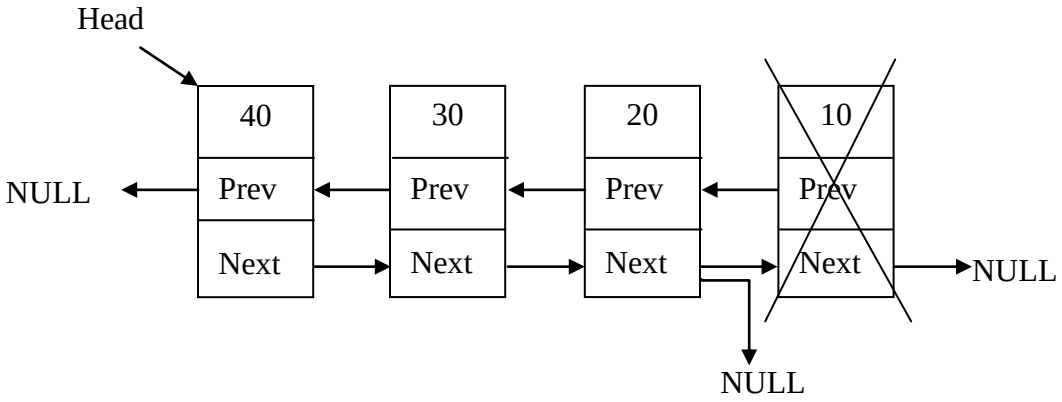
Inserting a node in the middle of a doubly linked list is also possible. Given a node *v* of a doubly linked list (which could possibly be the header, but not the trailer), let *z* be a new node that we wish to insert immediately after *v*. Let *w* be the node following *v*, that is, *w* is the node pointed to by *v*'s next link. To insert *z* after *v*, we link it into the current list, by performing the following operations:

- Make *z*' s *prev* link point to *v*
- Make *z*'s *next* link point to *w*
- Make *w*'s *prev* link point to *z*
- Make *v*'s *next* link point to *z*

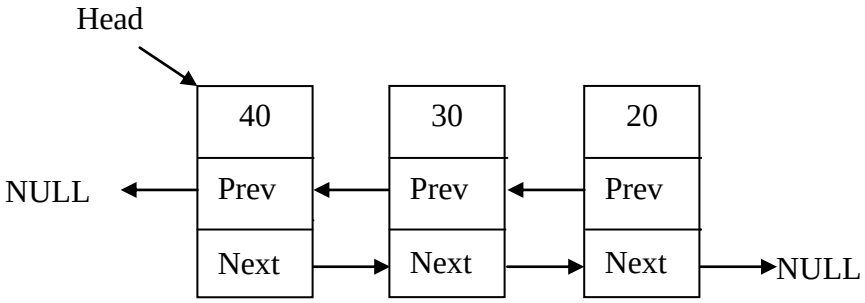
3.3.4. Deleting a Node from Doubly Linked List

When it comes to deleting nodes, we have three choices: Delete a node from the start (front end) of the list, delete one from the end (tail end) of the list, or delete one form somewhere in the middle.

Here let us see to delete node from the end of a double linked list (for others workout by your on). Deleting the last node from the doubly linked list is straight forward because there is direct access from the last node to its predecessor. When deleting the last node from the list, the temporary variable is set to the value in the node, then *tail* is set to its predecessor, and the last and now redundant node is deleted. In this way, the *next* to last node becomes the last node. The *next* member of the *tail* node is a dangling reference; therefore, *next* is set to *NULL*.



After deleting the last node, the doubly linked list looks the following.



The algorithm for deleting the last element from a doubly linked list is the following.

```
void Delete_Node_From_End ( )
 {
   // Declare a temporary pointer and move it to the end
   node *temp = current;
     while (temp -> next ! = NULL)
          temp = temp -> next;
  tail = temp -> prev;
  delete temp;
  tail -> next = NULL;
}
```

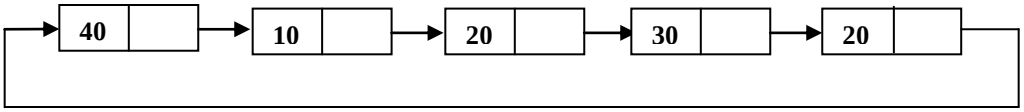
Likewise, it is easy to remove a node *v* in the middle of a doubly linked list. We access the nodes *u* and *w* on either side of *v* using *v*'s *Prev* and *Next* values (these nodes must exist). To remove node *v*, we simply have *u* and *w* point to each other instead of to *v*. We refer to this operation as the linking out of *v*. We also *NULL* out *v*'s *prev* and *next* pointers so as not to retain old references into the list.

- Make *u*'s *next* link point to *w* (i.e. u -> next = v -> next)
- Make *w*' s *prev* link point to *u* (i.e. w -> prev = v -> prev)
- Delete node *v*

3.4. Circular Linked Lists

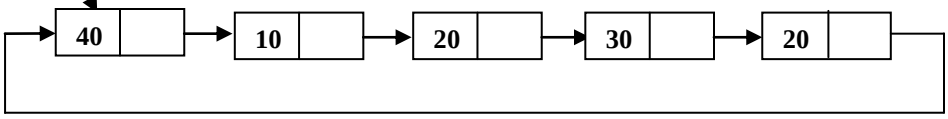
The next fields in the last node contain a pointer back to the first node rather than NULL pointer, make last node point to the first (instead of null). The pointer from the last element in the list points back to the first element, such lists are known as a circular list. A circular list is when nodes form a ring: the list is finite and each node has a successor. A circular list does not have a natural "first" or "last" node.

- A circular singly linked list without header node

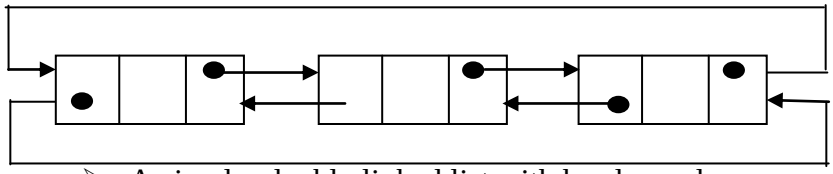


Head

- A circular singly linked list with header node



- A circular doubly linked list without header node



- A circular doubly linked list with header node