CS 2 : C++ Data Structures
Topic 6: Linked Lists
Author: Satish Singhal Ph. D.
Version 1.01

In topics, two and three we discussed a fixed storage implementation of stacks and queues. In those implementations, the maximum size of a stack or queue was fixed which was assigned at the compile-time. The static arrays are another fixed size data structures. The fixed size implementation is constrained in three different ways.

1. If number of elements in the (fixed size) data structure are much smaller than the maximum size then program becomes wasteful of the memory.
2. On the other hand, at times the fixed size implementation may fall short, when the number of data items may exceed the maximum size allotted for that implementation. In case of arrays we got around this problem by using the dynamic array, which when filled, grows by certain size. However, even there, the time taken to copy an array from old size to new size slows down the execution algorithm.
3. The array implementation of data structure will need the contiguous memory allocation in the memory space. When the memory becomes fragmented, then a contiguous block of memory may become unavailable.

The linked structures can avoid all of the above limitations. In linked structures, every element in the structure stores a reference or pointer to the next element in the structure. We also call such structure a dynamic structure as they can grow and shrink dynamically and have capacity limited, only by the memory available from the operating system. This last characteristic provides more flexibility to dynamic or linked structures because their capacity does not have to be known in advance. Fixed structures such as arrays are of course faster than the dynamic structures such as linked lists (because of their random access nature), but they are less flexible.

Basic Linked List

C++ does not allow a class to be its own member. For example the declaration given below will result in a compile error!

```
class MyClass{
        ItemType Data;
        MyClass Next; // Compile Error
};
```

On the other hand, a pointer to a class can be a member of that class. For example, the below is allowed:

```
class MyClass{
        ItemType Data;
        MyClass * Next;
};
```

The declaration such as above becomes a basis for constructing a dynamically growing list, where each member has two fields. The first field called Data, may hold some C++ or user

defined data object. The second field, which is a pointer to a MyClass type object, can hold the address of a MyClass type pointee. This allows each member in the list to hold the address of the next member in the list. Obviously, the last member of the list will hold the null address in its Next field.

Typically each member of linked list is called a Node. A Node class to create an integer linked list is given below, first in UML form and later in code form:

Node Class
The class diagram of Node class is shown below:

FIG. 1
Node class[1] has two data members, data, that is int type and self-referential pointer Node * called next. It has a default/explicit constructor and a toString function usable to print the data part of Node. In RAM typical Nodes would look like below (Figures 2A, and 2B).

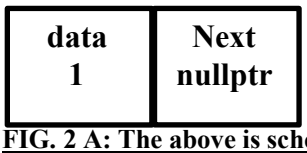


FIG. 2 A: The above is schematic picture of last node in the linked list. Last node must have the Next field pointing to null pointer or nullptr.

Note: Understand that in past 0, NULL, '\0' have been used for null pointers in C++. Use of all of those is now replaced by C++ 11 nullptr. Please do not use any of older literals for null pointer.

When a node is inside a linked list, its Next field would have address of the next link in the list. Figure 2B shows that construct.

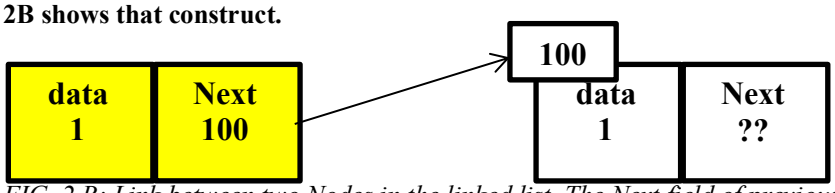


FIG. 2 B: Link between two Nodes in the linked list. The Next field of previous Node stores the memory address of next node. For example, the yellow node (see electronic copy) has the Next field storing 100, that is the memory address of the successor node.

Next field of following node may store nullptr if it is last node, or address of next linked list member.

---

[1]For sake of convenience we will call structs, classes and unions using word class. The code will differentiate between them.

The code of Node class in file Node.h is given below: (include guards are omitted).

```
#include <string>
using namespace std;

struct Node{
        int data; // Data part of the node. A list can have more than one data parts
        Node * Next;//Self-referential pointer
//Constructor just uses the initialization list. Default and explicit constructors are combined
        Node(int val = 0, Node * ptr = nullptr) :data(val), Next(ptr)
        {     }
//toString function returns the string form of the class member data.
        const string toString() const {
                return (to_string(this->data) + "    ");
        }
};
```

Now we show the linked list class that uses the above Node struct.

Class IntLinkedList

The UML diagram of linked list class that uses Node class is given below:

Figure 3. IntLinkedList class

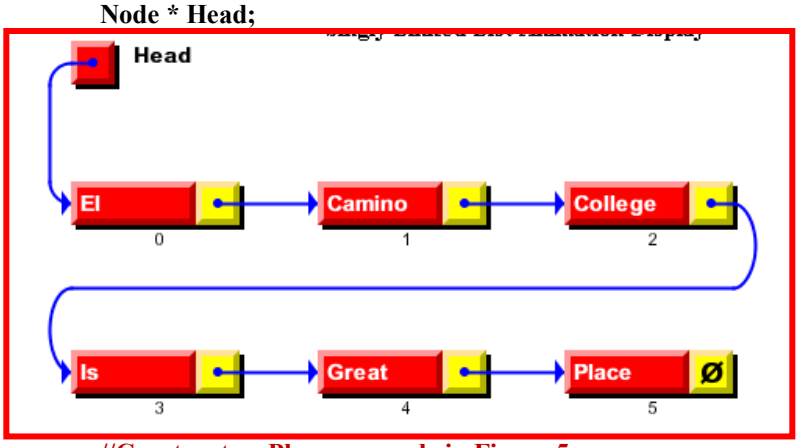
The file IntLinkedList.h below shows the data and other members of this class. Explanation is provided next to or above the class members.

```
#include "Node.h"
#include <iostream>
using namespace std;



struct IntLinkedList {
```

```
/Head node. Head node is the first node in the linked list. Picture below shows the linked list with
//Head node shown. The linked list in picture is a string linked list with words stored
//and shown in each node in background. Integer linked list would have integers in data fields instead
//of string.
        Node * Head;
```




```
        //Constructor. Please see code in Figure 5.
        IntLinkedList();
        //Rule of three implementation
        //A. Copy function. Please see code in Figure 5.
        void copy(const  IntLinkedList & Other);
        //1. Copy constructor. Please see code in Figure 5.
        IntLinkedList(const  IntLinkedList & Other);
        //B. destroy function. Please see code in Figure 5.
        void destroy();
        //2. Assignment operator. Please see code in Figure 5.
        IntLinkedList& operator =(const IntLinkedList & Other);
        //3. Destructor. Please see code in Figure 5.
        virtual ~IntLinkedList();
        //toString function. Please see code in Figure 5.
        const string toString() const;
        //Insert function. Inserts a node in the linked list. Please see code in Figure 5.
        void insert(int val);
};
```

FIG. 4 Header file for Linked List class to store integers. Understand that what is stored in a linked list is dependent upon the data field in its Node class.

Figure 5 below gives the implementation file for the IntLinkedList class.

```
#include "IntLinkedList.h"
/*
Notice that since IntLinkedList class has only one data member (called Head), only that needs
initialization. Such initialization left to the client is risky. That is why only the default constructor
is allowed and constructor is not overloaded. The default constructor creates a so-called dummy
node. The dummy node stores a character 'A' indicating that the data field of dummy node is
not the user data. [Understand that if a data field is an int type, one can hard code and store a
character in it ]. However, the presence of character data in dummy node indicates that this
node does not store user data. The code
new Node ('A', nullptr) ;
calls the constructor of Node class.

Question1. : Why would a constructor call such as Node ('A', nullptr) compile?
```

```
The constructor sets the Next field of dummy Head node to nullptr. That will be replaced later
by the address of next node in the list, when list grows.
*/
IntLinkedList::IntLinkedList() : Head(new Node('A', nullptr)) {
        cout << "From constructor.\n";
}
//---------------------------------------------------------------
/*
Copy function is a helper function to assist making deep copy of the linked list. This function is
called from inside of copy constructor and assignment operator (both of which are required as
per rule of three).
*/
void IntLinkedList::copy(const  IntLinkedList & Other) {
        Node *Current = Other.Head->Next;
        Node *Iterator = Head = new Node(0, nullptr); // create dummy node

        while (Current != nullptr){
                // create a copy of node pointed to by Current, and link up
                Iterator->Next = new Node(Current->data, nullptr); // ??

                // update pointers
                Current = Current->Next;
                Iterator = Iterator->Next;
        }
}
//---------------------------------------------------------------
//Copy Constructor
IntLinkedList::IntLinkedList(const  IntLinkedList & Other){
        cout << "From Copy constructor.\n";
        if (this == &Other){
                cout << "Self copying not allowed. Exiting program.\n";
                exit(0);
        }
        copy(Other);
}
//---------------------------------------------------------------
//Destructor
void IntLinkedList::destroy(){
        Node * Current = Head; // start with dummy node
        while (Current != nullptr){
                // do something
                Node *save = Current;
                Current = Current->Next;
                delete save;
        }
}
//---------------------------------------------------------------

//Overloaded assignment operator.
```

```
const IntLinkedList& IntLinkedList::operator = (const IntLinkedList & Other) {
	cout << "From assignment operator.\n";
	if (this != & Other)
	{
		destroy();
		copy(Other);
	}

	return *this;
}
//-----------------------------------------------------------------
IntLinkedList::~IntLinkedList(){
	cout << "From destructor.\n";
	destroy();
}
//-----------------------------------------------------------------------
const string IntLinkedList::toString() const{
	if (!(Head == nullptr)){
		string text = "";

		Node * Iterator = Head->Next;
		//Traverse the linked list using Iterator and add all data to text

		while (Iterator != nullptr){
			text += (Iterator->toString());
			Iterator = Iterator->Next;
		}
		return (text + '\n');
	}
	else{
		throw "Exception thrown";
	}
}
//-----------------------------------------------------------------
void IntLinkedList::insert(int val){
	Head->Next = new Node(val, Head->Next);
}
//-----------------------------------------------------------
```

Discussion of IntLinkedList class members

Here we discuss various members of IntLinkedList class for the better understanding of their functioning.

Constructor

The constructor uses an initialization list where pointer to head node is given a dynamic memory allocation as below:

Head(new Node('A', nullptr))

Notice that data type of head not is not integer type. Linked lists such as this, are called dummy head node lists because the head node is not part of user generated list. It is rather present as

means to simplify the management of the linked list. The data field of the dummy node may be set to some sentinel value that is not confused as user data. Using a character (such as 'A' in this case), for an integer linked list, fulfills that requirement.

Since memory allocation is done inside the constructor, and then member function such as an insert would add more dynamically allocated nodes to linked list, the rule of three requites that our IntLinkedList class has following programmer defined components, to make deep copies and avoid memory leaks:
1. Copy Constructor
2. Assignment Operator
3. Destructor

To aid the overall process we have added two helper functions: copy, and destroy. Copy function is called from inside of copy constructor and assignment operator. The destroy function is called from inside of assignment operator and destructor. Main purpose of these functions is code reuse.

Copy Function[2]

The copy function makes a deep copy of the IntLinkedList passed to it by reference into the current object. Imagine that Figure A.1 below shows the pre-built linked list that is passed to the copy function.

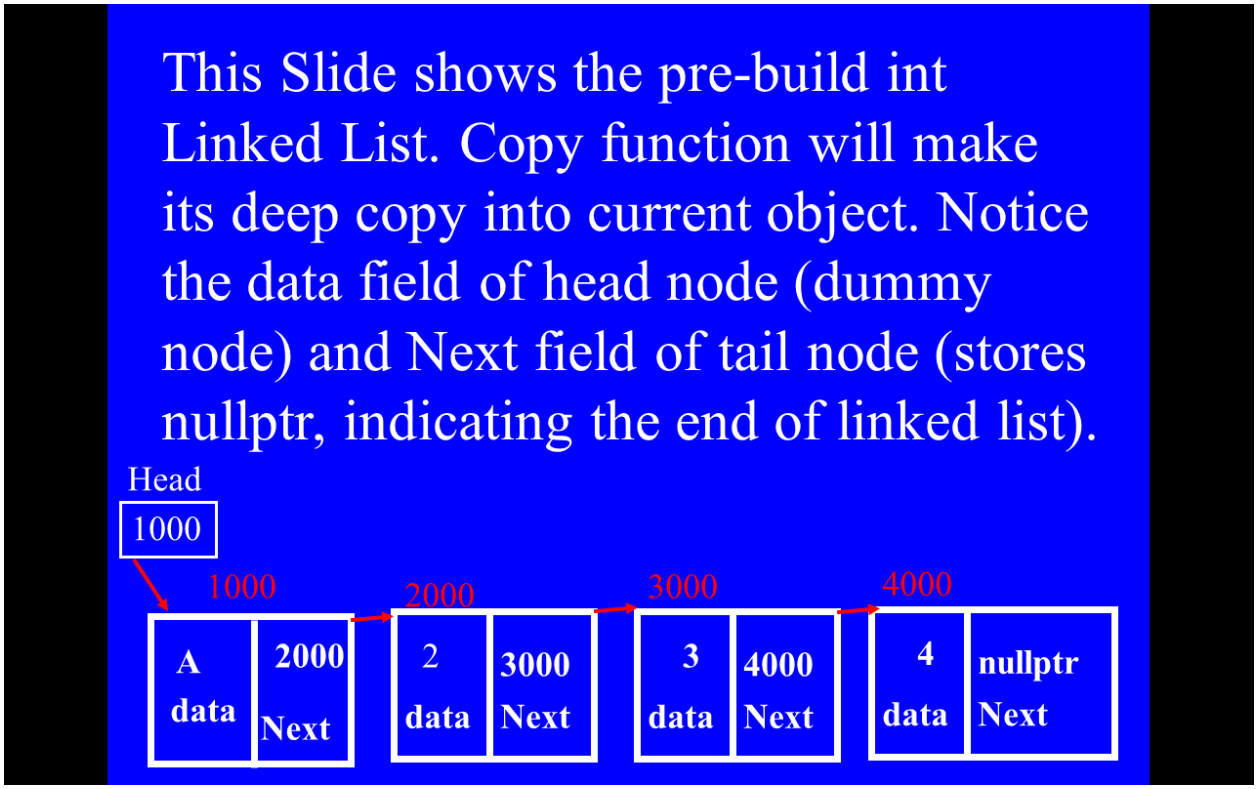


FIG. A.1

[2] Video and PowerPoints describing copy function are available. Video is at link: https://www.youtube.com/watch?v=XylL78AV-SE

The pointer head stores the address of dummy node (1000). Next field of dummy node stores the address of first user node (2000) and so on. Next field of last node stores a nullptr, which is C++11 null value for pointers. The Figure A.2 shows the copy function and our plan to explain its code and how it makes deep copy of the IntLinkedList.

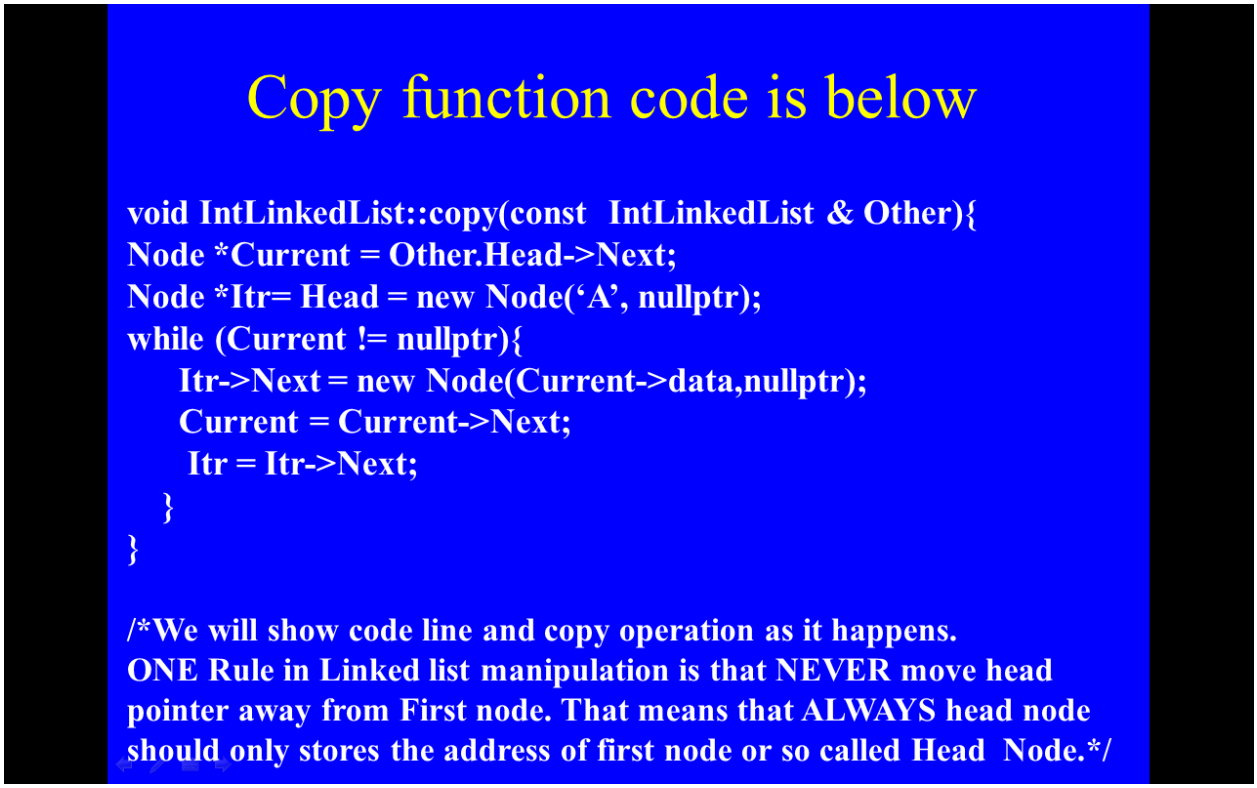


FIG. A.2

One rule in linked list manipulation is **NEVER MOVE THE POINTER, pointing to the head node! Once you do that, you have lost your linked list and it becomes a memory leak.** Now we do a code walk through the copy function code. Figure A.3 shows an already built linked list that is passed by reference to copy function. First significant code line in copy function is shown in black, where a local pointer current is assigned to copy the same address as the address stored in Head->Next pointer of the linked list called Other, that copy function must copy.

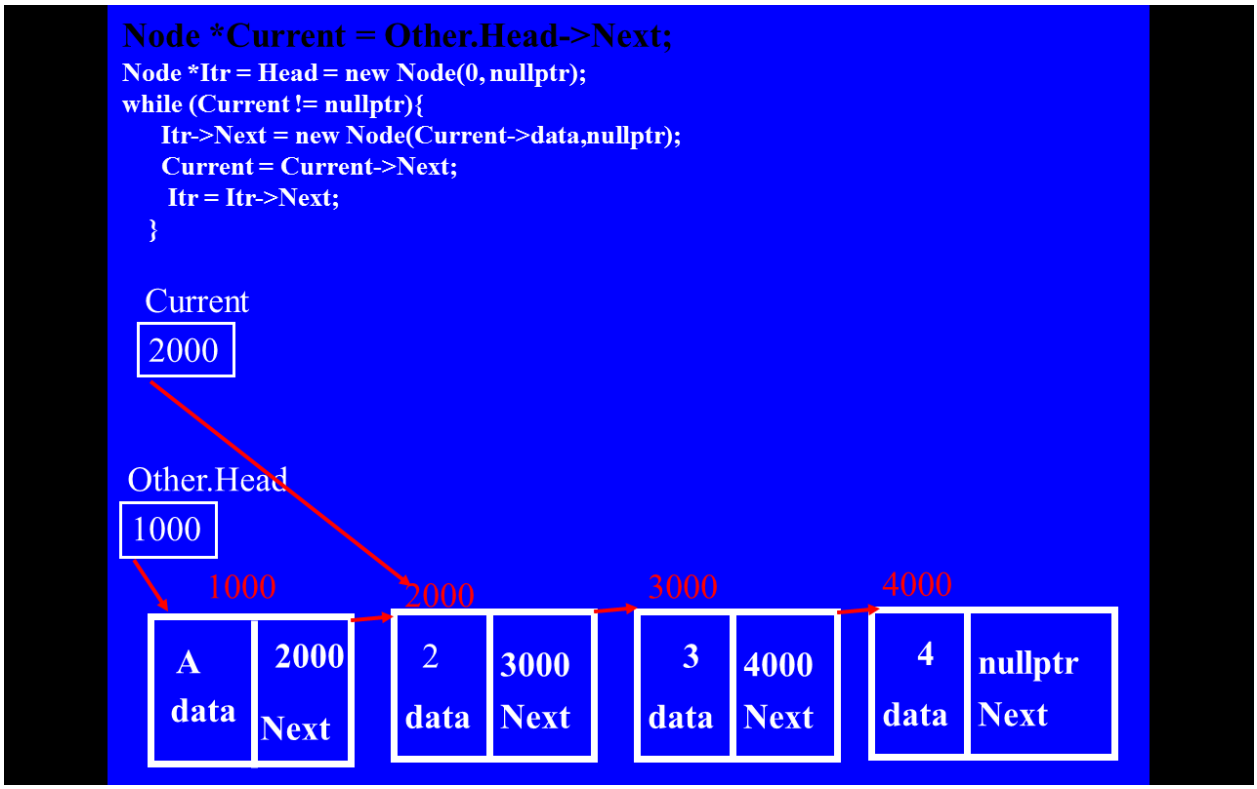


FIG. A.3

Thus the pointer Current points to the node after the dummy node. This kind of initialization will be used often because it allows us to create a starting point for the traversal of the linked list that would need copying, (and in coding of future functions such as printing of list etc.).

Then first order of business is to create a dummy head for the new linked list that is to be a deep copy of linked list called Other. This is done by an additional pointer called Itr. The Head data member of target linked list is also created by the code line shown in dark (Figure A.4).

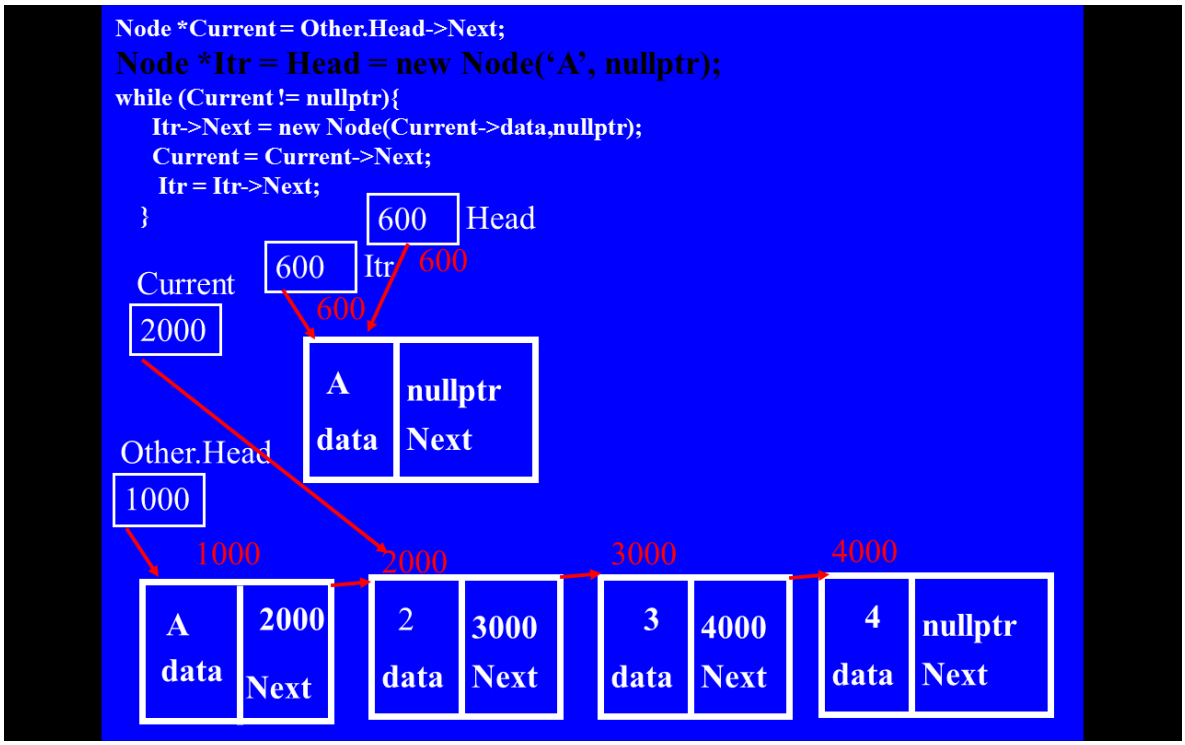


FIG. A.4

Copying of the Other linked list is done through the while loop. As long as the Other linked list has more than one node, the loop will be entered because then the pointer Current would have a non-null value (Figure A.5).

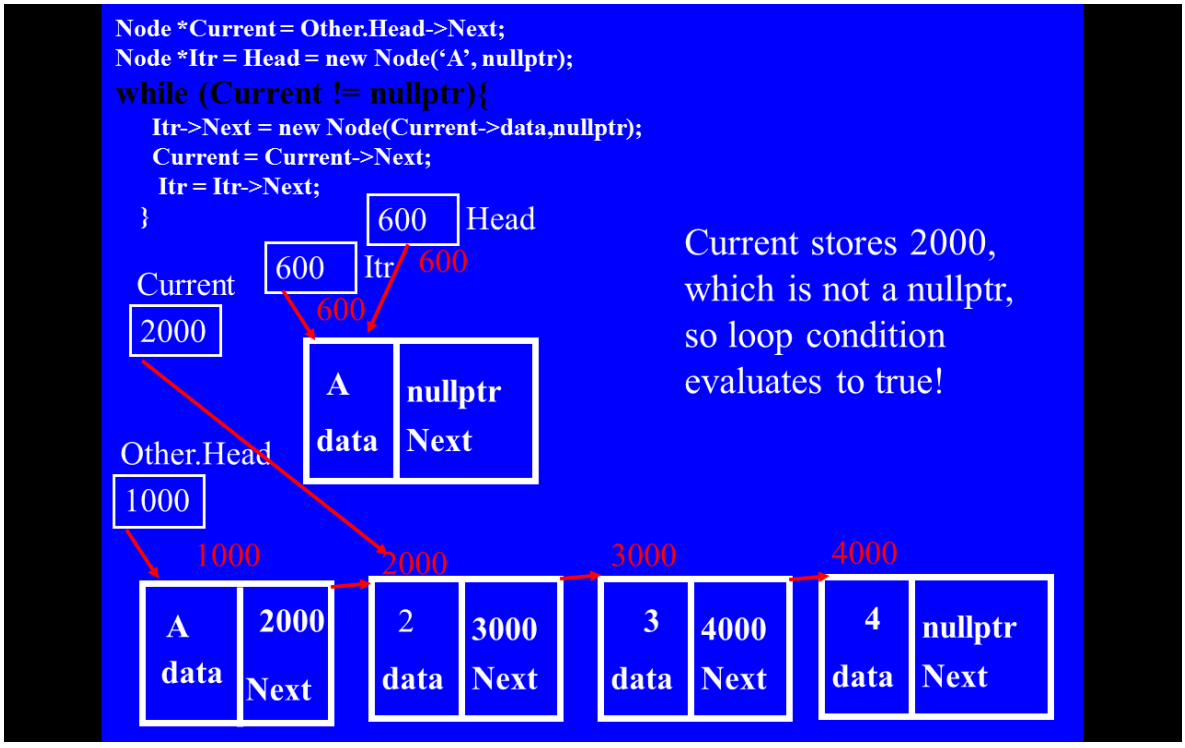


FIG. A.5

Copy operation is performed the first code line in the loop (Figure A.6).

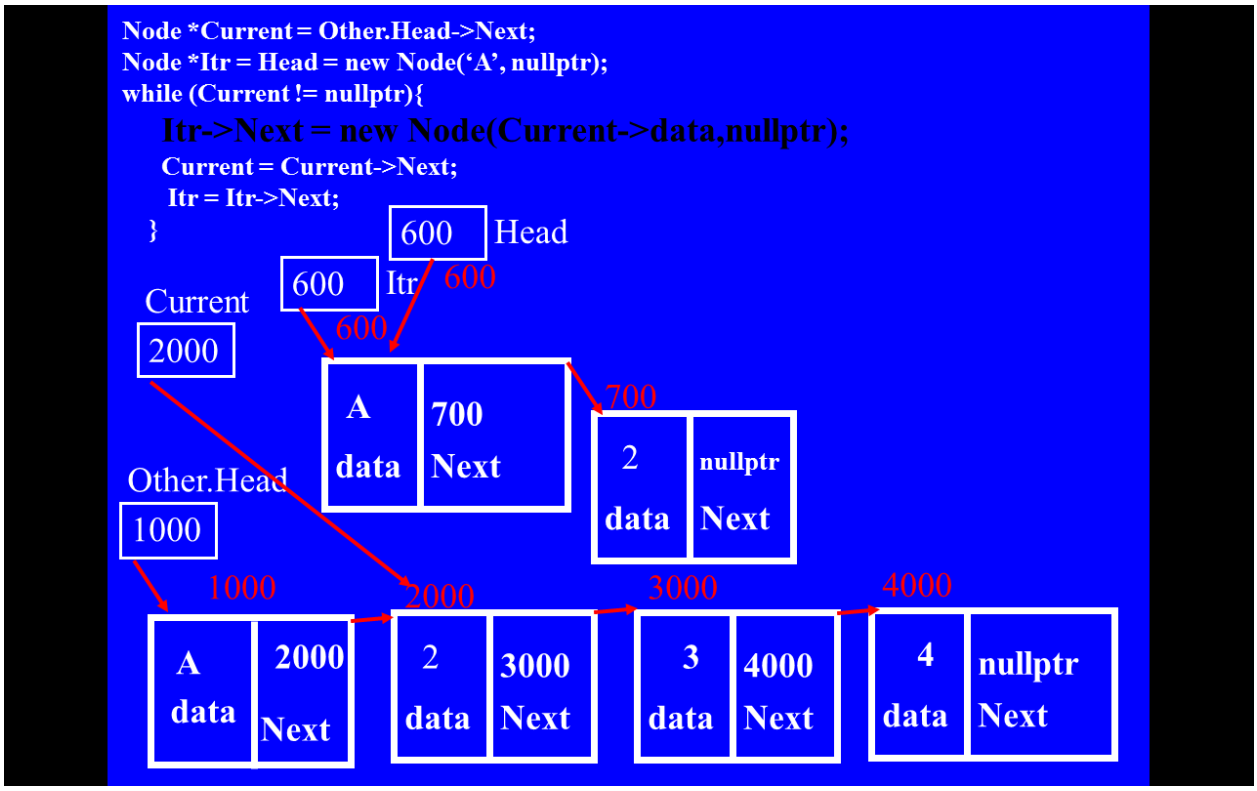


FIG. A.6

When dynamic allocation is done (as is being done in the code line below), one must think of being done in two parts. First the right side is executed in which the operator new calls the constructor of Node class, creates the node and returns the memory address of the new node, thus created.

Itr->Next = new Node(Current->data, nullptr);

Then the memory address returned by operator new is stored in the pointer Itr->Next. Once again, the member selection Itr->Next must be understood to be equivalent to operation (*Itr).Next. The part *Itr is an alias for the Node to which Itr points. Then (*Itr).Next is the member Next in the node pointed by Itr. The operation Itr->Next is merely a shorthand for operation (*Itr).Next. The above code line successfully copies the data field of first non-dummy node from target list and sets the Next field of newly generated node to nullptr.

In order to copy next node, we move the pointer Current by one node. The movement is always conveniently done by code shown in dark code lines in Figure A.7.

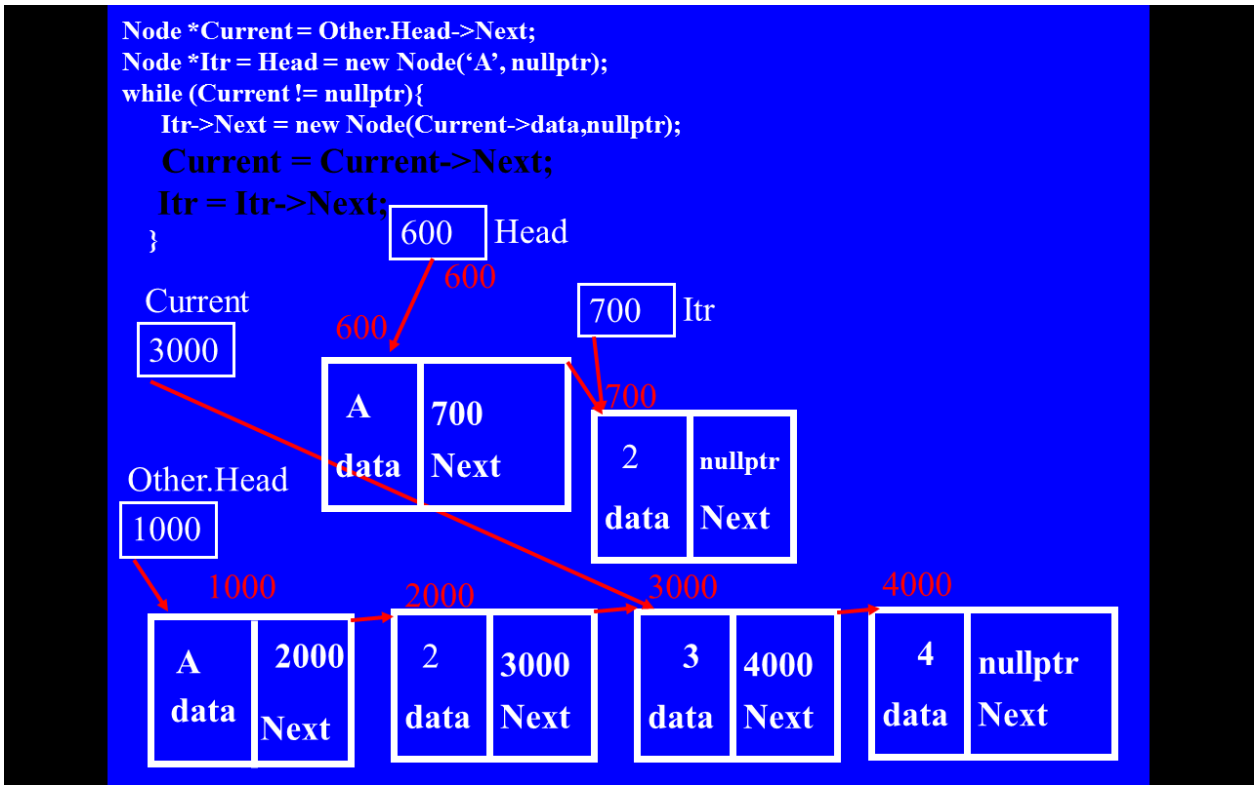


FIG. A.7

The rationale of code lines below is that Current is moved to point to the node to be copied, and Itr is moved to point to the node just copied. For example since, there are yet more nodes to be copied, the pointer Current still does not store nullptr (or has not reached the end of linked list to be copied).

Current = Current->Next;
Itr = Itr->Next;

Therefore loop is entered again, and process described in Figures A.5 to A.7 is repeated again, until entire Linked List is copied, at which point, the movement of pointer Current stores nullptr in it and copy loop exits (Figure A.8).

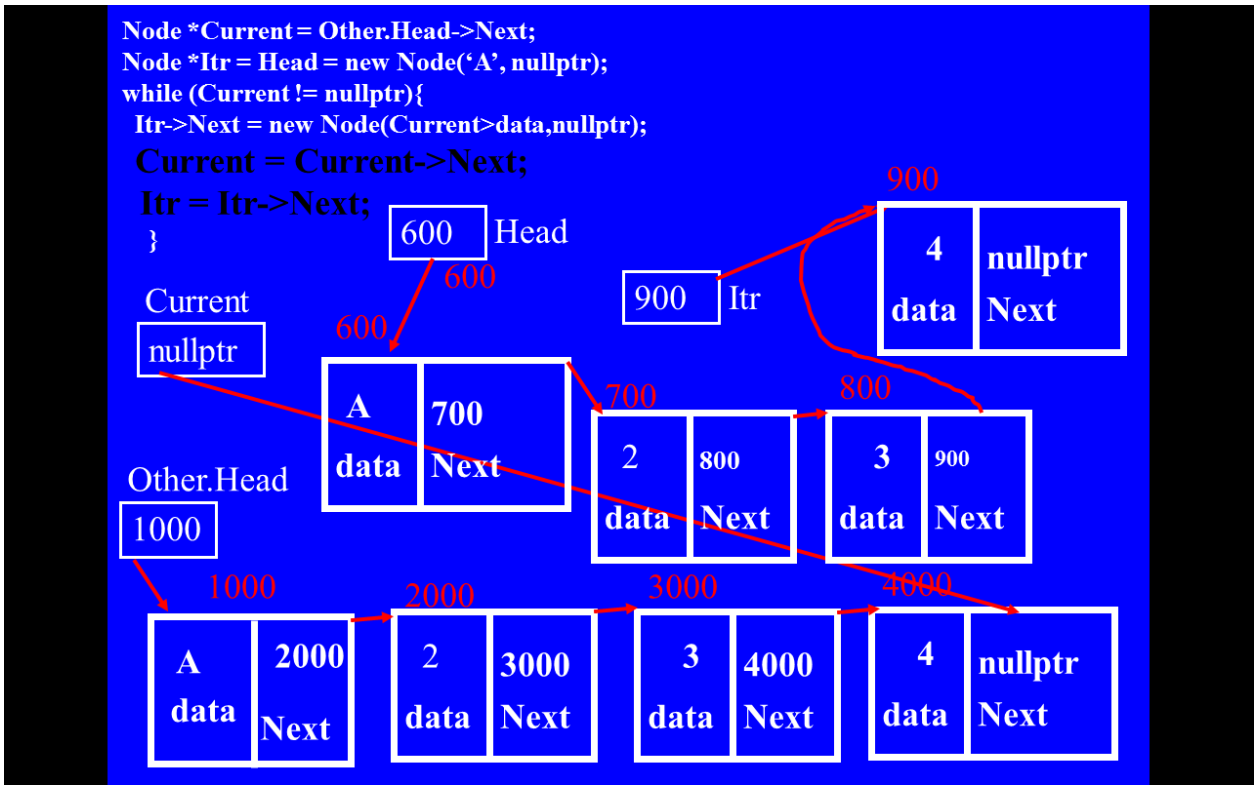


FIG. A.8

We highly recommend that you the reader not be a mere passive reader of these pages. We suggest that you do the drawings showing the linked list code walk through yourself to re-inforce a mental image of pointer movements in link list traversal, firmly in your mind. That kind of active learning is highly recommended to cement understanding of linked list data structure.

destroy[3] Member Function

destroy function deallocates the memory assigned to all linked list nodes and returns this memory to memory pool. The destroy function is called by the IntLinkedList class destructor and assignment operator. Figure B.1 shows the code for the destroy function.

[3] Video and PowerPoints describing destroy function are available. Video is at link: https://www.youtube.com/watch?v=sb0SqJwG9Ww

Code For destroy( )

```
Node * Current = Head; // start with dummy node
while (Current != nullptr){
Node * save = Current;
Current = Current->Next;
delete save;
}
```

FIG. B.1

Once again we set an iterator called Current have it copy address store in Head pointer. The reason this iterator is set differentlly from the one in copy function (recall that there the Current was set to Other.Head->Next), discussed earlier is that, even dummy node needs deletion or deallocation! (Afterall dummy node was also created using new!). The linked list to be deallocated/destroyed and passed to destroy function is shown in Figure B.2 below.

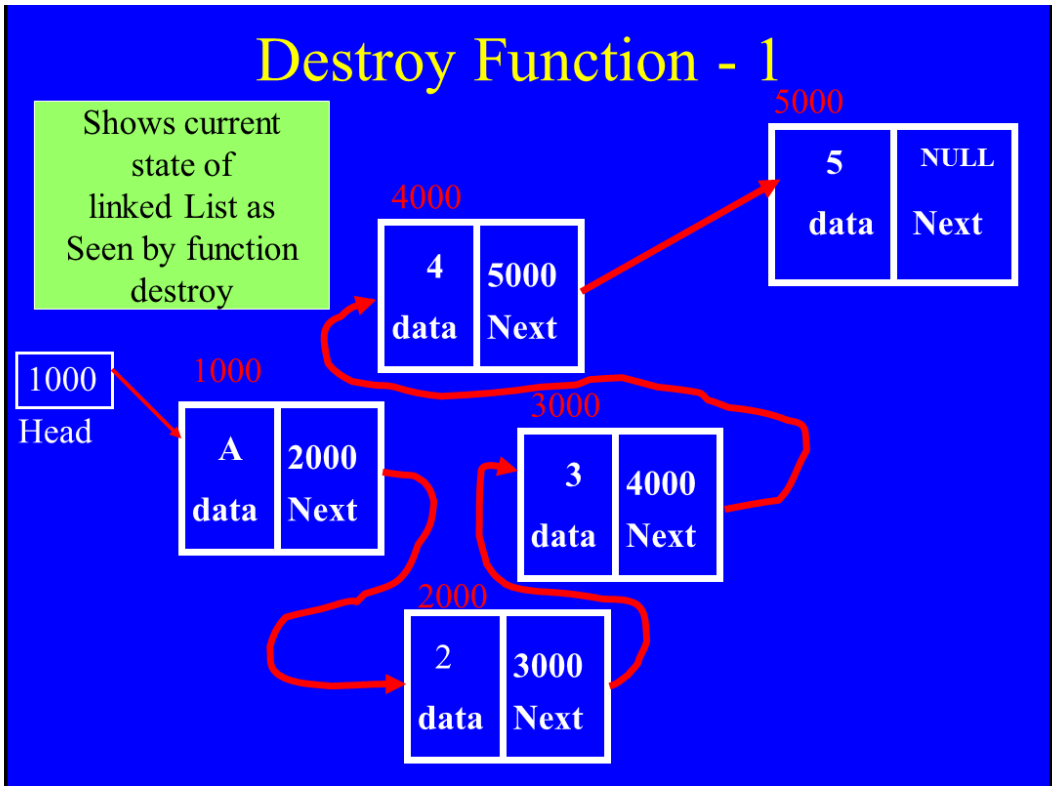


FIG. B.2

Now we show line-by-line execution of destroy function code and how linked list is deallocated one node at a time. Figure B.3 shows the state of current pointer when Current is set to point to same node that Head pointer is pointing.

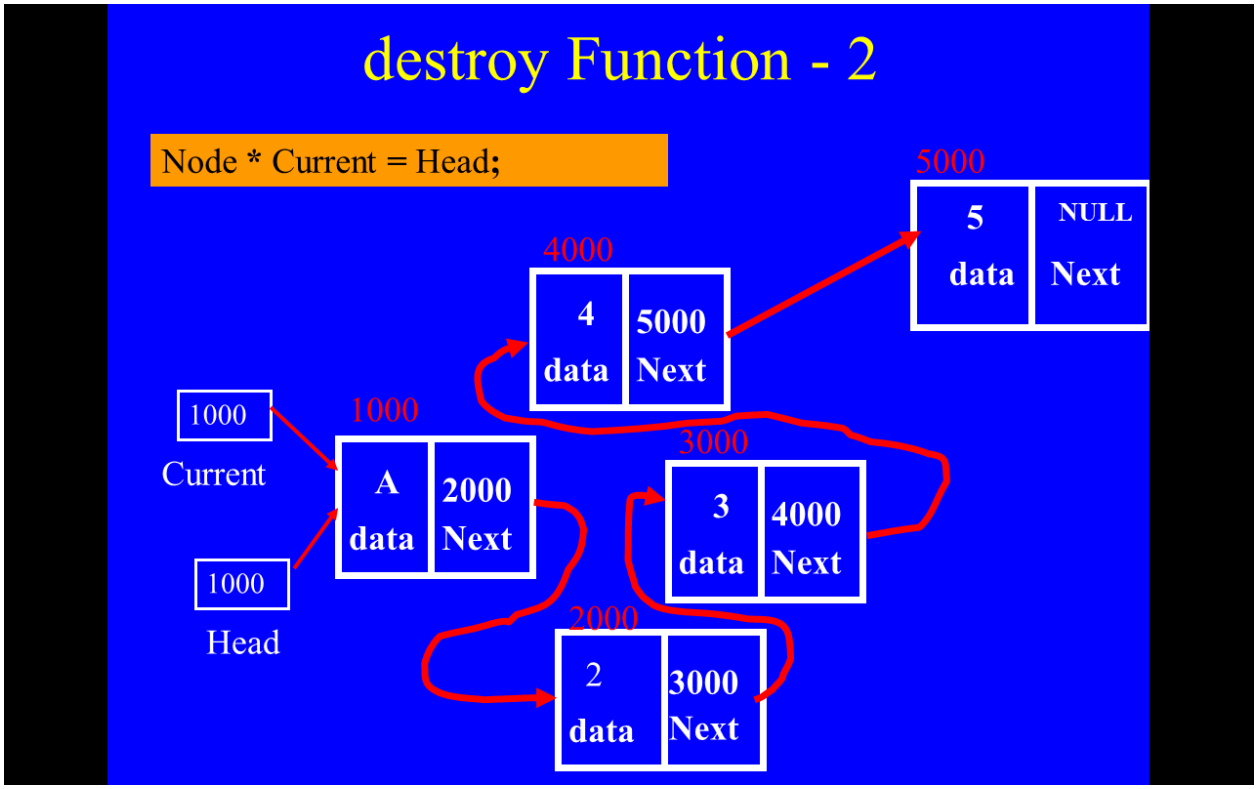


FIG. B. 3

This way the Current points to dummy head node. The loop condition is now tested next (Figure B.4).

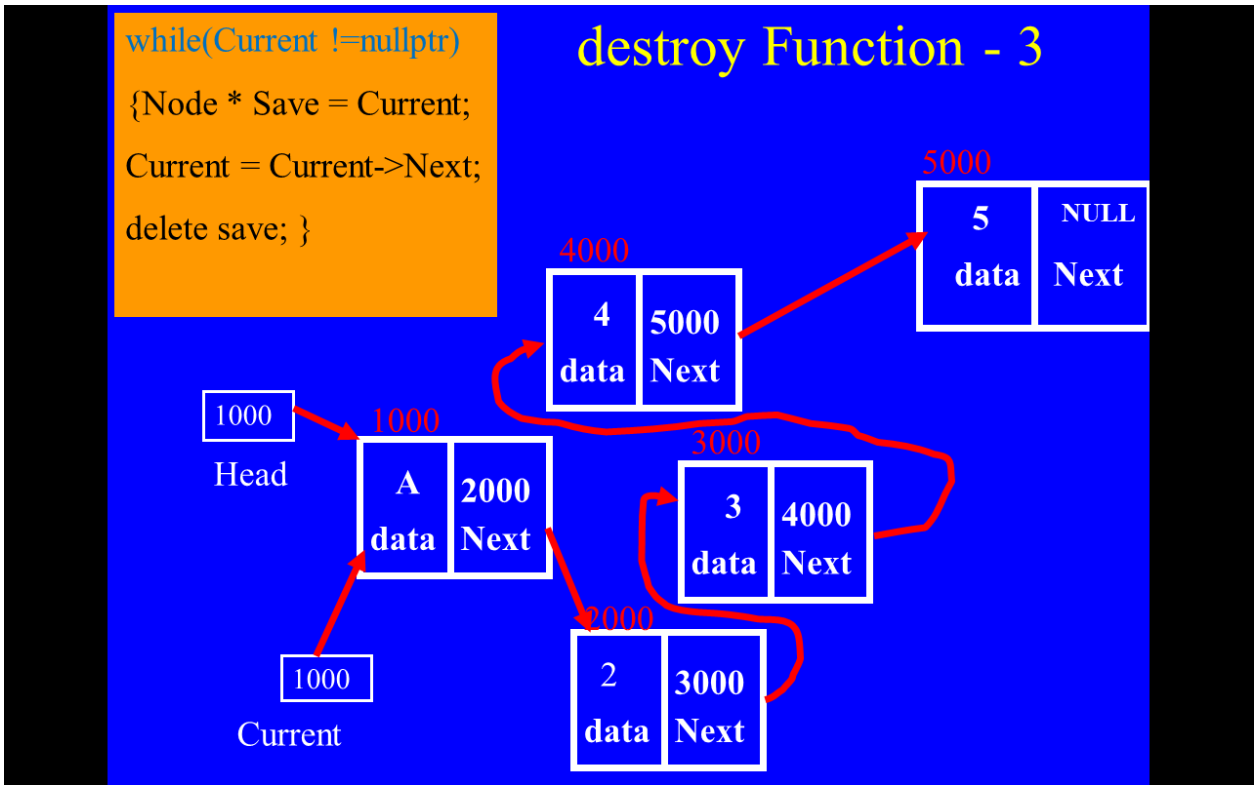


FIG. B. 4

Since Current is NOT storing nullptr in a linked list that has nodes, the loop condition is true and the loop is entered. Then fist line inside the loop is executed that assigns a pointer save to point to the pointee of Current. (Figure B.5).

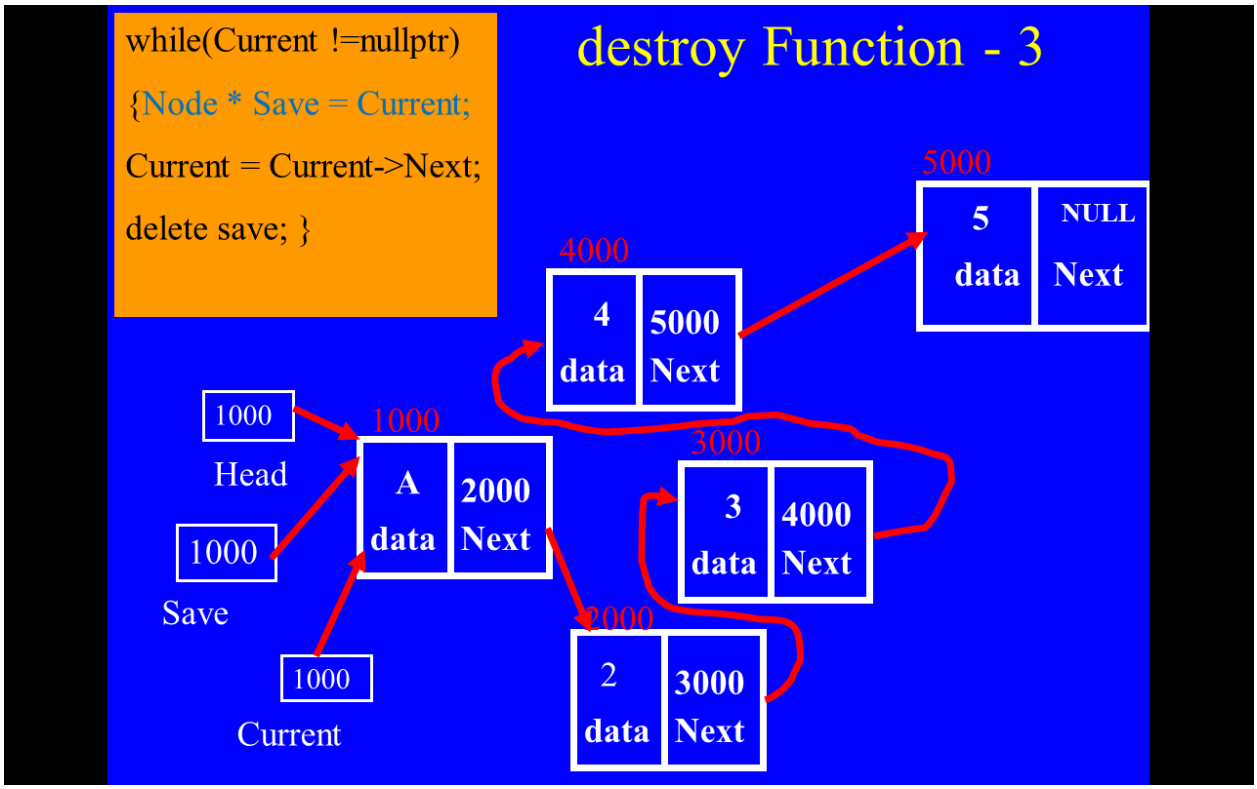


FIG. B. 6

The purpose of this extra pointer (Save) is to use it to delete the pointee it is pointing to after Current is moved to the next node in the linked list. The movement of Current to next node, while Save stays one node behind is shown in Figure B.7.

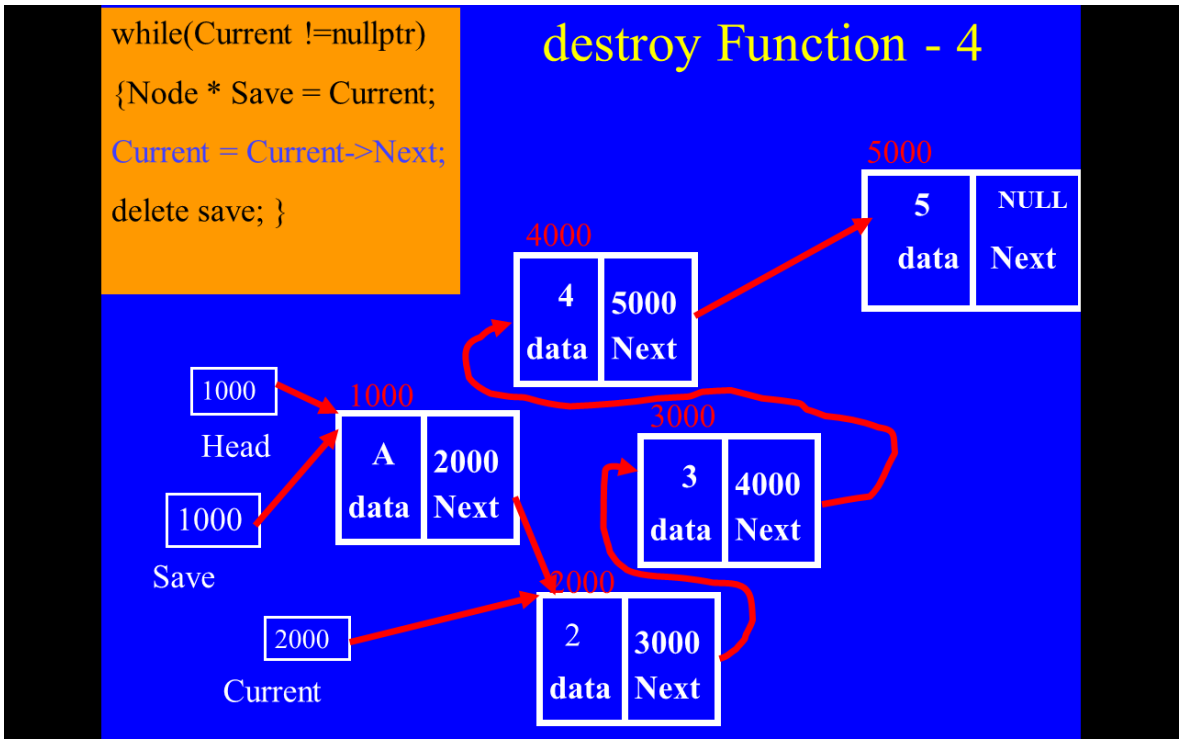


FIG. B. 7

Since Current is safely attached to the rest of the linked list, the Save can be used to delete the previous node (Figure B.8).

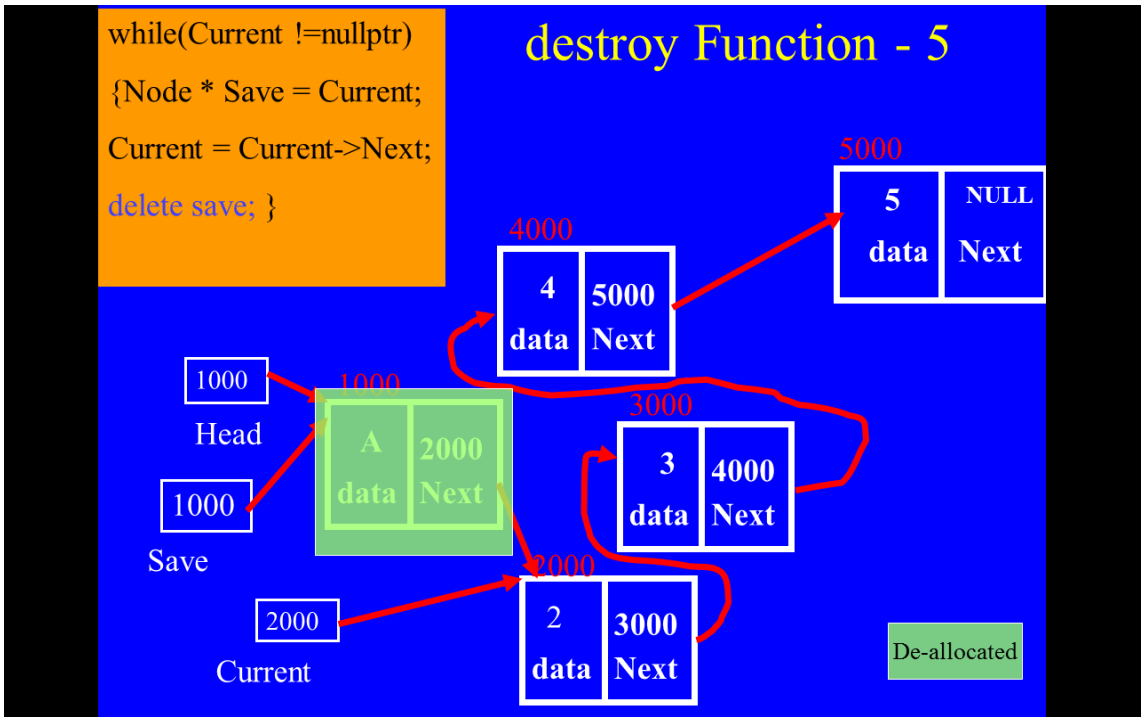


FIG. B. 8

Recall that delete operator ONLY deletes the memory allocated to the pointee of the pointer. The pointer itself remains unaffected. This is fine, because pointer itself is no the stack and all stack variables are automatically destroyed when the block in which they were declared ends and those variables are out of scope.

Loop condition (Current != nullptr) is tested again, and as long as the loop condition is true, the code inside the loop will be executed, deleting one node at a time, until entire linked list is deallocated (Figure B.9).

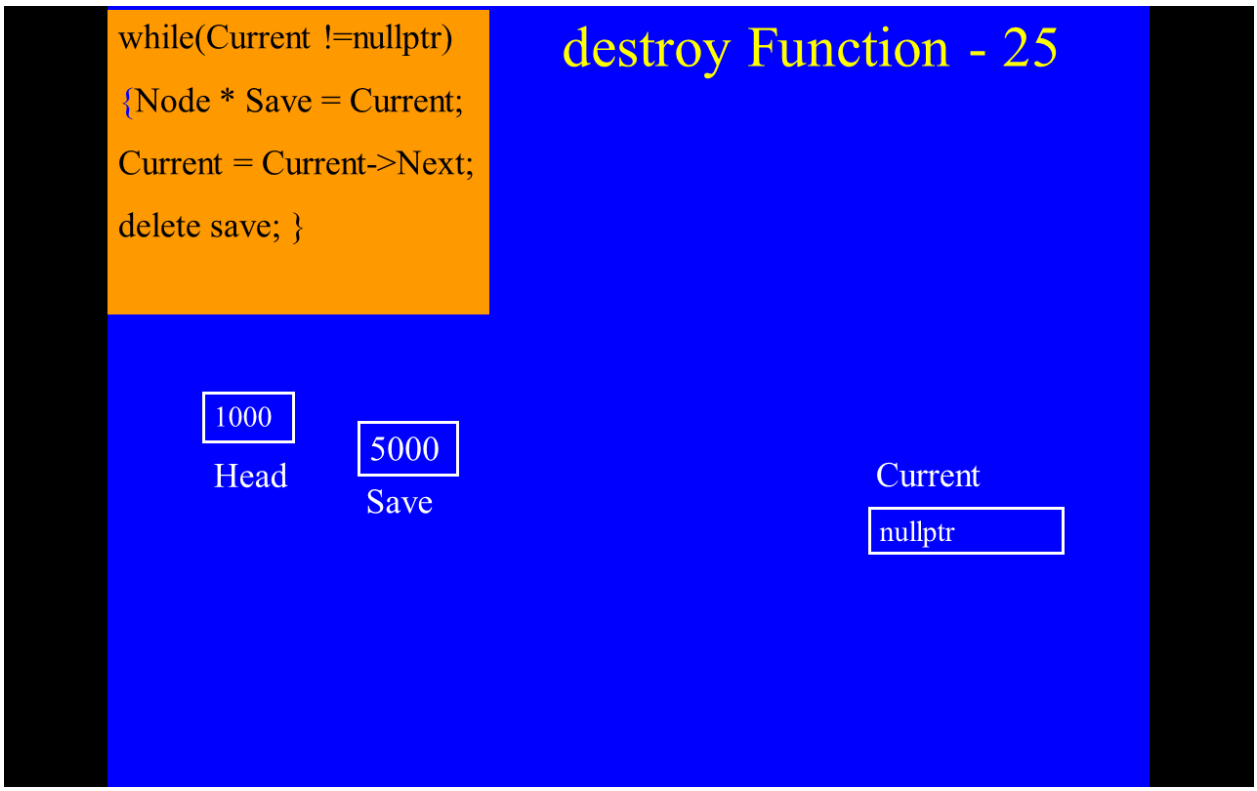


FIG. B. 9

Loop exits because just before last node is deleted, the movement of Current pointer (Current = Current->Next) will store nullptr in it.

insert[4] member function

The code for insert member function is deceptively simple but it works. Insert code below adds the node with data value val to the linked list.

Head->Next = new Node(val, Head->Next);

Understand that to build linked list with more than one node, insert must be called by the user's code from inside of a looping structure as insert is only responsible for adding one node per call.

[4] Video and PowerPoints describing destroy function are available. Video is at link: https://www.youtube.com/watch?v=hh2tp1kuiwk

Also understand that before user's nodes can be added to the linked list, the constructor call for creation of IntLinkedList object as below will automatically create a dummy head node.

IntLinkedList IL; // This calls the code {Head = new Node('A', nullptr);}, thus creating a dummy node

Thus, before call to insert function, the linked list looks like below (Figure C.1).

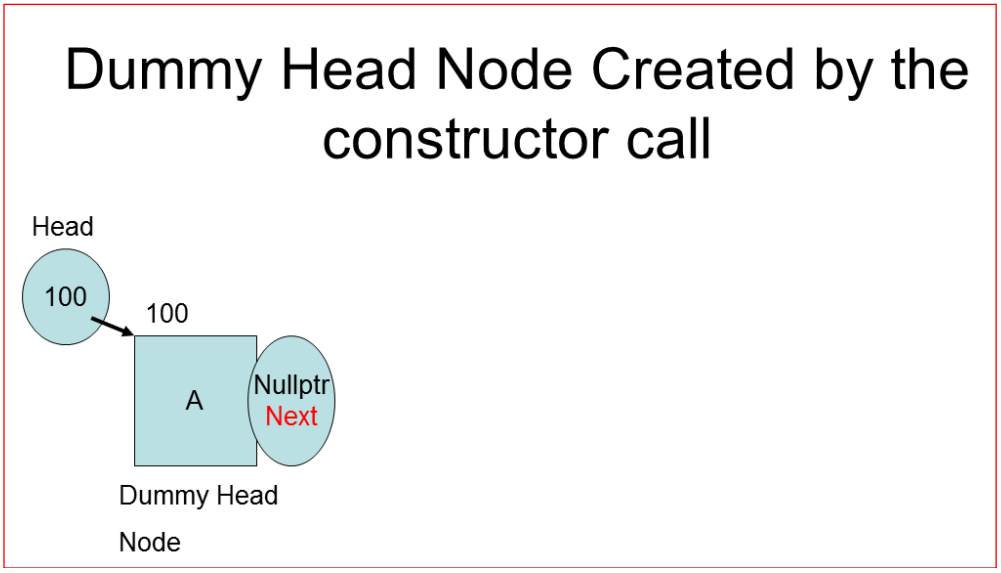


FIG. C.1

Then user's code calls the insert function (assume that value to be added is 5) (Figure C.2).

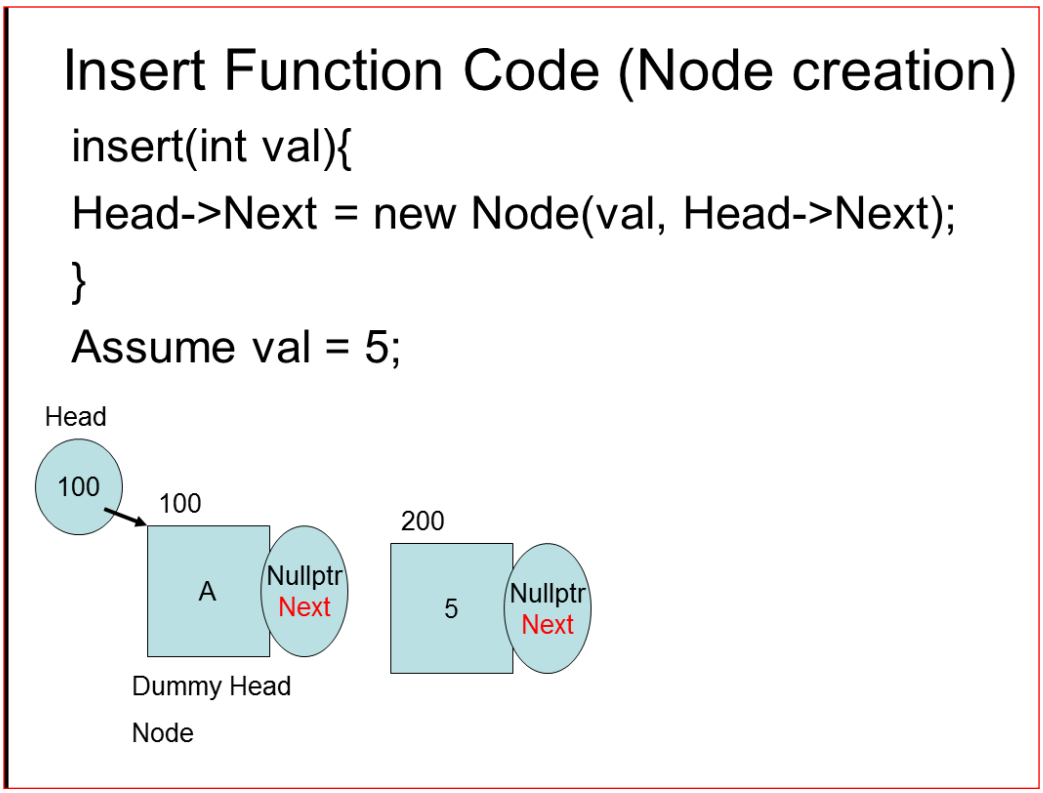


FIG. C.2

As is C++ rule that in a program statement including an assignment operator, the code to the right of assignment is executed first. Thus the code (new Node (val, Head->Next)) creates a new Node with value val (5 in this case), and stores nullptr in the Next field because that is the value stored in Head->Next. new will return a memory address of this fresh node (assumed 200 in this case), and that is stored in pointer Head->Next. Thus the assignment operator forms the link between the dummy head and the newly added user node! (Figure C.3).

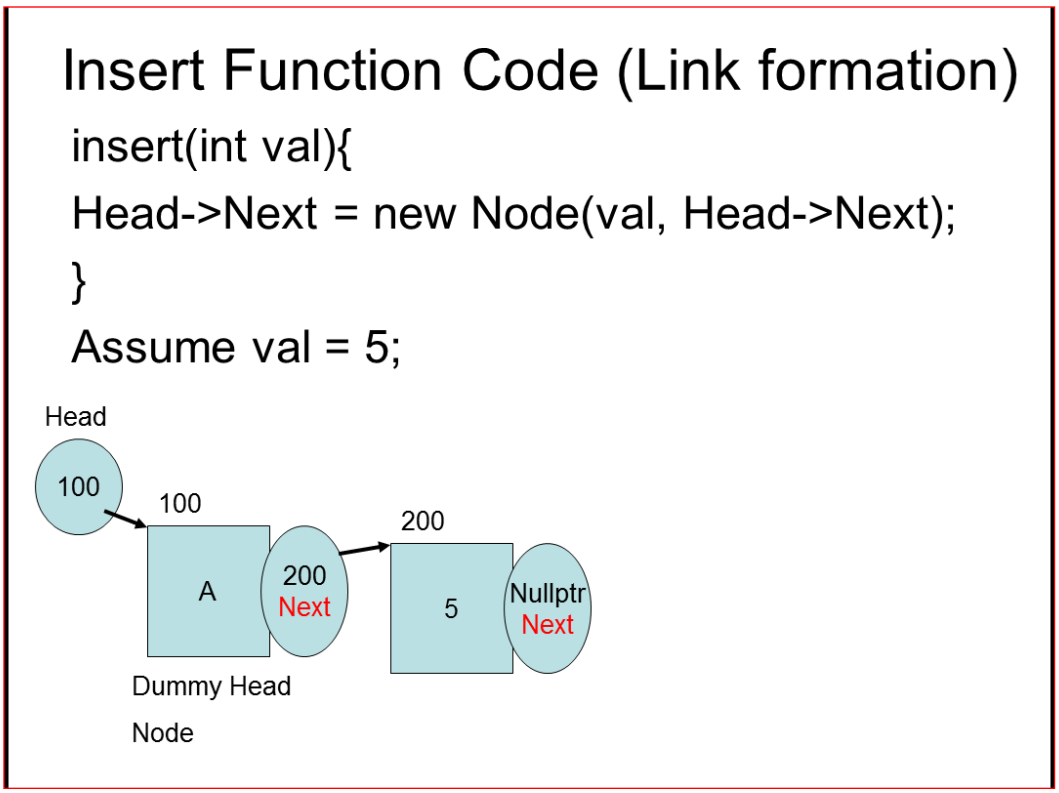


FIG. C.3

Thus node formation and linking it to the dummy head node is conveniently accomplished by same program statement. Imagine now that user wishes to add next node (assume a value = 10). The process of node creation and linking to dummy head node shown in figures C.2 and C.3 is repeated again (Figure C.4).

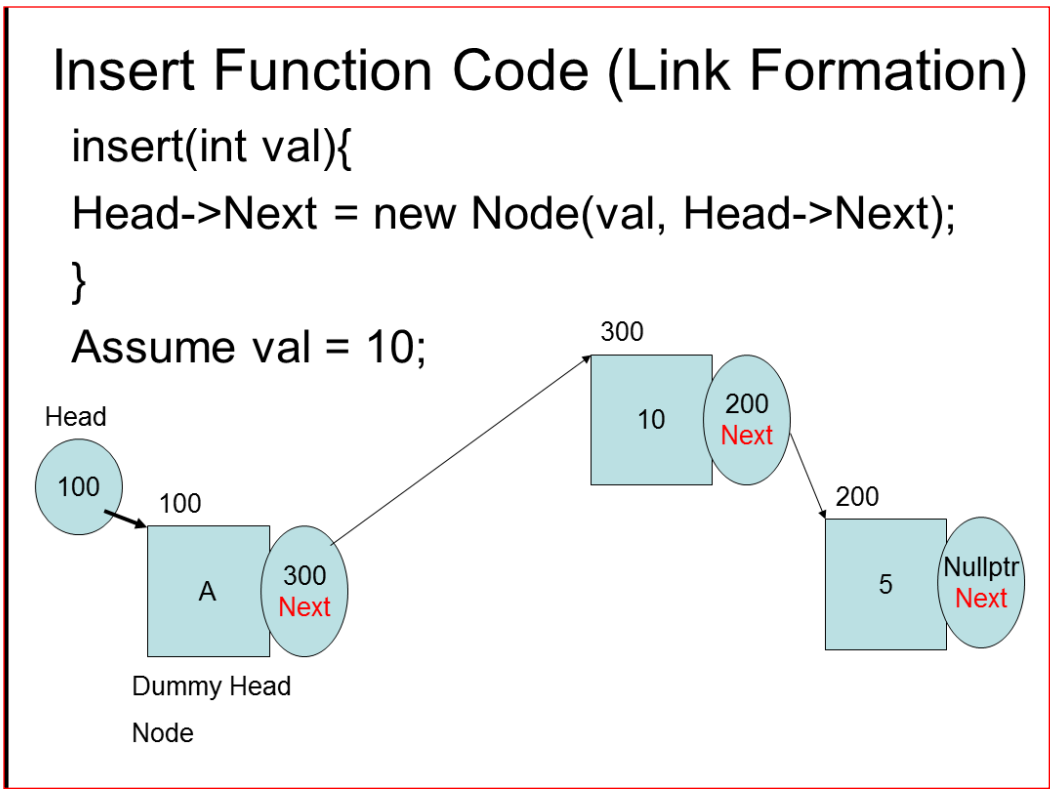


FIG. C.4

Understand that the insert function adds node to the front of the list, right after the dummy head node. Extra coding would be needed if user wished to have any of the below adding schemes:

1. Add the new node to the end of the current linked list.
2. Add the node in the order so that linked list is built as an already sorted list in ascending order.
3. Add the node in the order so that linked list is built as an already sorted list in descending order. Items 2, and 3 are only possible if the linked list members have a natural order, or such order is defined by the class that is the data part of the linked list.
4. Add the new node before certain existing key in the linked list.
5. Add the new node after certain existing key in the linked list.

In later pages we have done a bigger version of a template linked list which shows the code for inserting code to the front of the list (addFront function), or to the rear (addRear). Templated linked list can be understood after learning about templates in chapter 8.

Templated Linked List

Very often we are required to create linked lists of varied data types in the same software application. In that case it would become inconvenient to create multiple code versions of linked lists of different data types. Use of templates in such situations, where generic nature of a data structure or algorithm is to be extended to different data types is ideal. Templates are discussed in depth in another chapter. The template version of a Node class is given below:

```
template<typename Type>
class Node
{
public:
        Type data;
        Node * Next;
        //more code
};
```

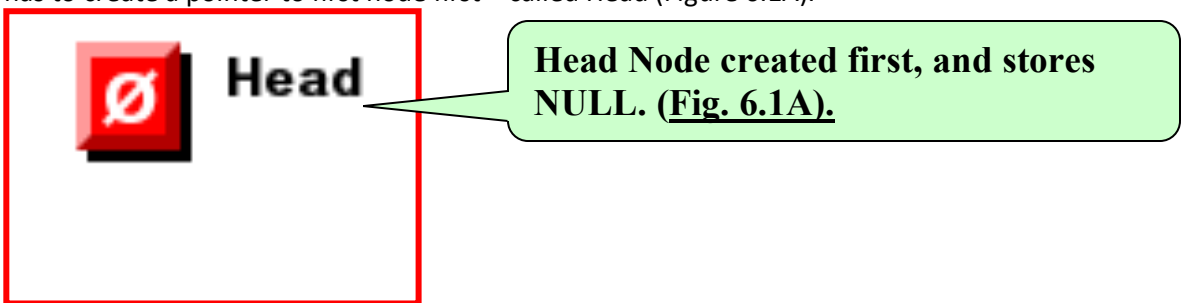
Holds the data for the node. data can be basic type or user defined type.

Holds the address of the next member in the linear list. Last member will hold NULL.

We show a dynamic presentation to illustrate the mechanism of formation, searching and altering a linked list.

Linked List Demonstration

Every linked list has a pointer to its first node. This pointer is called a head node. It simply holds the address of first (or head) node of the list. Note that pointer just holds the address the head node of the list. The pointer itself is not the head node. In process of creating a linked list, one has to create a pointer to first node first – called Head (Figure 6.1A).



Assume that we insert in the head Node the word "El" (for El Camino College) then the list at that point will look as follows (Figure 6.1B).

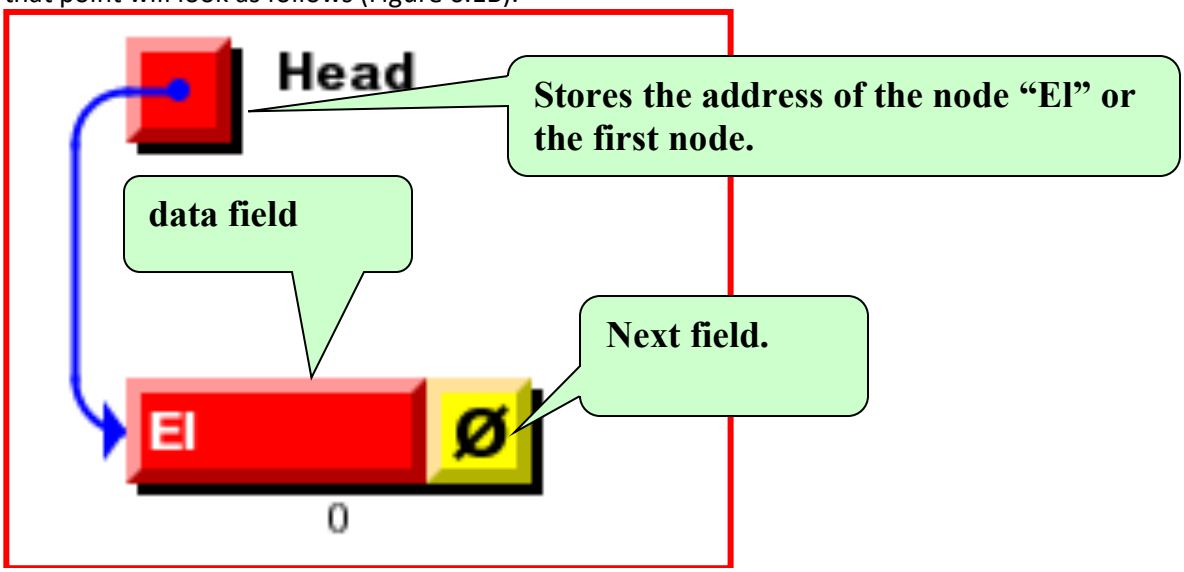


FIG. 6.1B

As we add the following words to the list (one at a time) "College Is Great Place", the list will look like Figure 6.1C.

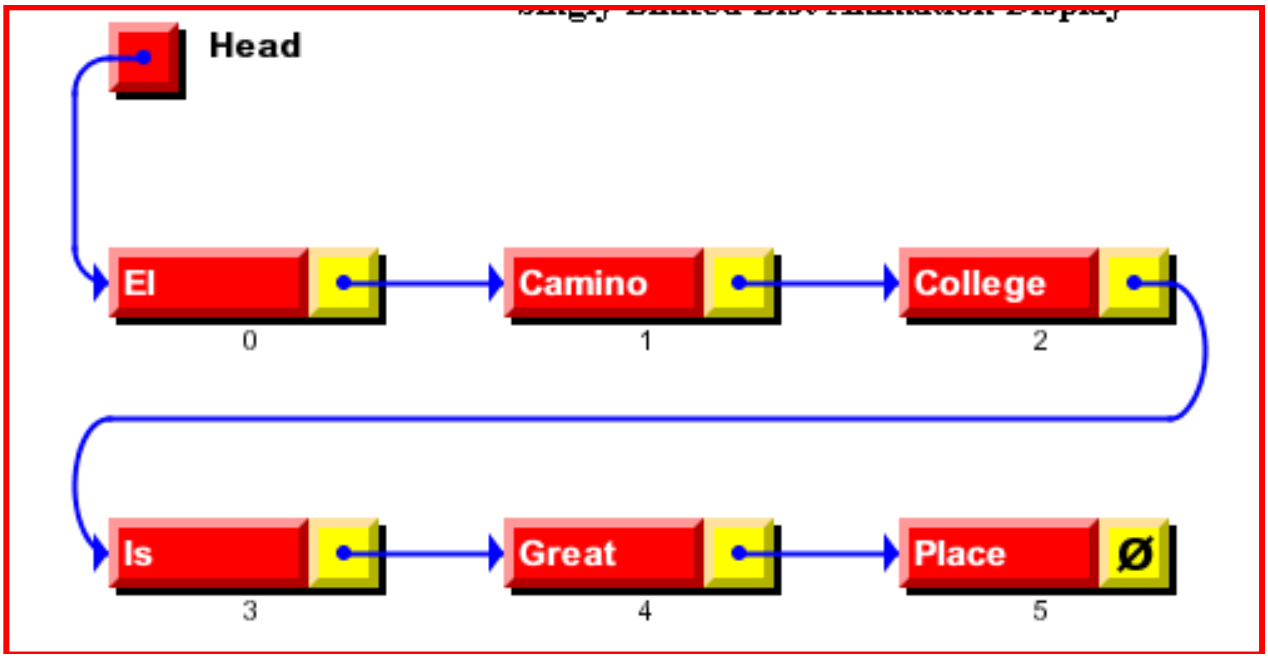


FIG. 6.1C

In the above list Table 6.1 gives the address storage scheme.

Pointer Name	Stores address of Node Number	Data field of the Pointee
Head	0	El
0.Next	1	Camino
1.Next	2	College
2.Next	3	Is
3.Next	4	Great
4.Next	5	Place
5.Next is NULL	NONE	NONE

Table 6.1

We write a simple program, where we design a templated Node Class and main and helper functions to the followings:

1. Create the Linked List
2. Print the Linked List
3. Destroy the Linked List, so that dynamically allocated memory is released.

Table 6.2 gives specifications for the templated class Node.

template<typename Type>
class Node< Type >

Node is a templated class, which creates a node to be added to a linked list of either basic types or user-defined types.

Public Member Functions	
	Node (Type init_data=Type(), Node *Next1=NULL) ***Default/explicit constructor for class Node.***
virtual	~Node () ***Virtual destructor. No code needed inside the body of destructor.***
Public Attributes	
Type	data ***data is the template parameter, which is either basic type or user-defined type.***
Node *	Next ***Next is the pointer to Node type object.***
Friends	
ostream &	operator<< (ostream &out, const Node &lL) ***Overloaded insertion operator >> for class Node.***
istream &	operator>> (istream &in, Node &lL) ***Overloaded extraction operator for class Node.***

Table 6.2

Listing 6.1 shows the source code for the Node class, as well as for the helper and main functions.

```
00001 #include <iostream>
00002 #include <fstream>
00003 #include <string>
00004 #include <typeinfo>
00005 using namespace std;
00006
00010 static int counter = 0;

00015 template<typename Type>
00016 class Node
00017 {
00018 public:
00023         Type data;




00027         Node * Next;
00031

    Node(Type init_data=Type(), Node * Next1 = NULL)
00032         {
00033                 data = init_data;
00034                 Next = Next1;
00035         }
00039   friend ostream& operator << (ostream& out, const Node& IL)
00040         {
00041                 out<<IL.data<< "    ";
00042                 return out;
00043         }
```

1. Class Node is template class, which takes Type as a template parameter.

2. Stores the data in form of basic type or user-defined type.

3. Pointer to the next Node in the list

4. Default/explicit constructor. Notice that no dynamic memory allocation is done inside the constructor.

```
00047         friend istream& operator >>(istream& in, Node& IL)
00048         {
00049                 in>>IL.data;
00050                 return in;
00051         }
00055         virtual ~Node() { }
00056 };
00058
00061 template<typename Type>
00062 void printList( Node<Type> * Head);
00066 template<typename Type>
00067 void destroyList(Node<Type> * Head);
00071 template<typename Type>
00072 void createList(Node<Type> * Head);
00077 void main1();
00078
00079 int main()
00080 {
00081         {
00082                 main1();
00083         }
00084     cout<<"The value of memory counter = "<<counter<<endl;
00085         return 0;
00086 }
00088         void main1()
00089         {
00090                 bool done = false;
00091
00092                 while(!done)
```

5. Prints the linked list. Understand that outside the Node class one must use its qualified name.

6. De-allocates the dynamically allocated memory.

7. Creates the Linked List. Allocates Nodes as List grows.

8. Builds, prints and disposes the integer, double and string linked lists.

```
00093                 {
00094                     int flag =0;
     cout<<"Enter 1 to create an integer list, 2 for double list "
00096         <<" 3 for string list, and 4 to exit: ";
00097                     cin>>flag;
00098
00099                     if(flag == 1)
00100                     {
00101           Node<int> * Head = new Node<int>();
00102                     counter++;
00103                     createList(Head);
00104                      cout<<"Printing your list now:\n";
00105                          printList(Head);
00106
                        destroyList(Head);
00107                     }
00108                     else if(flag==2)
00109                     {
00110                 Node<double> * Head = new Node<double>();
00111                        counter++;
00112                         createList(Head);
00113                       cout<<"Printing your list now:\n";
00114                            printList(Head);
00115                            destroyList(Head);
00116                     }
00117                     else if(flag==3)
00118                     {
```

9. Creates the dummy head node for the integer type List.

10. Calls createList function to create the integer list.

11. Prints the integer linked list.

12. De-allocates the memory assigned to integer linked list. (Deletes the List).

```
00119                   Node<string> * Head = new Node<string>();
00120                           counter++;
00121                           createList(Head);
00122                          cout<<"Printing your list now:\n";
00123                                printList(Head);
00124                                destroyList(Head);
00125                         }
00126                         else if(flag == 4)
00127                         {
00128                                break;
00129                         }
00130                         else
00131                         {
     cout<<"Invalid list type. Type 0 to continue or 1 to exit: ";
00133                                cin>>done;
00134                         }
00135                 }
00136         }
00138         template<typename Type>
00139         void createList(Node<Type> * Head)
00140         {
00141                 bool done = false;
```

13. Checks the typeid of the data stored in the dummy node. Executes the if block when Head->data is int type.

```
   if(typeid(Head->data).name()==string("int"))
00143                 {
00144                         Node<int> Item;
```

14. Still must cast Head pointer into Node<int>* type because the function createList is templated and compiler does not know at compile time as to what actual parameter will be passed to function.

```
00145     Node<int> * IHead = (Node<int>*)Head;
00146
00147                         while (!done)
00148                         {
      cout<<"Please enter an integer to add to the linked list: ";
```

```
00150                                   cin>>Item;
```

15. Create a stand-alone Node whose pointer is Tnode, which has the data field as Item.data (Item is entered by the user), and whose Next member stores the address that Next field of dummy node has.

```
        Node<int> * TNode =
         new Node<int>(Item.data,IHead->Next);
00152                                   counter++;
00153
00154
```

16. Makes the Next field of dummy node point to the TNode. This attaches the dummy node to the newly added node.

```
                    IHead->Next = TNode;
00155         cout<<"More data ? Enter 0 to continue, 1 to end : ";
00156                                   cin>>done;
00157                               }
00158                   }
```

17. Repeats the steps 13 to 15 if Head pointer points to a double type dummy node.

```
else if(typeid(
Head->data).name()==string("double"))
00160                   {
00161                               Node<double> Item;
00162 Node<double> * DHead = (Node<double>*)Head;
00163                               while (!done)
00164                               {
   cout<<"Please enter a double type to add to the linked list: ";
00166                                   cin>>Item;
  Node<double> * TNode = new Node<double>(Item.data,DHead->Next);
00168                                   counter++;
00169
00170                                   DHead->Next = TNode;
00171         cout<<"More data ? Enter 0 to continue, 1 to end : ";
00172                                   cin>>done;
00173                               }
00174                   }
```

18. Repeats the steps 13 to 15 if Head pointer points to a string type dummy node.

```
00175                   else
00176                   {
00177                               Node<string> Item;
00178               Node<string> * SHead = (Node<string>*)Head;
00179                               while (!done)
00180                               {
   cout<<"Please enter a string to add to the linked list: ";
00182                                       cin>>Item;
   Node<string> * TNode = new Node<string>(Item.data,SHead->Next);
00184                                       counter++;
00185                                       //Attach the node to the
dummy node
00186                                       SHead->Next = TNode;
00187           cout<<"More data ? Enter 0 to continue, 1 to end : ";
00188                                       cin>>done;
00189                               }
00190                   }
00191           }




00193           template<typename Type>
00194       void printList ( Node<Type> * Head)
00195           {
00196                   //No need to print dummy node
```

19. Does the type checking and Head pointer casting similar to the function createList

```
00197                   if(typeid(Head->data).name()==string("int"))
00198                   {
00199                           Node<int> * Iter;
00200                     Node<int> * IHead = (Node<int>*)Head;
```

20. No need to print the dummy node. Therefore, Iter is moved to point to node after the dummy node.

```
00201                       Iter = IHead->Next;
00202
00203                   while(Iter !=NULL)
00204                       {
00205                        cout<<Iter->data<<"   ";
00206                                Iter = Iter->Next;
00207                       }
00208                       cout<<endl;
00209               }
00210       else if(typeid(Head->data).name()==string("double"))
00211                   {
00212                           Node<double> * Iter;
00213                 Node<double> * DHead = (Node<double>*)Head;
00214                           Iter = DHead->Next;
00215                           while(Iter !=NULL)
00216                           {
00217                                   cout<<Iter->data<<"   ";
00218                                   Iter = Iter->Next;
00219                           }
00220                           cout<<endl;
00221               }
```

25. Repeats steps 19 to 23 if the dummy node stores string type data.

```
00222
                else
00223             {
00224                         Node<string> * Iter;
00225               Node<string> * SHead = (Node<string>*)Head;
00226                         Iter = SHead->Next;
00227                         while(Iter !=NULL)
00228                         {
00229                                 cout<<Iter->data<<"  ";
00230                                 Iter = Iter->Next;
00231                         }
00232                         cout<<endl;
00233               }
00234       }
00236       template<typename Type>
00237       void destroyList(Node<Type> * Head)
00238       {
00239     if(typeid(Head->data).name()==string("int"))
00240     {
00241                         Node<int> * Iter;
00242      Node<int> * IHead = (Node<int>*)Head;
00243                         Iter = IHead;
00244
00245       while(Iter !=NULL)
00246       {
00247             IHead = IHead->Next;
00248             delete Iter;
00249             counter--;
00250       Iter = IHead;
00251       }
```

30. Move Iter to Node where head node is pointing.

```
00252   }
00253       else if(typeid(Head->data).name()==string("double"))
00254       {
00255            Node<double> * Iter;
00256            Node<double> * DHead = (Node<double>*)Head;
00257            Iter = DHead;
00258            while(Iter !=NULL)
00259            {
00260               DHead = DHead->Next;
00261               delete Iter;
00262               counter--;
00263              Iter = DHead;
00264            }
00265      }
00266           else
00267          {
00268              Node<string> * Iter;
00269    Node<string> * SHead = (Node<string>*)Head;
00270              Iter = SHead;
00271              while(Iter !=NULL)
00272              {
00273                  SHead = SHead->Next;
00274                  delete Iter;
00275                  counter--;
00276                  Iter = SHead;
00277              }
00278         }

00279  }//Listing 6.1
```

We describe the Listing 6.1 as follows:

class Node

Node class is a templated class, which takes (Type) as the template parameter **(bubble#1).** Node class declares a public parameter data, which is of type template parameter Type. data can be either C++ basic type or user-defined type (bubble#2). Class Node also has a pointer data Next, which can store address of a Node type object (bubble#3). The default/explicit constructor for class Node sets the fields to either user defines values or the data field to the default constructor value provided by the Type parameter and the Next field to NULL (bubble#4). Note that no dynamic memory allocation is done in the constructor. Therefore, the class Node does not need the copy constructor and overloaded assignment operator as C++ provided functions for copying and assigning will be adequate. We do however need a code-less virtual destructor

for dynamic casting and run-time type identification of derived classes5. The class Node also has overloaded insertion << and extraction << operators overloaded to print the data field or to input its value.

Stand-alone functions

Listing 6.1 has three stand-alone functions for the following purposes.

1. To print the list in forward order. The templated function printList does that (bubble #5). Since this function takes a Node type formal parameter, the qualified name of the Node class, Node<Type> must be used as formal parameter.
2. Templated function destroyList, de-allocates the dynamically allocated memory for the linked list (bubble #6).
3. And the function createList creates the linked list taking user input. The main program passes the head node of the linked list with dummy node in it, and createList creates the linked list and its head pointer becomes available in the main function (bubble #7).

4. The function main1 contains the code, which creates, prints and disposes the linked list (bubble #8).

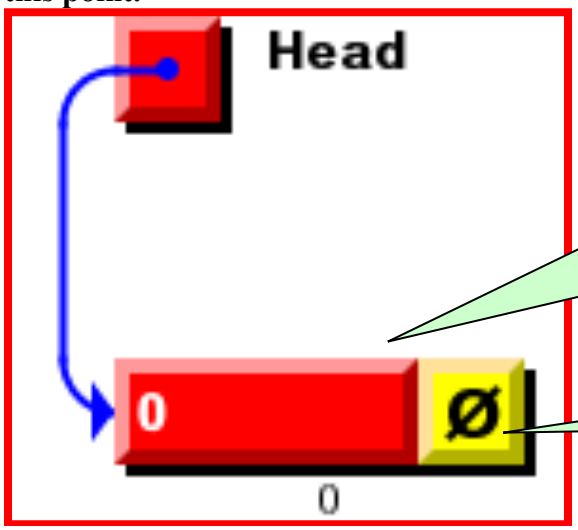
Function main1

In helper function main1, the user is prompted to enter a flag value (1, 2, or 3) to indicate if they wish to create a linked list of integer, double, or string. For the flag value of one, the pointer Head is created by statement:

```
Node<int> * Head = new Node<int>();
```

Default constructor called (bubble #9).

When we instantiate a template class, it requires the actual data type for the template argument. Therefore it is necessary to put data type inside the angular brackets, following the class Name. Notice that the default constructor is called to create a node whose pointee is Head. The Figure 6.2 shows how the list looks like at this point.



Default constructor call new Node<int>( ) creates a dummy node with data field having a value of zero and Next field as NULL (indicated by zero).

Next field contains NULL address.

FIG. 6.2

[5] **C++ has a casting operator called dynamic_cast. It does not work on a class unless the class has a virtual destructor. Therefore it is always essential to include a virtual destructor in every class.**

Linked list created in this manner is called a "dummy head linked list", since its first node contain dummy data. The function main1 passes the Head pointer to the function createList, where user inputted data creates the linked list (bubble #10). The list is then printed by making a call to the function printList (bubble #11), destroyed by calling the function destroyList (bubble #12).

Function createList

The function createList gets the head node with a dummy node as an argument. First the function determines the data type of the dummy node to see as to what kind of list to build. This is done by checking the typeid of the data field of dummy node (bubble #13). If the Head->data is of type int, then linked list will of Integer type. A local variable of type Node, item whose template parameter is int is created to take user input. Since Head pointer must also be a Node<int> *, we must cast the Head pointer and create a local head pointer to the list called IHead (bubble #14). Notice that even though we performed typeid check, the Head pointer still must be cast into type Node<int>* because at compile time compiler cannot ascertain as to what actual parameter will be passed to the function createList at run–time, and a copy of createList is only created at run-time. Inside the flag controlled loop we take user input for the data for integers to be placed in the linked list. Since Node class has an overloaded extraction operator >>, it is enough to use Item variable to get user input for integer data. The statement

```
Node<int> * TNode =
        new Node<int>(Item.data,IHead->Next);
```

creates an integer type Node, TNode, containing user inputted integer (Item.data), and its Next field pointing to the pointee of Next pointer of dummy node. The situation is shown in Figure 6.3 below (bubble # 15).

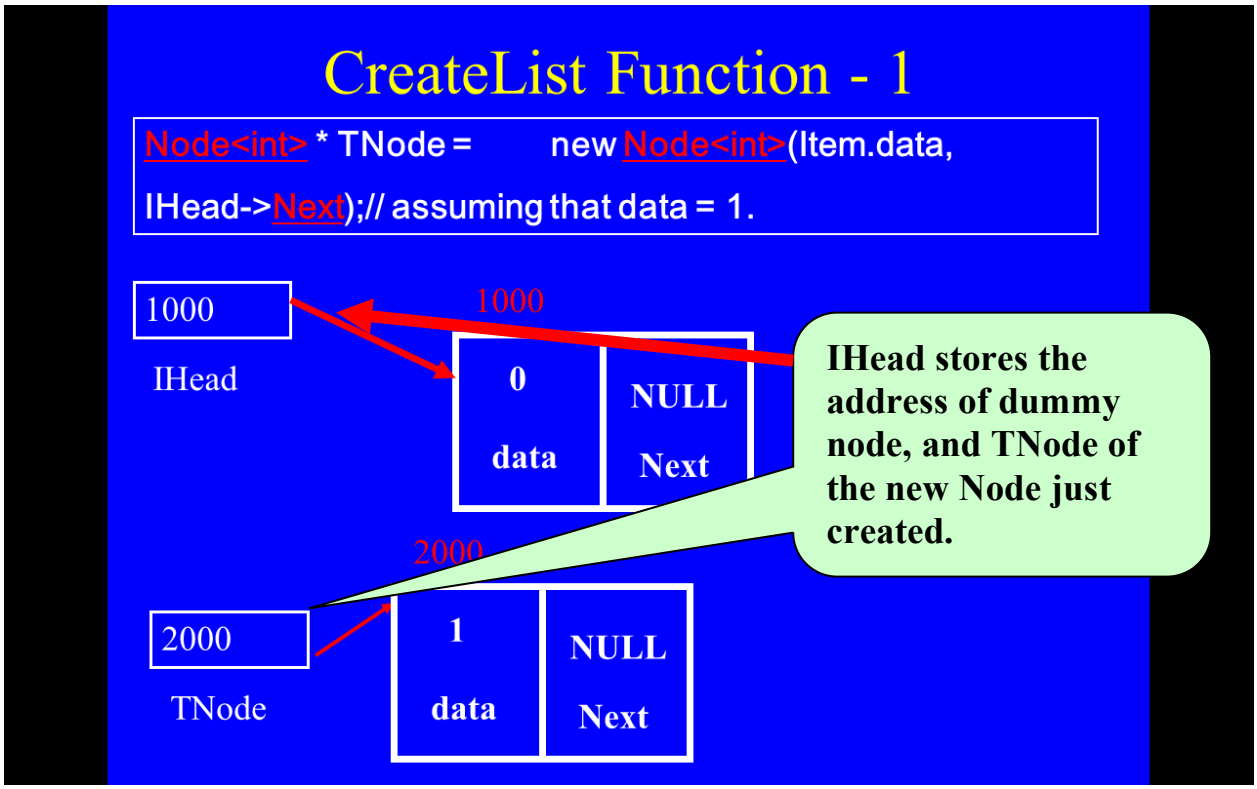


FIG. 6.3

Since we wish to add this Node, whose pointer is TNode to the list, we perform the following assignment:

```
IHead->Next = TNode;
```

This simply means that make the pointer IHead->Next point to the pointee of TNode (bubble #16). Figure 6.4 below now shows the situation.

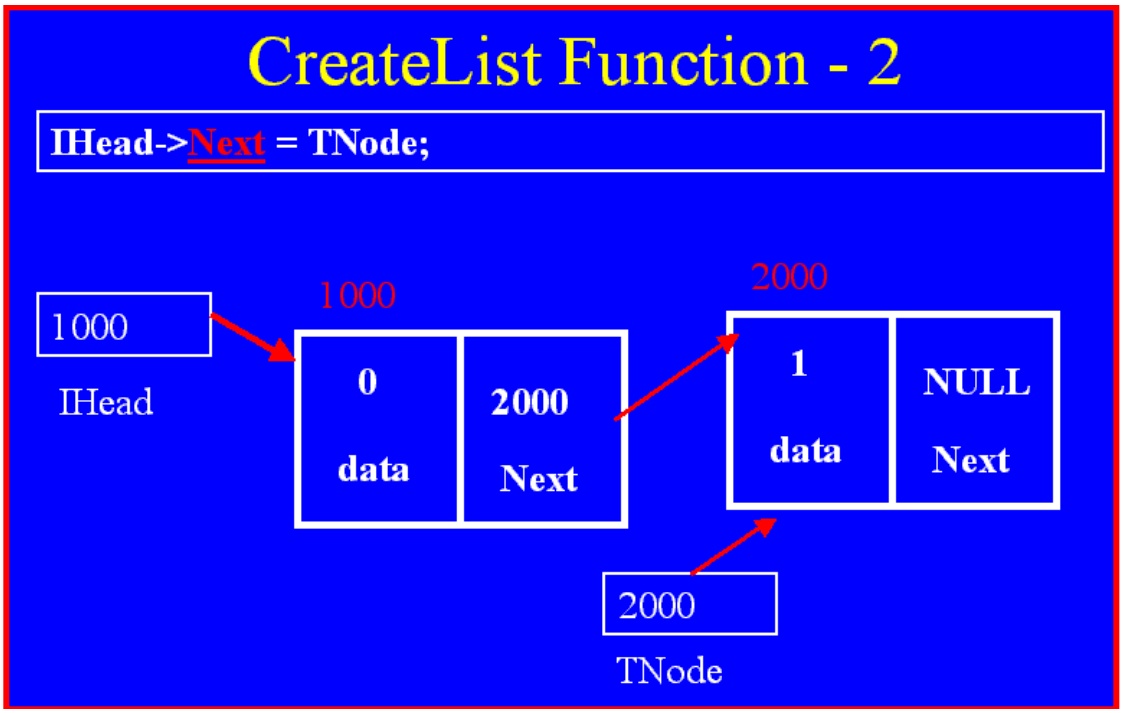


FIG. 6.4

Function then prompts user to either add more nodes to list or exit. If user chooses to continue to add more nodes to the list, then flag control loop pretest condition tests to true and user is prompted to input next integer. Process shown in bubble #15 and Figure 6.3 is repeated (Figure 6.5).

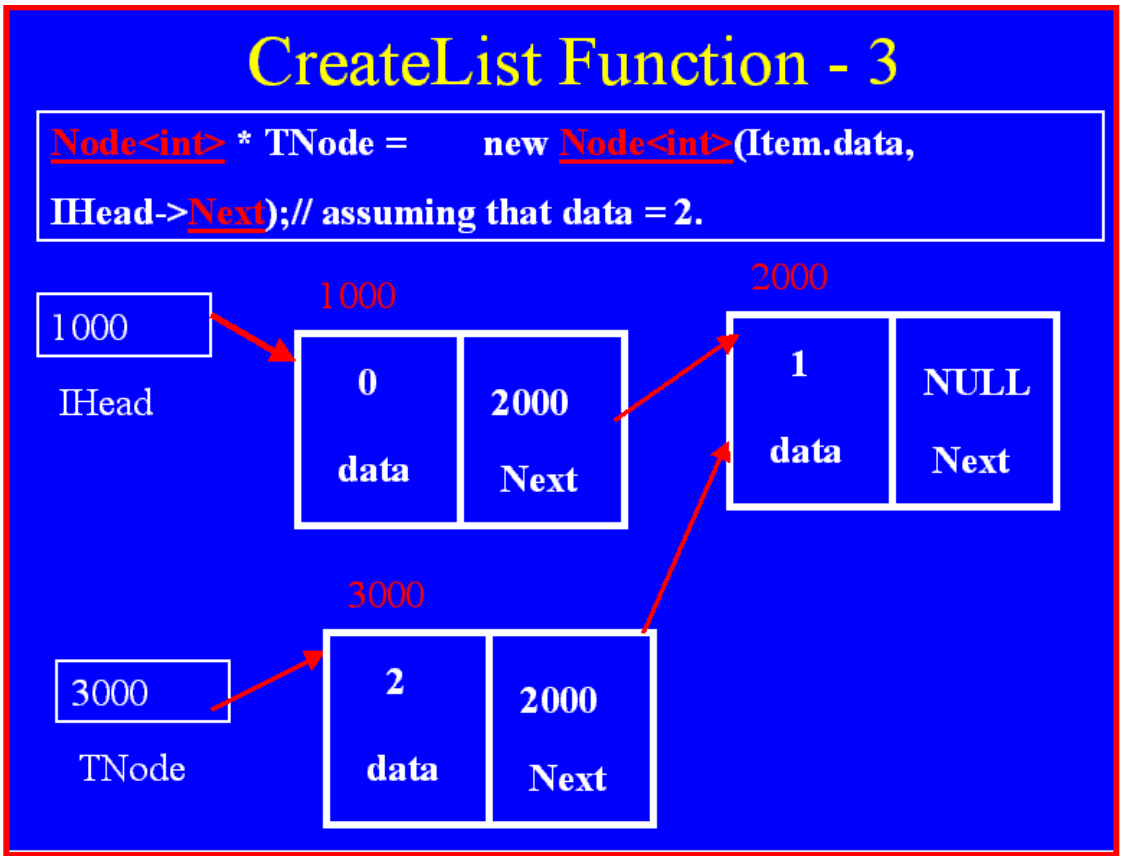


FIG. 6.5

We repeat the process shown by bubble #16 again, which involves making the Next field of the dummy node point to the pointee of TNode. The result is shown in Figure 6.6.

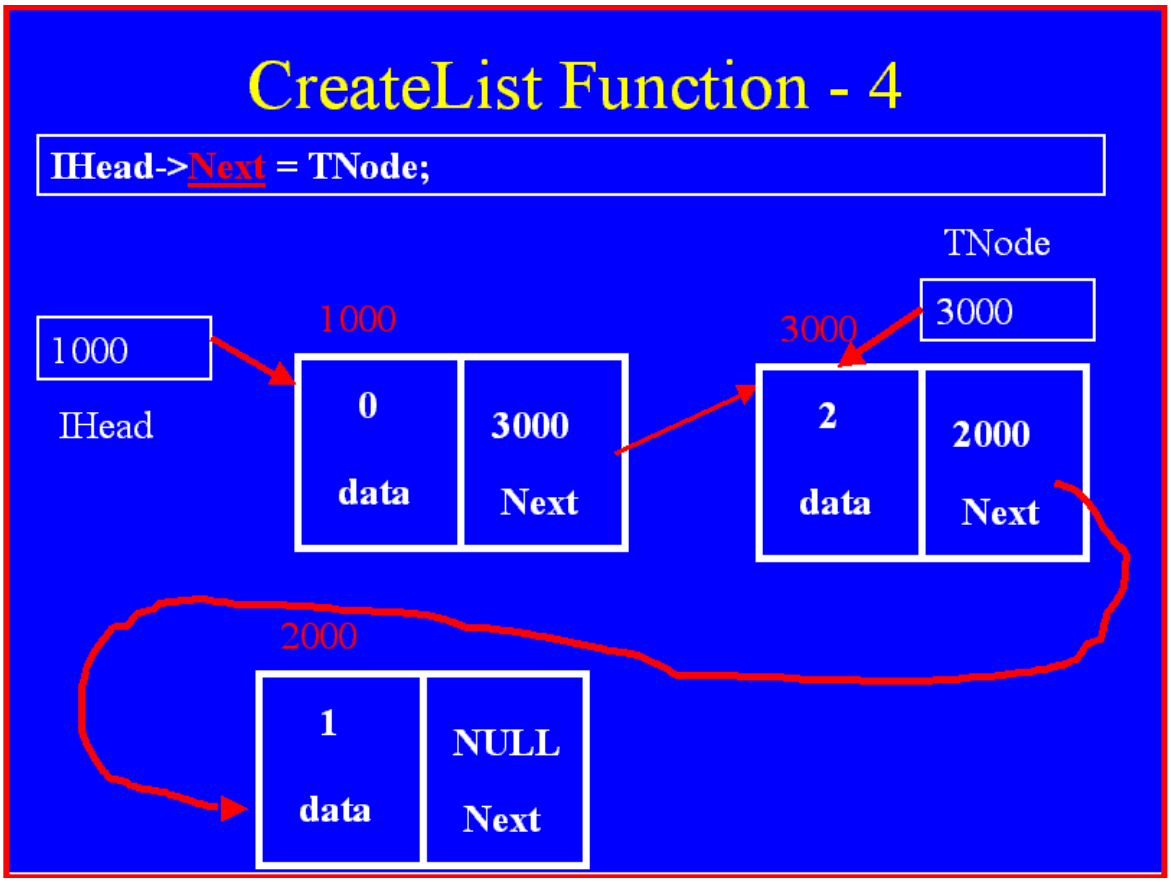
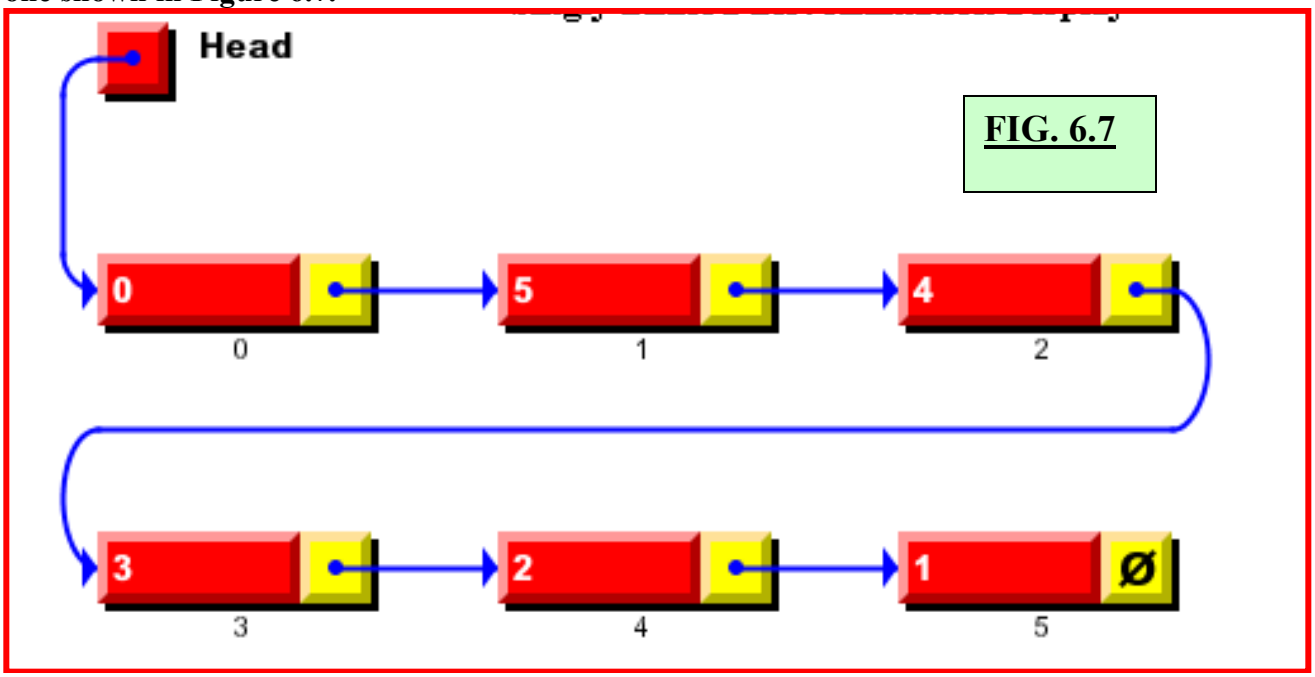


FIG. 6.6

If this process continues, the client can build a linked list of integers similar to the one shown in Figure 6.7.



The above linked list builds in the reverse order to the data entered, since we are always adding the new data after the dummy node.

The PowerPoint presentation above shows as to how the createList function executes to build an integer linked list. If the dummy node contains double type data, then steps 13 to 16 are repeated for the double type data to build a linked list of double data type (bubble # 17). If the dummy node contains the string type data then the steps 13 to 16 are repeated to build string linked list (bubble #18).

Function printList

Similar to function createList, the function printList gets the head node of the linked list (which is a dummy node) as an argument. The type checking of the type of data stored is then done (bubble #19). Based on the type checking either of the three block of if/else structure are executed. For example if the data stored in the first node is an int type then the first block is executed (and so on). Depending upon the type of data stored in the list an appropriate iterator is created. For example – to print integer list an int type iterator is created as follows:

```
Node<int> * Iter;
```

The node Head is the cast to int type. The first node is the dummy node. Therefore there is no need to print its data member. Therefore the iterator Iter is moved to next node (bubble #20). To print the list, we run a while loop based on the following logic:

> **Since Next field of last node stores NULL (FIG. 6.1C) the loop termination condition would be:**
>
> ```
> Iter == NULL;
> ```
>
> **Therefore the loop pre-test condition is negation of loop termination condition, which is !(Iter == NULL), which becomes:**
>
> ```
> Iter != NULL;
> ```
>
> **Thus loop is codes as**
>
> ```
> while ( Iter != NULL) // bubble #21
> ```

The first loop task prints the data part of the node to which iterator is pointing to (bubble #22), and next loop task moves the iterator to next node (bubble #23).

Process described above is repeated for the linked list of double or string data types (bubbles #24, and #25).

Function destroyList

The purpose of function destroyList is to de-allocate the memory allocated by the function createList and the main function. The function de-allocates memory one node at a time starting with the front node. Assume that Figure 6.8 shows the current state of the linked list when the pointer to the head of the list is passed to the function destroyList.

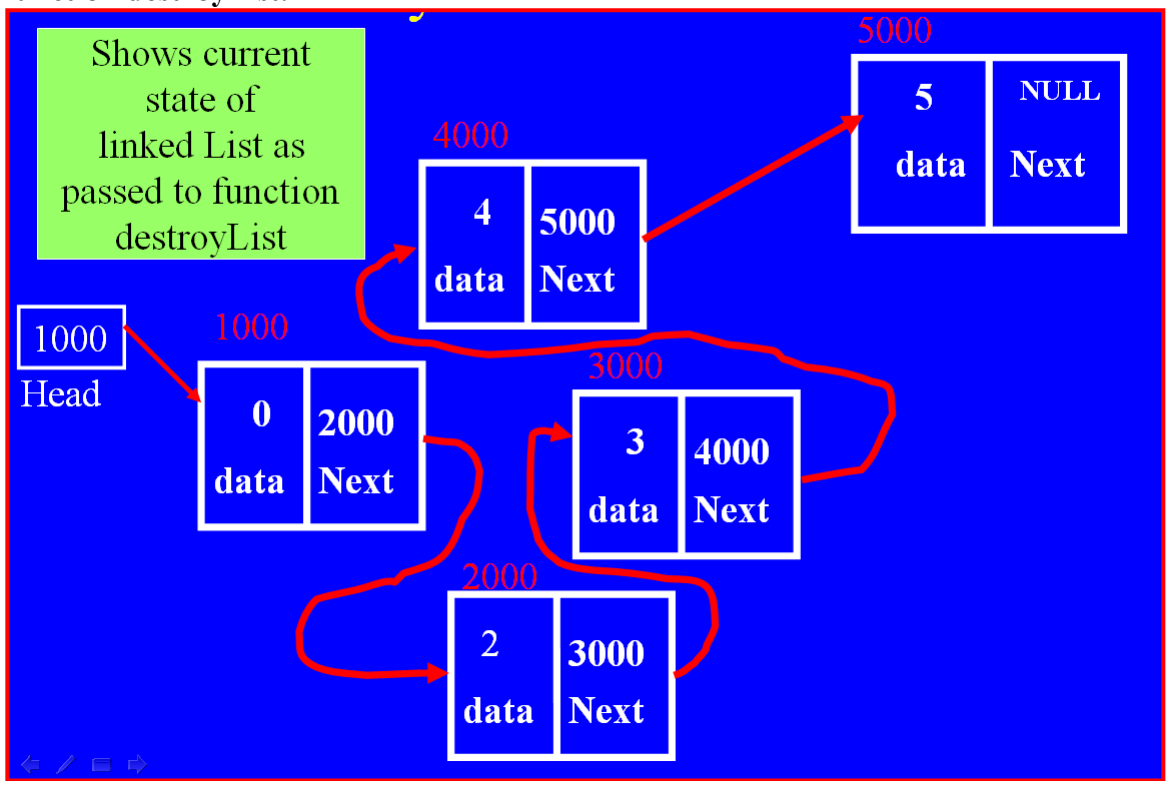


FIG. 6.8

Then the following three code lines are executed, which in affect assign two new pointers to the first node in the list. These two new pointers are Iter and IHead.

```
Node<Type> * Iter;
Node<Type> * IHead = Head;
Iter = IHead;
```

Situation at that point is shown by the Figure 6.9.

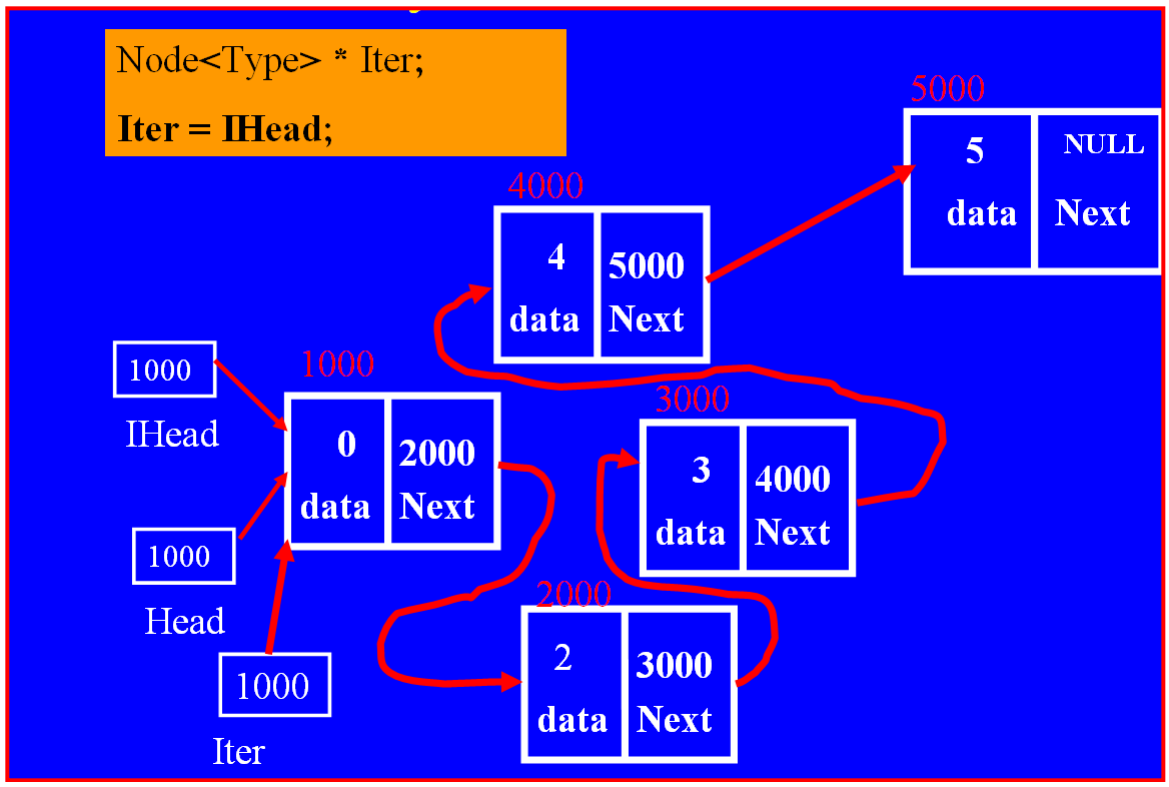


FIG. 6.9

The function follows the following algorithm to de-allocate the memory bound in linked list. Starting from situation similar to Figure 6.9, inside a loop first the pointer IHead is moved to the next node. Then pointer Iter is used to delete its pointee. Finally Iter is moved to point to where IHead is pointing. Figure 6.10 shows loop tasks sequentially. First IHead is moved to next node (Figure 6.10A).

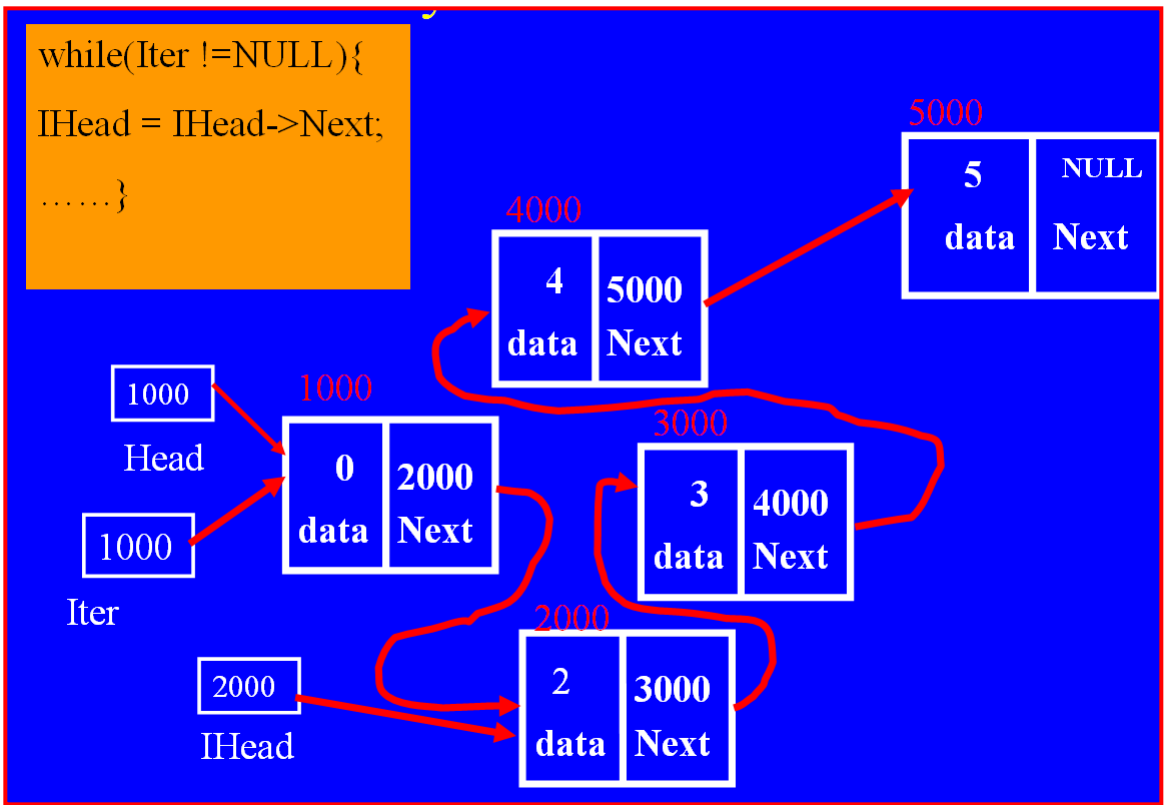


FIG. 6.10A

Then since Iter is now pointing to the previous node this node is deleted by invoking the delete operator (Figure 6.10B).

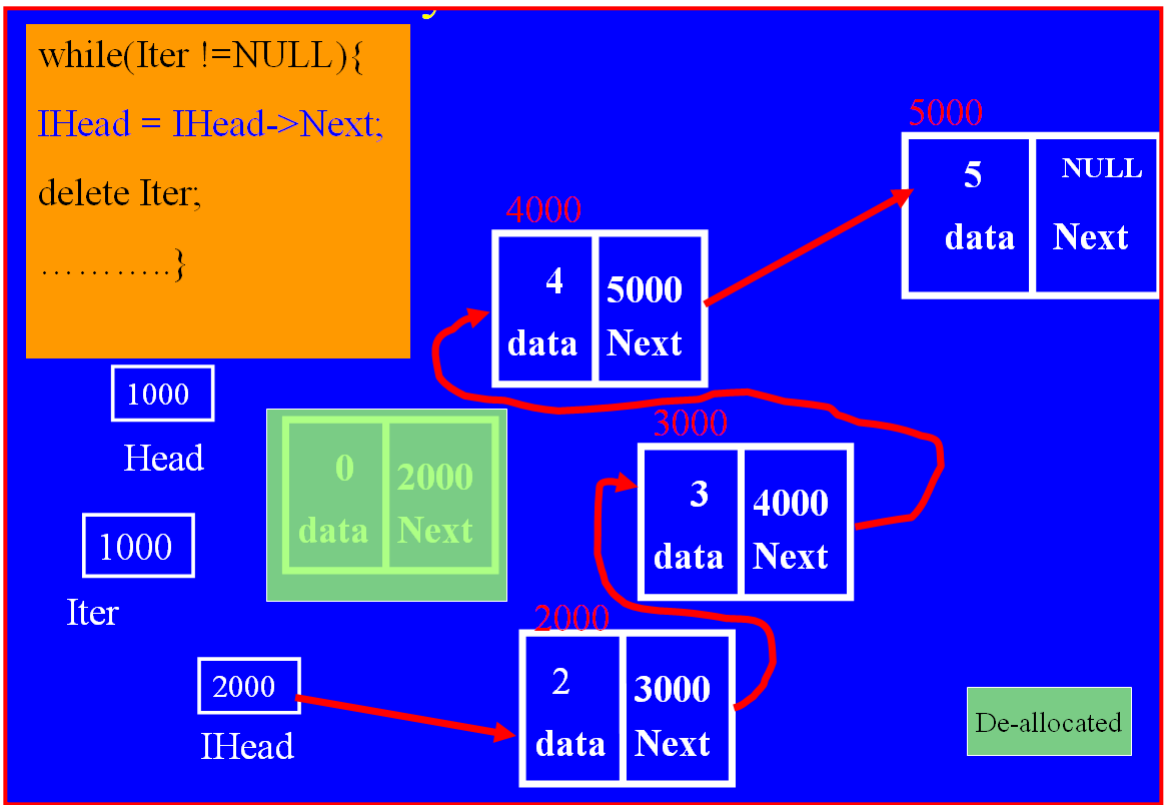


FIG. 6.10B

Understand that after deletion Iter and Head pointers would still have previous addresses stored in them. However, the memory at that address is returned back to the operating system and is no longer the part of our program. In third and last loop task the pointer Iter is made to point to where IHead is pointing (Figure 6.10C).

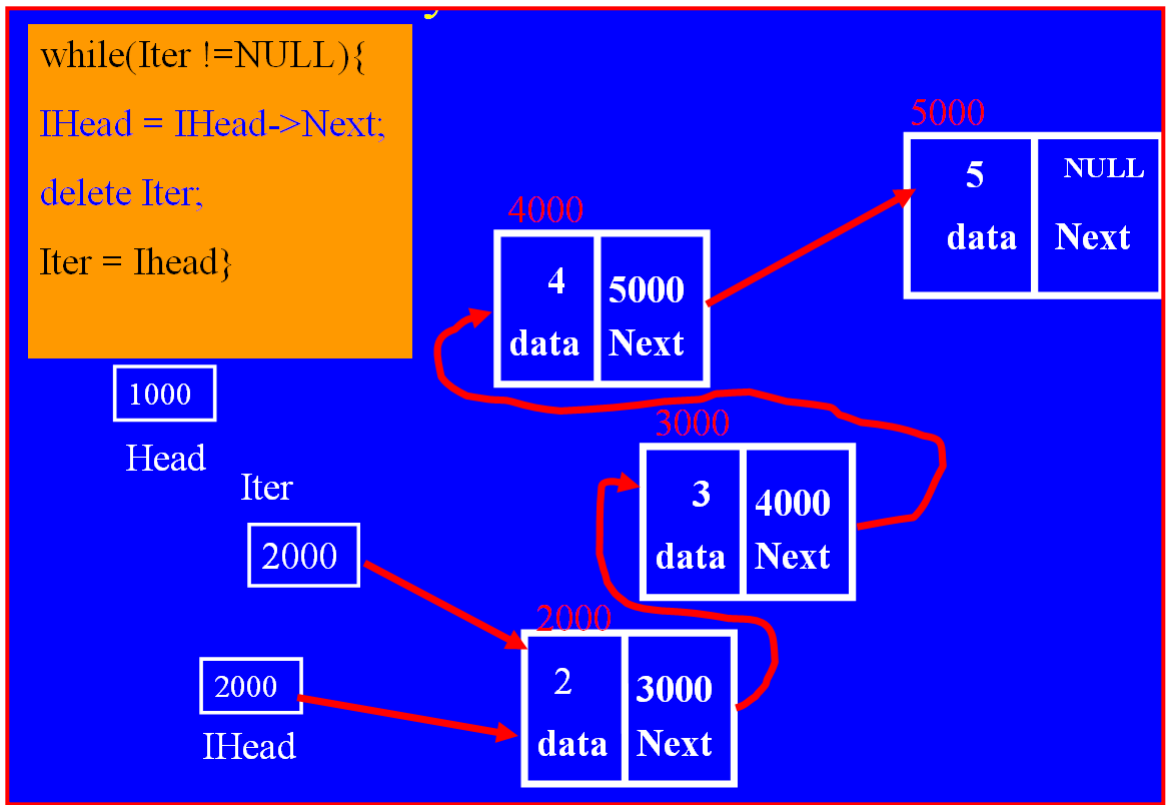


FIG. 6.10C

Now loop pretest is done again. Since Iter is not NULL, the three loop tasks would be done again and this time the node would data value 2 would be de-allocated. This process continues until all nodes are de-allocated and the pointer Iter stores a NULL value which is also the Next field of the last node. Figure 6.10D shows the situation just before de-allocating the last node.

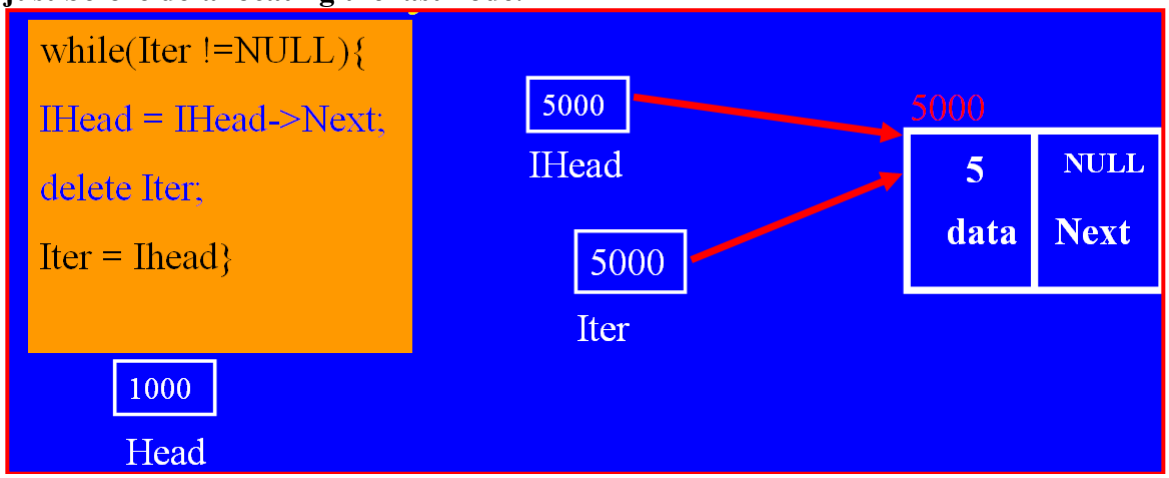


FIG. 6.10D

At this point code line

IHead = IHead->Next ;

would store NULL in IHead. Then code line

delete Iter;

would de-allocate the last node. Finally the code line

Iter = IHead;

would store NULL in it and then loop pre-test evaluates to false and the loop is exited.

It is possible to write a linked list program using a modified Node class where data member is a void pointer and even for data members the memory is allocated dynamically. Appendix 6A shows the listing for this generic linked list using the void pointer.

Generic Linked List

The linked list built above is ad-hoc in the nature as it can only store integers, double and strings. In addition the code blocks to create, print and de-allocate list for each data type are similar and tediously repetitive in nature. Also the functions to add data to the list, print the list, and destroy the list are generated as stand-alone functions and they are not neatly wrapped as class members. C++ provides us the power and flexibility to create objects that will facilitate generation of linked list, which can store primitives and objects. We follow the same design philosophy to design a generic linked list as we did for designing the data structures stacks and queues. First, we express our design through an abstract class called LinkedListInterface that details the functions that every linked list type class deriving from it must have. The Table 6.3 gives the summary of the interface functions and then Listing 6.2 provides most of the code details. Class name declaration is as follows:

template<typename Type>
class LinkedListInterface< Type >

Function Summary	
`virtual void`	**addFront (Type Item)=0** **Function: Transforms the `List` by adding an object to its front** **Effect: List size is increased by one** **Postcondition: List size = List Size + 1**
`virtual void`	**`addRear (Type Item)=0`** **Function: Transforms the `List` by adding an element to its rear.** **Effect: List size is increased by one** **Postcondition: List size = List Size + 1**

virtual bool	contains (const Type &Item) const=0 Function: Inspects the List to see if the given element is in the List Effect: Returns true if Item is in list otherwise returns false. The List is unaltered.
virtual const Type	getFront () const=0 Function: Observes the List and returns a copy of the front element. Effect: The List is unaltered.
virtual boolean	isEmpty() const = 0 Function: Observes the List to see if it is empty or not. Returns true if List is empty else returns false. Effect: The List is unaltered.
virtual const Type	getRear () const=0 Function: Observes the List and returns a copy of the rear element. Effect: The List is unaltered.
virtual void	removeFront() = 0 Function: Transforms the List by removing an element from its front Effect: List size is decreased by one Postcondition: List size = List Size - 1
virtual void	removeRear() = 0 Function: Transforms the List by removing an element from its rear Effect: List size is decreased by one Postcondition: List size = List Size - 1
virtual int	size() const = 0 Function: Observes and determines the number of objects in the List container Effect: Returns the number of elements in the List. The list is unaltered.

	LinkedListInterface () Default constructor
virtual	~LinkedListInterface () Destructor

Table 6.3

Notice that in a generic linked list we may wish to have functionality that would allow us to add and remove elements to the front or rear, or allow us to peek at the front and rear elements. Therefore, our List interface has the facilitating functions. Observer functions such as isEmpty and size ( ) are needed to find the List size or to see if List is empty. Notice that abstract class LinkedListInterface specifications in no way constrain the user as to how the links are made between the elements of the List. The links may point only to their followers – as singly linked list or links may be established both (to predecessor and follower) – like doubly linked list. These are implementation details, which are left to the implementer of the List interface. Listing 6.2 provides the code for the LinkedListInterface class.

```
00001 #ifndef LINKED_LIST_INTERFACE_H
00002 #define LINKED_LIST_INTERFACE_H
00003
00004 #include <iostream>
00005 #include <fstream>
00006 #include <string>
00007 #include <typeinfo>
00008 using namespace std;
00013 static int counter = 0;

00072 template<typename Type>
00073 class LinkedListInterface
00074 {
00075 public:
00079         virtual void addFront(Type Item)=0;
00083         virtual void addRear(Type Item)=0;
00087         virtual void removeFront()=0;
00091         virtual void removeRear()=0;
00095         virtual const Type getFront() const =0;
00099         virtual const Type getRear()const =0;
00103       virtual bool contains(const Type& Item)const=0;
00107         virtual int size() const=0;
00111         virtual bool isEmpty() const =0;
00115         LinkedListInterface() {}
00119         virtual ~LinkedListInterface() { }
00120 };
```

//Since all code would be in a single file, the includes are common to all classes.

Listing 6.2

Singly Linked List class

Now we write a class called SinglyLinkedList, which is also a templated class and it inherits from base class LinkedListInterface and also uses the Node class discussed earlier. Table 6.4 gives the summary of fields, constructors and functions for the class SinglyLinkedList and the detailed code is given in the Listing 6.3. Figure below gives the inheritance diagram for the SinglyLinkedList class.

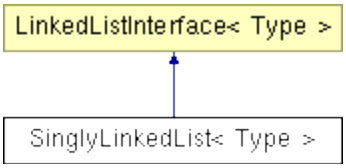


FIG. 6.11

As a matter of fact in designing all we need to add are the data members and any new member functions. Since SinglyLinkedList is a concrete class derived from abstract class LinkedListInterface, former would implement all the pure virtual functions of latter. Thus concrete class SinglyLinkedList derives from the LinkedListInterface abstract class as a public derivation and provides the implementation of all of the base class pure virtual functions.

template<typename Type>
class SinglyLinkedList< Type >

Public Member Functions	
	SinglyLinkedList ( ) *Default constructor for the linked list.*
	SinglyLinkedList (const SinglyLinkedList &Other) *Copy constructor for the linked list.*
const SinglyLinkedList &	operator= (const SinglyLinkedList &Other) *Overloaded assignment operator for linked list*
void	destroy ( ) *Method deletes all dynamically allocated memory in the list.*
virtual	~SinglyLinkedList ( )
void	addFront (Type Item)
void	addRear (Type Item)
void	removeFront ( )
void	removeRear ( )
const Type	getFront ( ) const
const Type	getRear ( ) const
bool	contains (const Type &Item) const
int	size ( ) const
bool	isEmpty ( ) const

Inherited from base class

void	**printForward** (ostream &out) const *Prints the list from front to back.*
void	**printBackward** (ostream &out) const *Prints the list from back to front.*
void	**getList** (istream &in, int flag) *Makes the list either from user input or by reading data from a file.*
void	**addAfter** (Type obj, Type target) *Inserts the obj, after the target in the list.*
void	**addBefore** (Type obj, Type target) *Inserts the obj before the first occurrence of target as scanning the list forward from the Head node.*
void	**removeAfter** (Type target) *Removes the Node after the Node containing target, when the list is scanned forward from the Head node.*
void	**removeBefore** (Type target) *Removes the Node before the Node containing target, when the list is scanned forward from the Head node.*
const Type	**elementAfter** (Type target) const *Returns the data value in the Node after the Node which contains the data value target.*
const Type	**elementBefore** (Type target) const *Returns the data value in the Node before the Node which contains the data value target.*
void	**printForwardRecursively** (ostream &out) const *Prints linked list from front to rear using a recursive print function.*
Protected Member Functions	
void	**copy** (const SinglyLinkedList &Other) *Copies the linked list Other to the caller object.*
const Node< Type > *	**getNode** (Type value) const *Gets the pointer to the first node containing the data field as value, while scanning the list forward from Head pointer.*
const Node< Type > *	**nodeBefore** (const Node< Type > *Some_Node) const *Returns a pointer to the Node before Some_Node if found or returns NULL if not found.*

Private Attributes or Data Members	
Node< Type > *	**Head** *Pointer to the head or front node of linked list.*
Node< Type > *	**Rear** *Pointer to the rear node or last node in the linked list.*
Node< Type > *	**Iterator** *Pointer which, can iterate through the linked list as needed.*
int	**num_elements** *Number of elements in the linked list.*
Friend Functions	
ostream &	operator<< (ostream &out, const SinglyLinkedList &SL) *Prints the list in forward order only.*
istream &	operator>> (istream &in, SinglyLinkedList &SL) *Reads the data from file only to build the linked list and adds the data to linked list adding them to the rear of the list.*
const SinglyLinkedList	operator+ (const SinglyLinkedList &List1, const SinglyLinkedList &List2) *Merges two or more lists together and returns the merged list.*

Table 6.4

Listing 6.3 below gives the header and implementation of SinglyLinkedList class. Some member functions are not implemented as we require them to be implemented as part of laboratory exercise #7.

```
00127 template<typename Type>
00128 class SinglyLinkedList:public LinkedListInterface<Type>
00129 {
00130         private:
00134                 Node<Type> * Head;
00138                 Node<Type>* Rear;
00142                 Node<Type>* Iterator;
00146                 int num_elements;
00147         public:
00151                 SinglyLinkedList();
00155                 SinglyLinkedList(const SinglyLinkedList& Other);
   const SinglyLinkedList& operator = (const SinglyLinkedList& Other);
00163                 void destroy();
00164                 /*
00165                 *Virtual destructor for the linked List.
00166                 */
00167                 virtual ~SinglyLinkedList();
00168
```

```
00169                   void addFront(Type Item);
00170                   void addRear(Type Item);
00171                   void removeFront();
00172                   void removeRear();
00173                   const Type getFront()const;
00174                   const Type getRear()const;
00175                   bool contains(const Type& Item)const;
00176                   int size()const;
00177                   bool isEmpty()const;
00181                   void printForward(ostream& out)const;
00185                   void printBackward(ostream& out)const;
00189                   void getList(istream& in, int flag);
00193                   void addAfter(Type obj, Type target);
00198                   void addBefore(Type obj, Type target);
00205                   void removeAfter(Type target);
00212                   void removeBefore(Type target);
00219                   const Type elementAfter(Type target) const;
00226                   const Type elementBefore(Type target) const;
00230                   void printForwardRecursively(ostream& out)const;
00231           protected:
00235                   void copy(const SinglyLinkedList& Other);
friend ostream& operator <<(ostream& out, const SinglyLinkedList& SL);
friend istream& operator >>(istream& in,  SinglyLinkedList& SL);
00249                   const Node<Type> * getNode(Type value) const;
00254const Node<Type> * nodeBefore(const Node<Type> * Some_Node) const;
friend const SinglyLinkedList operator + ( const SinglyLinkedList &
List1, const SinglyLinkedList& List2);
00260 }; //End of class SinglyLinkedList

00262 //Code for below functions is not given as that would be part of
//Lab 7.
00264 template<typename Type>
const SinglyLinkedList<Type> operator +
(const SinglyLinkedList<Type> & List1, const SinglyLinkedList<Type>&
List2)
00267 {
00268         //Complete as part of Lab 7
00274
00275 }
00277 template<typename Type>
00278 void SinglyLinkedList<Type>::removeAfter(Type target)
00279 {
00280         //Complete as part of Lab 7
00281 }
00283 template<typename Type>
00284 void SinglyLinkedList<Type>::removeBefore(Type target)
00285 {
00286         //Complete as part of Lab 7
00287 }
00289 template<typename Type>
00290 void SinglyLinkedList<Type>::copy(const SinglyLinkedList<Type>&
Other)
00291 {
00292         //Complete as part of Lab 7
00293 }
```

```
00295 //Lab7 Assignment finished.
00297 template<typename Type>
00298 SinglyLinkedList<Type>::SinglyLinkedList()
00299 {
00300         Head = NULL;
00301         Rear = NULL;
00302         Iterator = NULL;
00303         num_elements = 0;
00304 }
00306 template<typename Type>
00307 SinglyLinkedList<Type>::SinglyLinkedList(const
SinglyLinkedList<Type>& Other)
00308 {
00309         if(this == &Other)
00310         {
cerr<<"Self copying is illegal and unnecessary. Exiting the program.\n";
00312                 exit(1);
00313         }
00314         copy(Other);
00315 }
00317 template<typename Type>
00318 const SinglyLinkedList<Type>& SinglyLinkedList<Type>::
00319 operator = (const SinglyLinkedList<Type> & Other)
00320 {
00321         if(this == &Other)
00322                 return *this;
00323         destroy();
00324         copy(Other);
00325         return *this;
00326 }
00328 template<typename Type>
ostream& operator <<(ostream& out, const SinglyLinkedList<Type>& SL)
00330 {
00331         SL.printForward(out);
00332         return out;
00333 }
00335 template<typename Type>
00336 void SinglyLinkedList<Type>::addFront(Type Item)
00337 {
00338         Node<Type> * New_Node = new Node<Type>(Item , Head);
00339         counter++;
00340         Head = New_Node;
00341         if(num_elements == 0)
00342                 Rear = Head;
00343         num_elements++;
00344 }
00346 template<typename Type>
00347 void SinglyLinkedList<Type>::addRear(Type Item)
00348 {
00349         Node<Type> * New_Node = new Node<Type>(Item, NULL);
00350         counter++;
00351
00352         if(num_elements == 0)
00353         {
00354                 Head = New_Node;
00355                 Rear = New_Node;
```

1. Default constructor

2. Copy Constructor

3. Overloaded assignment operator

4. Overloaded insertion operator

// For first node, Head and Rear must point //to same node.

```
00356         }

00357         else
00358         {
00359                 Rear->Next = New_Node;
00360                 Rear = New_Node;
00361         }
00362         num_elements++;
00363 }
00365 template<typename Type>
00366 void SinglyLinkedList<Type>::printForward(ostream& out)const
00367 {
00368         Node<Type> * Iterator1 = Head;
00369
00370         while(Iterator1 !=NULL)
00371         {
00372                 out<<Iterator1->data<<"  ";
00373                 Iterator1 = Iterator1->Next;
00374         }
00375         cout<<endl;
00376 }
00378 template<typename Type>
00379 bool SinglyLinkedList<Type>::isEmpty() const
00380 {
00381         return num_elements == 0 ;
00382 }
00384 template<typename Type>
00385 int SinglyLinkedList<Type>::size() const
00386 {
00387         return num_elements;
00388 }
00390 template<typename Type>
00391 void SinglyLinkedList<Type>::removeFront()
00392 {
00393         if(isEmpty())
00394         {
00395           cerr<<"The linked list is empty. Program will end.\n";
00396                 return;
00397         }
00398
00399         Iterator = Head;
00400         Head = Head->Next;
00401         delete Iterator;
00402         counter--;
00403         num_elements--;
00404
00405         if(num_elements == 0)
00406                 Rear = NULL;
00407 }
00409 template<typename Type>
00410 void SinglyLinkedList<Type>::removeRear()
00411 {
00412         if(isEmpty())
00413         {
00414           cerr<<"The linked list is empty. Program will end.\n";
00415                 exit(1);
```

```
00416         }
00417         else if(num_elements == 1)
00418         {
00419                 delete Head;
00420                 counter--;
00421                 Head = NULL;
00422                 Rear = NULL;
00423         }
00424         else
00425         {
00426                 Iterator = Head;
00427
00428                 while(Iterator->Next != Rear)
00429                         Iterator = Iterator->Next;
00430
00431                 delete Rear;
00432                 counter--;
00433                 Rear = Iterator;
00434                 Iterator->Next = NULL;
00435         }
00436
00437         num_elements--;
00438 }
00440 template<typename Type>
00441 const Type SinglyLinkedList<Type>::getFront() const
00442 {
00443         if(isEmpty())
00444         {
00445           cerr<<"The linked list is empty. Program will end.\n";
00446                 exit(1);
00447         }
00448

00449         return Head->data;
00450 }
00452 template<typename Type>
00453 const Type SinglyLinkedList<Type>::getRear() const
00454 {
00455         if(isEmpty())
00456         {
00457           cerr<<"The linked list is empty. Program will end.\n";
00458                 exit(1);
00459         }
00460
00461         return Rear->data;
00462 }
00464 template<typename Type>
00465 bool SinglyLinkedList<Type>::contains(const Type & Item) const
00466 {
00467         Node<Type> * Iterator1 = Head;
00468
00469         while(Iterator1 != NULL && !(Iterator1->data == Item))
00470                 Iterator1 = Iterator1->Next;
00471
00472         return Iterator1 != NULL;
00473 }
```

```
00475 template<typename Type>
00476 void SinglyLinkedList<Type>::destroy()
00477 {
              If(num_elements == 0)
                   return;
00478         Iterator = Head;
00479
00480         while(Iterator !=NULL)
00481         {
                      Head = Head->Next;
00483                 delete Iterator;
00484                 counter--;
00485                 Iterator = Head;
00486         }
00487         Rear = NULL;
00488         Iterator = NULL;
00489         num_elements = 0;
00490 }
00492 template<typename Type>
00493 SinglyLinkedList<Type>::~SinglyLinkedList()
00494 {
00495         destroy();
00496 }
00498 template<typename Type>
00499 void SinglyLinkedList<Type>::printBackward(ostream& out)const
00500 {
00508         for(int index = num_elements; index>0; index--)
00509         {
00510                 Node<Type>* Iterator1 = Head;
00511
00512                 int count = index;
00513                 while(--count)
00514                         Iterator1 = Iterator1->Next;
00515                 out<<Iterator1->data<<"  ";
00516         }
00517         out<<endl;
00518 }
00520 template<typename Type>
00521 const Node<Type> * SinglyLinkedList<Type>::getNode(Type value)
const
00522 {
00523         Node<Type> * Iterator1 = Head;
00524         Node<Type> * Result = NULL;
00525
00526         while(Iterator1 != NULL)
00527         {
00528                 if(Iterator1->data == value)
00529                 {
00530                         Result = Iterator1;
00531                         break;
00532                 }
00533                 Iterator1 = Iterator1->Next;
00534         }
00535
00536         return Result;
00537 }
```

```
00539 template<typename Type>
00540 const Node<Type> * SinglyLinkedList<Type>::
00541 nodeBefore(const Node<Type> * Some_Node) const
00542 {
00543         if(Some_Node != NULL && Some_Node != Head)
00544         {
00545                 Node<Type> * Previous = Head;
00546
00547                 while(Previous->Next != Some_Node)
00548                         Previous = Previous->Next;
00549                 return Previous;
00550         }
00551         else
00552                 return NULL;
00553 }
00555 template<typename Type>
00556 void SinglyLinkedList<Type>::addAfter(Type obj, Type target)
00557 {
00558         if(contains(target) && !contains(obj))
00559         {
00560                 Node<Type> * itemNode = getNode(target);
00561     Node<Type> * newNode = new Node<Type>(obj, itemNode->Next);
00562                 counter++;
00563                 itemNode->Next = newNode;
00564                 num_elements++;
00565
00566                 if(Rear == itemNode)
00567                         Rear = newNode;
00568         }
00569 }
00571 template<typename Type>
00572 void SinglyLinkedList<Type>::addBefore(Type obj, Type target)
00573 {
00574         if(contains(target) && !contains(obj))
00575         {
00576                 Node<Type> * itemNode = getNode(target);
00577             Node<Type> * newNode = new Node<Type>(obj, itemNode);
00578                 counter++;
00579
00580                 if(Head == itemNode)
00581                         Head = newNode;
00582                 else
00583                 {
00584                  Node<Type> * beforeNode = nodeBefore(itemNode);
00585                         beforeNode->Next = newNode;
00586                 }
00587
00588                 num_elements++;
00589         }
00590 }
00592 template<typename Type>
00593 const Type SinglyLinkedList<Type>::elementAfter(Type target)
const
00594 {
```

```
00595         if(!contains(target) || (getNode(target) == Rear))
00596         {
00597          cerr<<"The value "<<target<<" is not in the list or is "
00598          <<"the last value in the list. Exiting the program.\n";
00599                 exit(1);
00600         }
00601
00602         return ((getNode(target))->Next)->data;
00603 }
00605 template<typename Type>
00606 const Type SinglyLinkedList<Type>::elementBefore(Type target)
const
00607 {
00608         if(!contains(target) || (getNode(target) == Head))
00609         {
00610          cerr<<"The value "<<target<<" is not in the list or is "
00611         <<"the first value in the list. Exiting the program.\n";
00612                 exit(1);
00613         }
00614
00615         return (nodeBefore(getNode(target)))->data;
00616 }
00618 template<typename Type>
00619 void SinglyLinkedList<Type>::getList(istream& in, int flag)
00620 {
00621         bool done = false;
00622         if(flag==1)
00623         {
cout<<"The data will be added to the Head (front) of linked list.\n";
00625                 while(!done)
00626                 {
00627                         Type temp = Type();
00628       cout<<"Enter the data item to be added to linked list : ";
00629                         cin>>temp;
00630                         addFront(temp);
00631           cout<<"More data? enter 0 to continue or 1 to exit : ";
00632                         cin>>done;
00633                 }
00634         }
00635         else if(flag == 2)
00636         {
00637    cout<<"The data will be added to the Rear of linked list.\n";
00638                 while(!done)
00639                 {
00640                         Type temp = Type();
00641        cout<<"Enter the data item to be added to linked list : ";
00642                         cin>>temp;
00643                         addRear(temp);
00644           cout<<"More data? enter 0 to continue or 1 to exit : ";
00645                         cin>>done;
00646                 }
00647         }
00648         else if(flag == 3)
00649                 in>>*this;
00650         else
00651                 cerr<<"Illegal value of the flag is entered.\n";
```

```
00652 }
00654 template<typename Type>
00655 istream& operator >>(istream& in,  SinglyLinkedList<Type>& SL)
00656 {
00657         Type temp = Type();
00658
00659         in.peek();
00660
00661         while(!in.eof())
00662         {
00663                 in>>temp;
00664                 SL.addRear(temp);
00665         }
00666
00667         return in;
00668 }
00670 template<typename Type>
00671 void print(Node<Type>* Iterator1, ostream & out)
00672 {
00673         if(Iterator1 !=NULL)
00674         {
00675                 out<<Iterator1->data<<"  ";
00676                 print(Iterator1->Next, out);
00677         }
00678 }
00680 template<typename Type>
00681 void SinglyLinkedList<Type>::printForwardRecursively(ostream&
out)const
00682 {
00683         Node<Type> * Iterator1 = Head;
00684         print(Iterator1,out);
00685         cout<<endl;
00686 }
00688 #endif
```

//Listing 6.3

We would like to emphasize here that depending upon the compiler used link errors may be there that may be caused by the placement of friend functions. In that case the bodies of errant friend functions must be moved inside the class! For serious user of templates Microsoft compilers are not recommended.

Discussion of Functions from SinglyLinkedList Class

We discuss some of the important functions in the class SinglyLinkedList.

Function addRear (Type Item)

As seen in the specifications (Listing 6.3), the purpose of the function addRear ( ) is to add the element Item to the list, so that it becomes the last element in it. The power point presentation given below illustrates the mechanism of adding a node to the rear of list dynamically.

C:\Folderes from D\
_CoursesTaught\CS2

When an object is to be added to the list, then at that time list may be empty or it may have some elements in it. When the function addRear ( ) is invoked, first a New_Node is created which has the Item as its data member item and the Next field of New_Node is set to NULL (Figure 6.12).

```
Node<Type> * New_Node = new Node<Type>(Item, NULL);
if(num_elements == 0){
Head = New_Node;
Rear = New_Node;}
else{Rear->Next = New_Node;
        Rear = New_Node;       }
num_elements++;
```

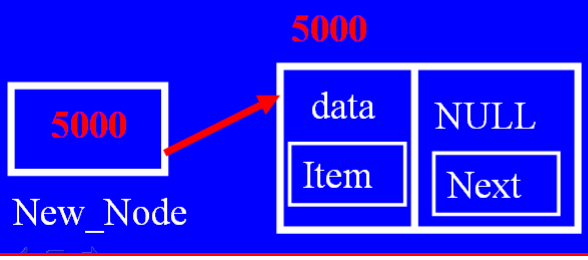


FIG. 6.11

If the list is empty, then num_elements will be zero and the assertion: num_elements == 0 is true. Therefore, the task in the scope of if structure is executed. The first statement in the if structure sets the Head and Rear pointers to point to pointee of New_Node (Figure 6.12). All three pointers, New_Node, Head, and Rear point to the first node in the list.

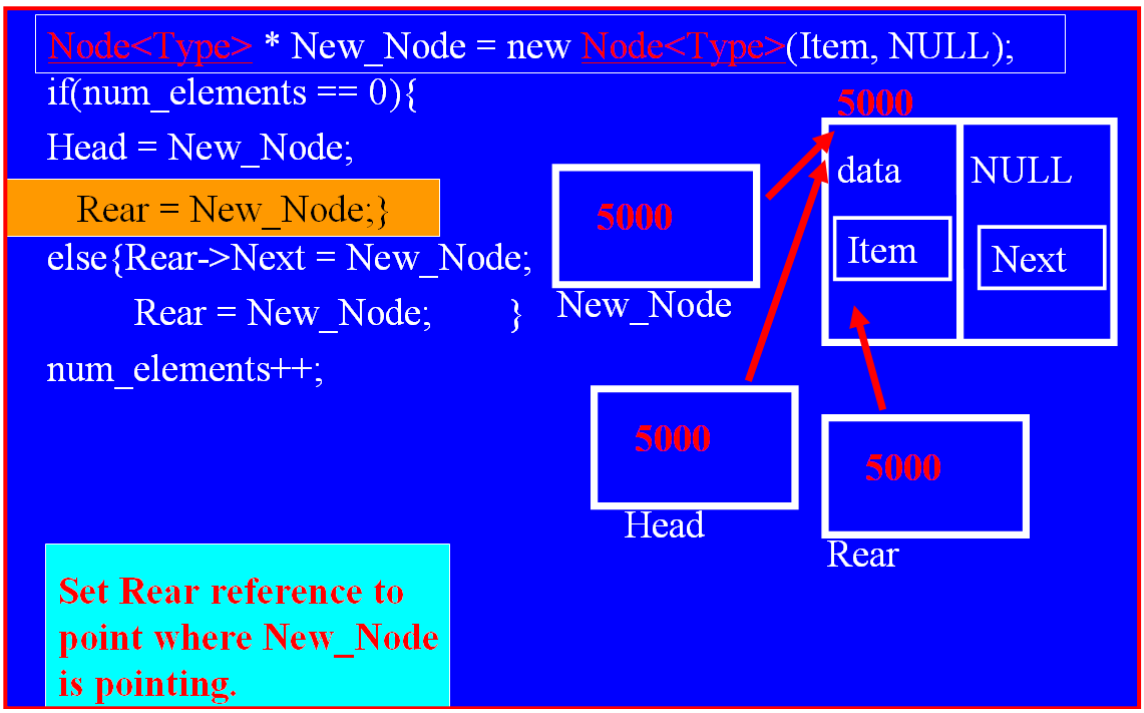


FIG. 6.12

The else clause is executed if the list already has one or more elements. Figure 6.13 shows schematically, a possible scenario.

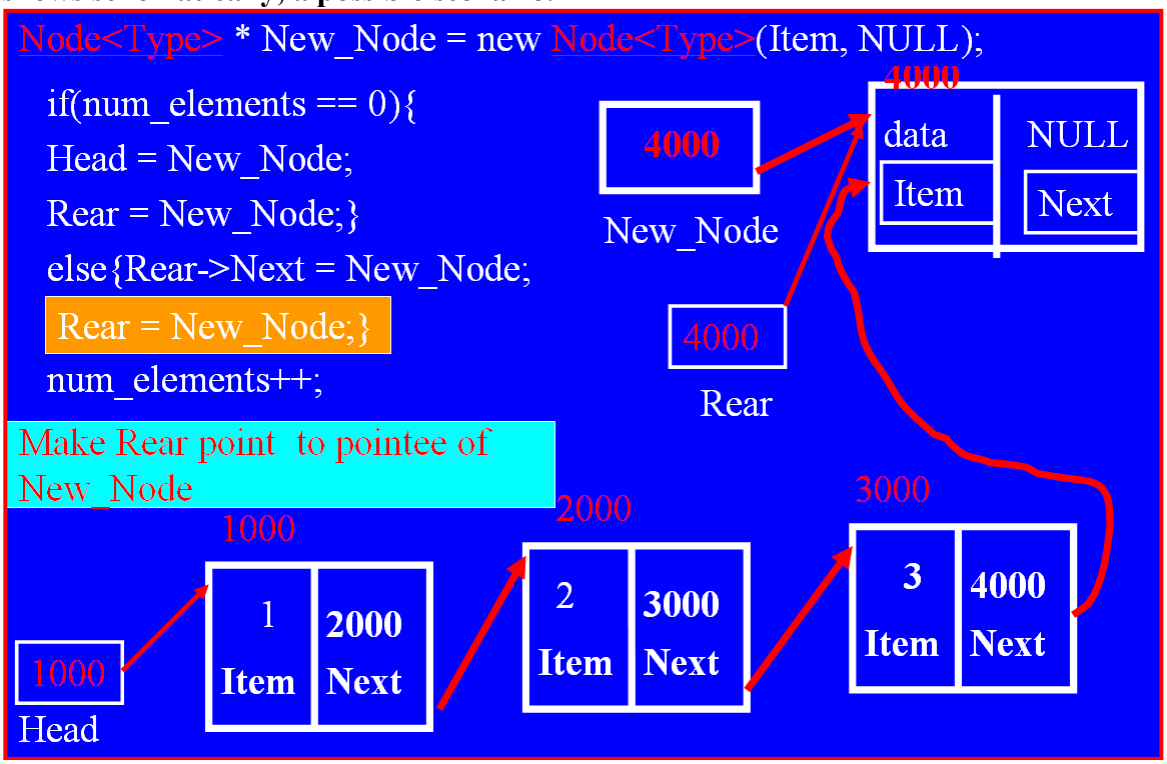


FIG. 6.13

The first statement in the else clause makes the Rear->next point to the pointee of New_Node. Thus Next pointer of current last node stores an address of 4000. Then the next statement (Rear = New_Node;) makes rear pointer to also point to the

pointee of New_Node. The two code lines safely connect the former last node and Rear pointer to the current last node. Once the task in the scope of either if or else clause is concluded, the num_elements is increased by one (Figure 6.14).

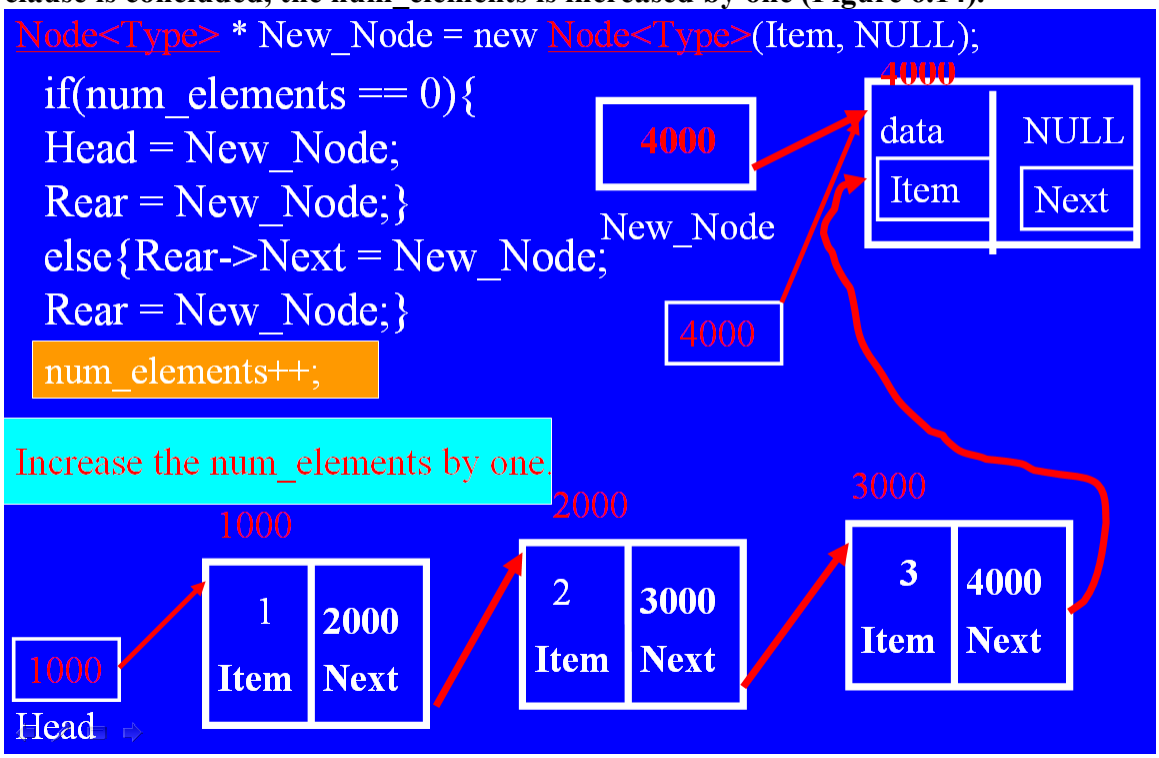


FIG. 6.14

The function addFront ( ) works in a manner exactly similar to addRear ( ).

Function removeRear ( )

Function removeRear ( ) takes no arguments and simply removes the rear element from the list. The PowerPoint presentation below shows the mechanism of function removeRear ( ) dynamically.

C:\Folderes from D\
_CoursesTaught\CS2

The function has three if / else blocks. Obviously if list is empty, then there is no element available for removal. Thus, the first block is executed and after a message "The linked list is empty. Program will end." is printed the function ends execution. (Figure 6.15A).

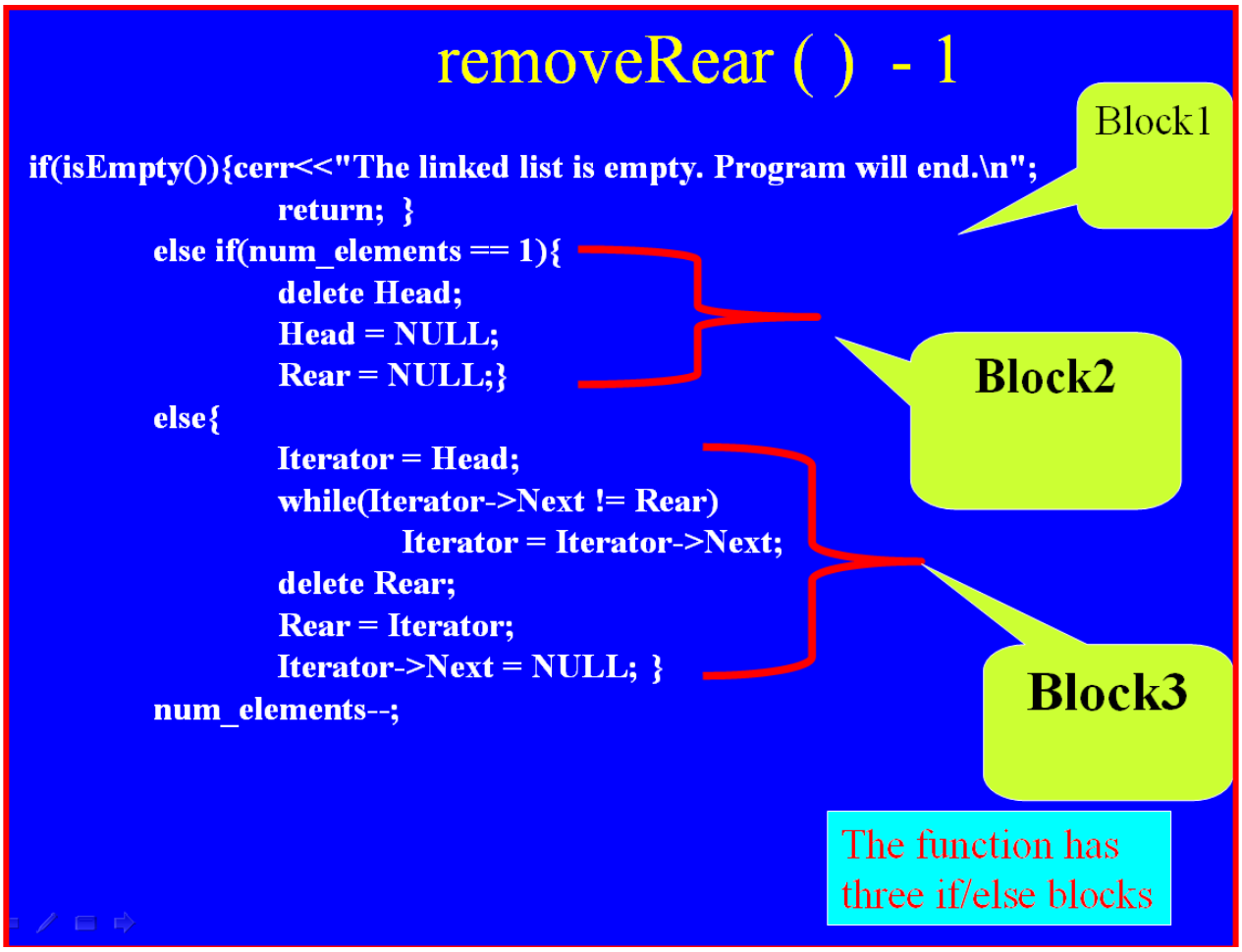

removeRear ( ) - 1
if(isEmpty()){cerr<<"The linked list is empty. Program will end.\n";
return; }
else if(num_elements == 1){
delete Head;
Head = NULL;
Rear = NULL;}
else{
Iterator = Head;
while(Iterator->Next != Rear)
Iterator = Iterator->Next;
delete Rear;
Rear = Iterator;
Iterator->Next = NULL; }
num_elements--;
Block1
Block2
Block3
The function has three if/else blocks

FIG. 6.15A

For non-empty list the removal scheme differs depending upon whether the list has only one element or more than one. If list has only one element, then the block 2 is entered (Figure6.15B).

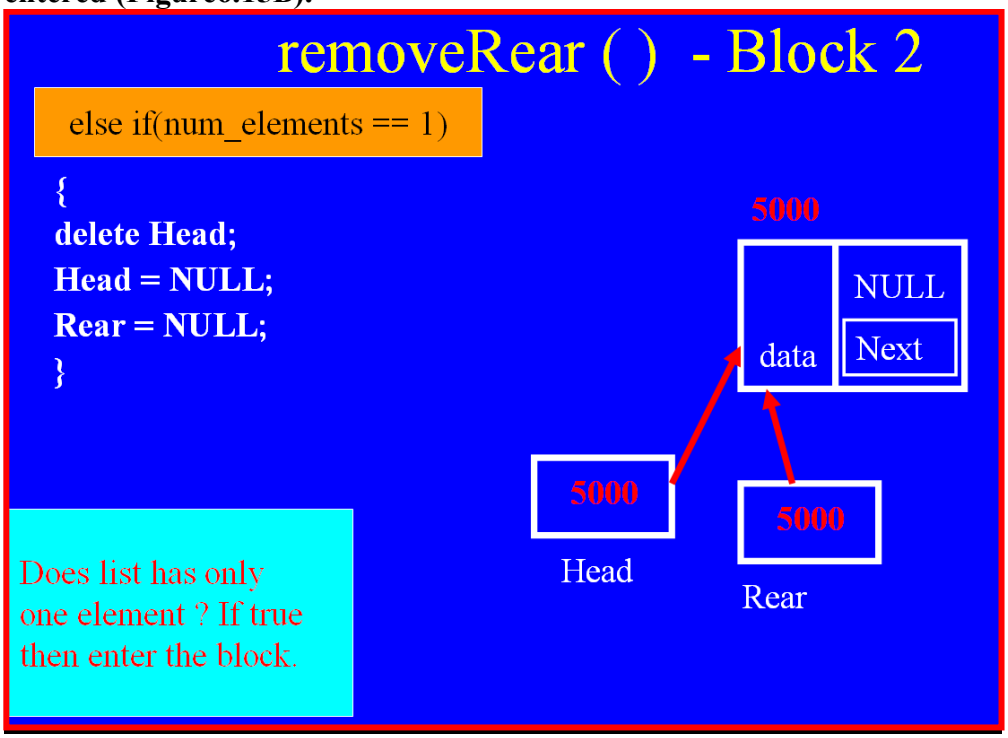


FIG. 6.15B

In block 2, first the pointee of Head is deleted (Figure 6.15C).

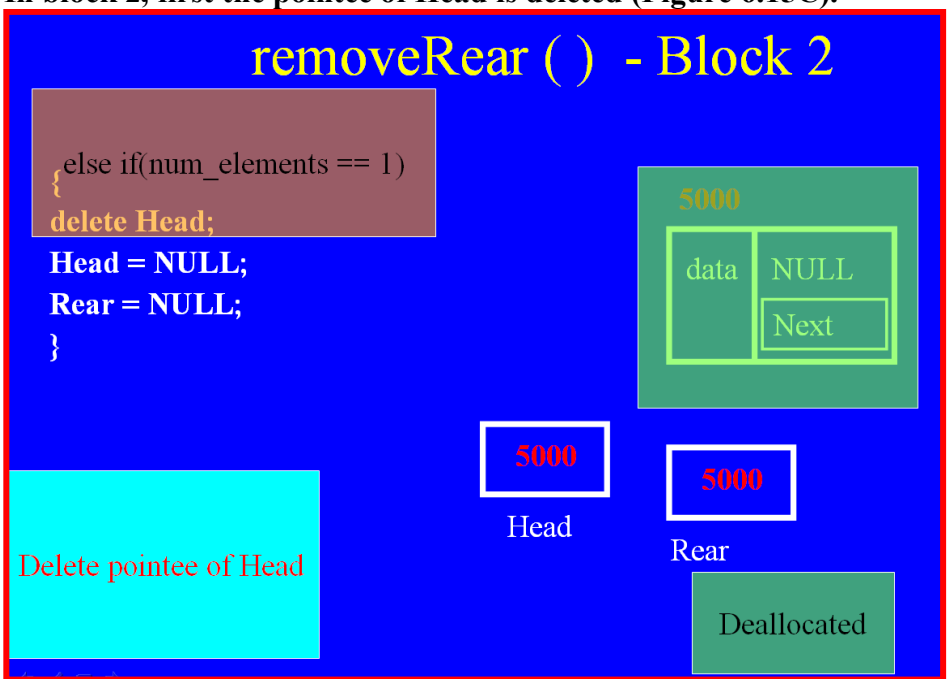


FIG. 6.15C

Then both Head and Rear pointers are set to NULL. (Figure 6.15D).

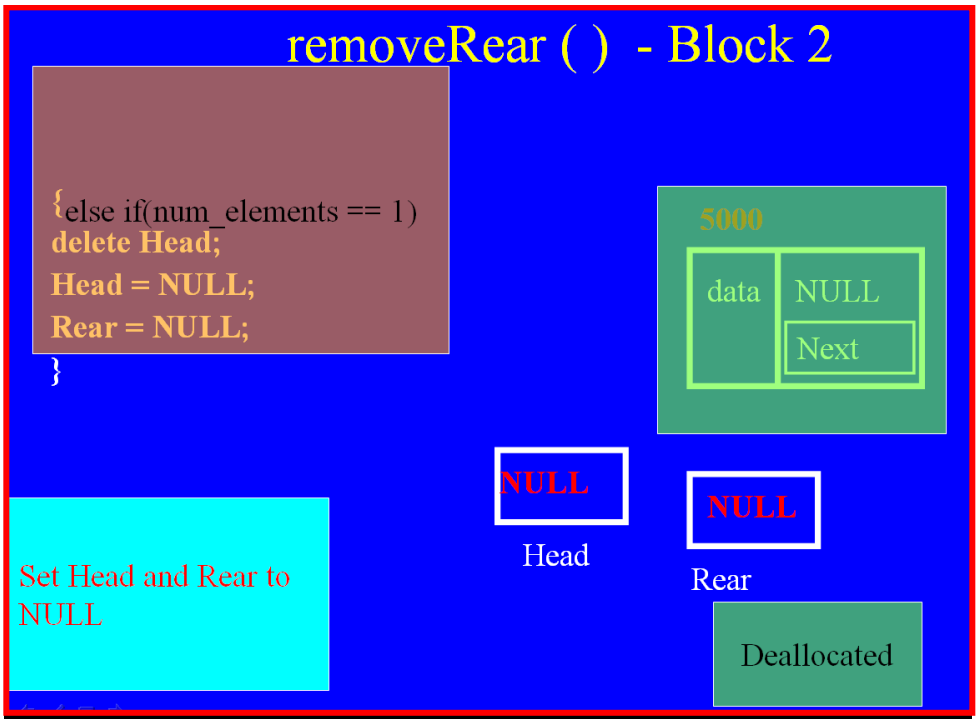


FIG. 6.15D

If the list has more than one element than the block 3 is entered (6.15E).

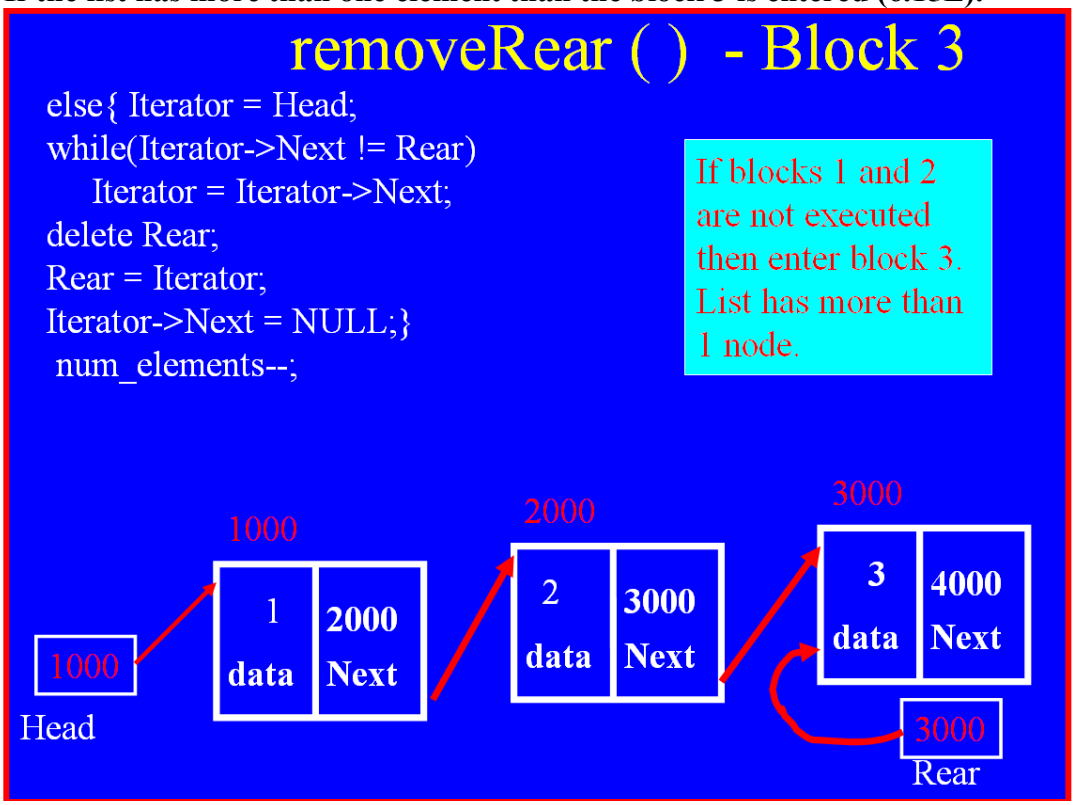


FIG. 6.15E

The first statement sets Iterator to point to the pointee of Head pointer (Figure 6.15F).

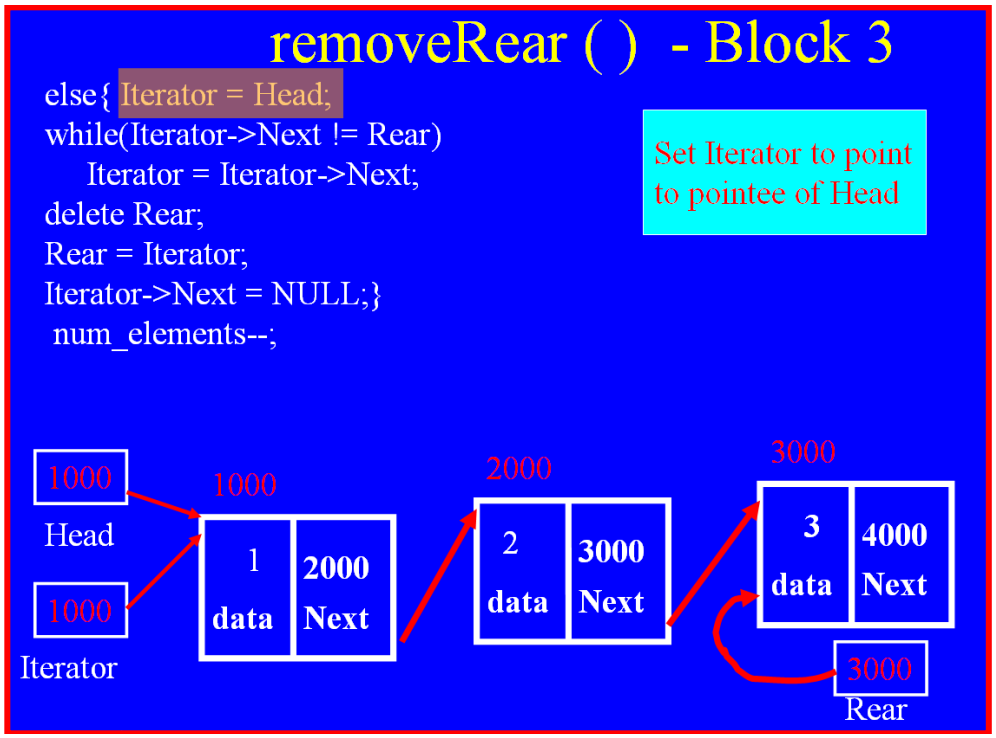


FIG. 6.15F

Now the pretest condition in the loop checks whether the pointer Iterator->Next is at last node or not? (Figure 6.15G).

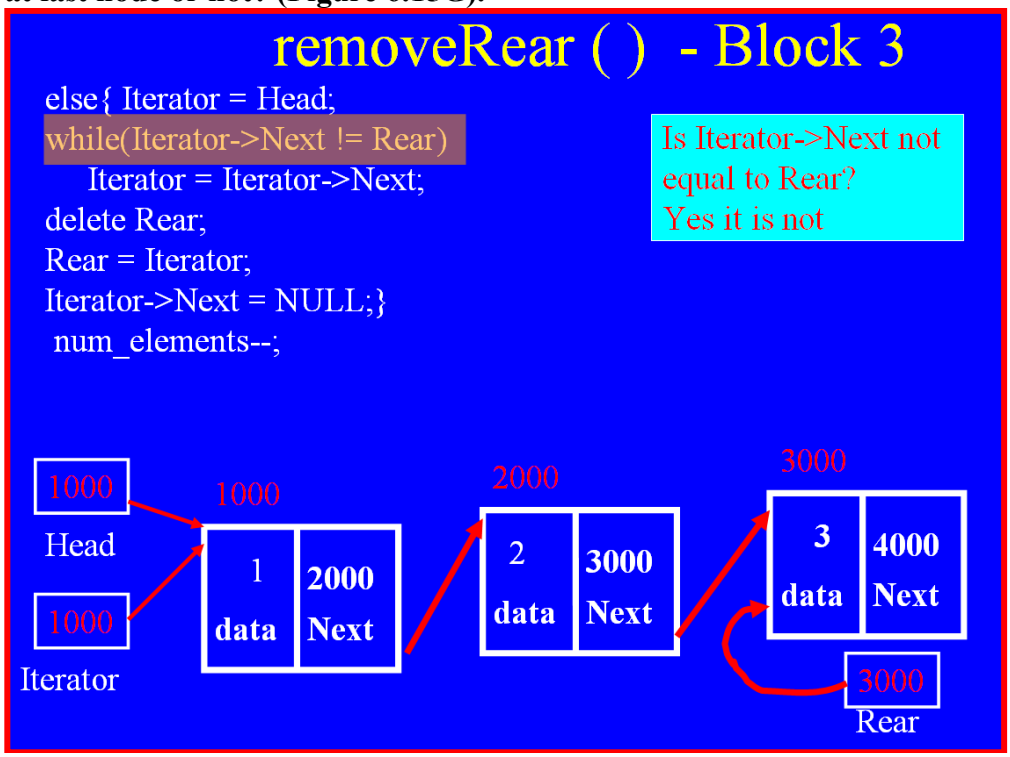


FIG. 6.15G

The purpose of the loop is to set the pointer Iterator at the node before the last node. After doing so we can safely remove the last node and then make Rear pointer to

point to the resultant last node. In Figure 6.15G, the pointer Iterator->Next is not at the last node. Therefore, the loop is entered. Inside the loop, the pointer Iterator is moved by one node (Figure 6.15H).

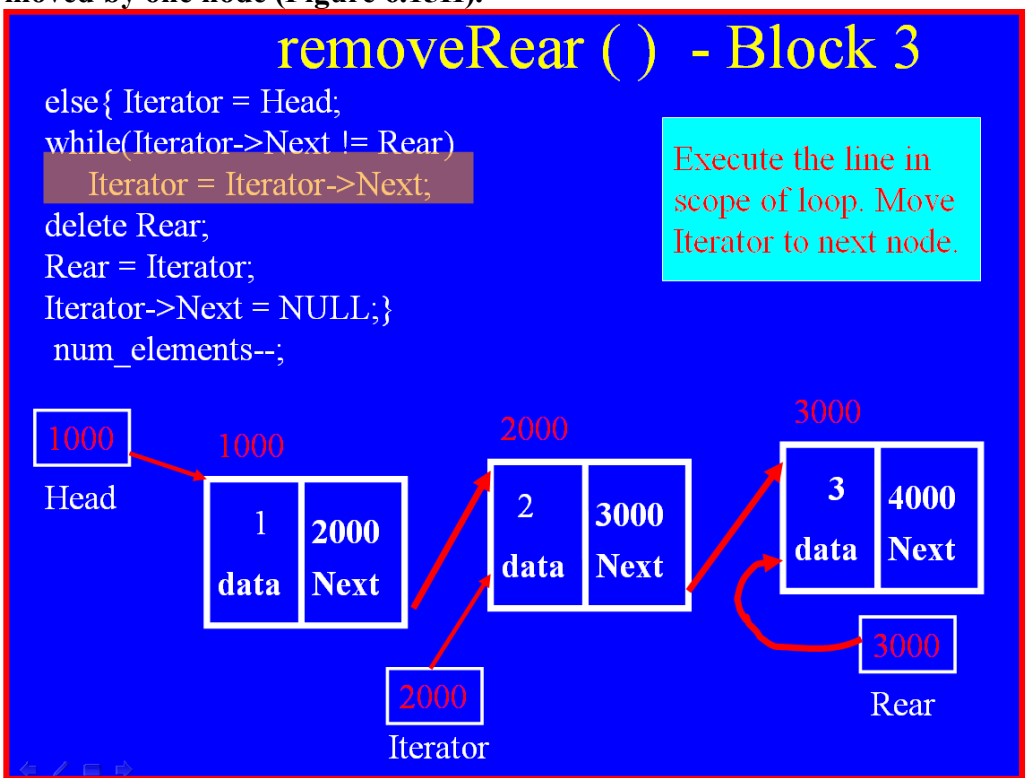


FIG. 6.15H

This time the loop pretest evaluates false because the Iterator->Next is pointing to where Rear pointer is pointing. Therefore, the loop is exited. The first statement after the loop deletes the pointee of Rear pointer (Figure 6.15I). This de-allocates the memory allocated for the last node.

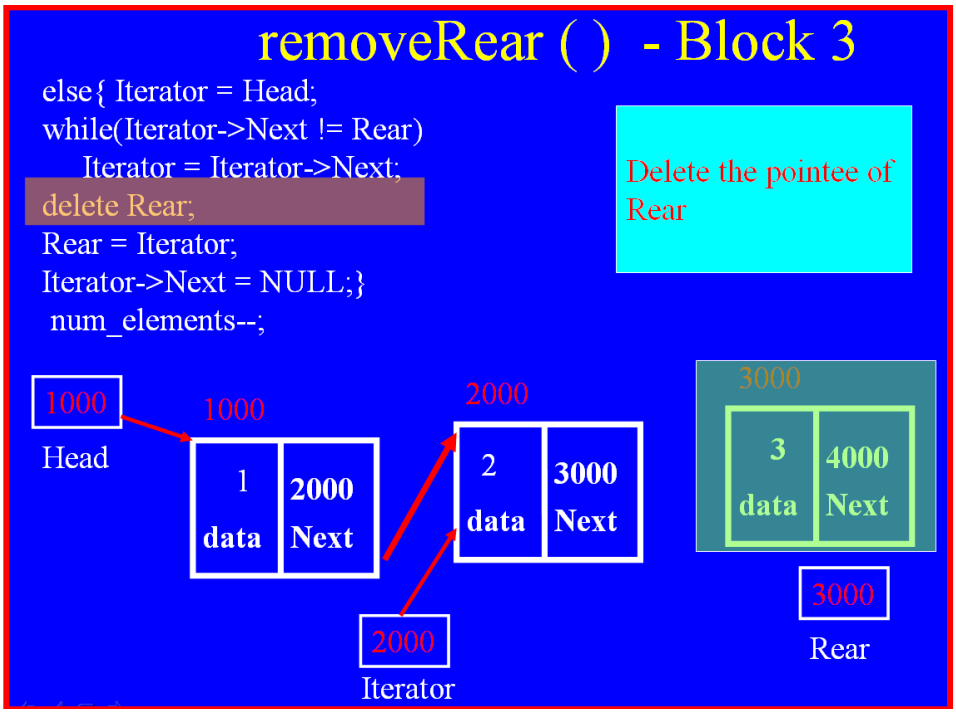


FIG. 6.15I

The Rear pointer is then set to point to where Iterator is pointing. Then the Next field of the pointee of Iterator (which is now the last node in the list) is set to NULL. This ends the block 3 and then the next statement reduces the list size by one (Figure 6.15J).

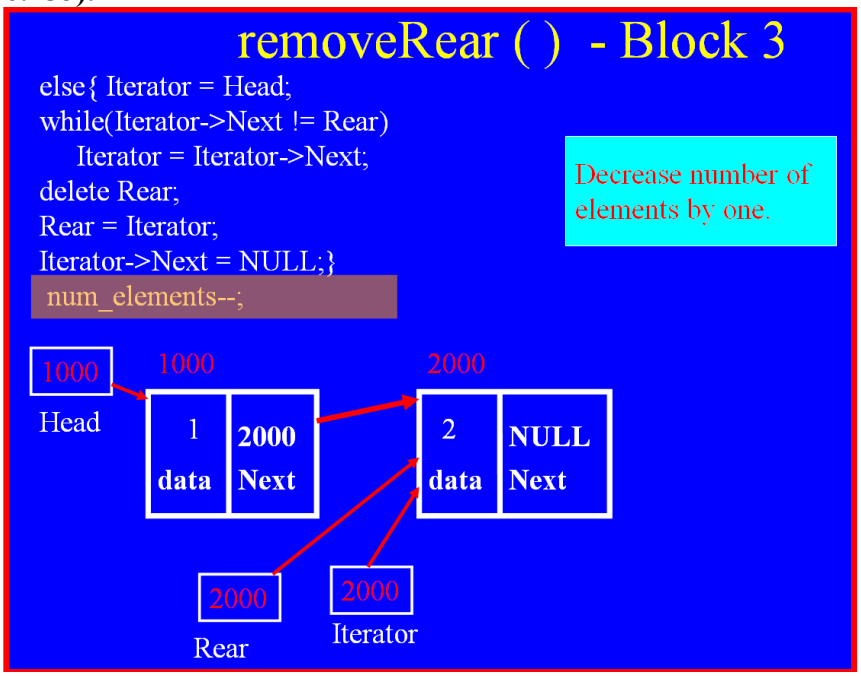


FIG. 6.15J

The function removeFront ( ) works exactly the way function removeRear ( ) does.

Function addAfter (Type obj, Type target)

The function addAfter ( ) adds the obj, after the item called target in the list. The rules of adding using addAfter function are as follows:

- **Add object obj after the first occurrence of object target in the list (scanning from Head pointer towards Rear pointer).**
- **Do nothing if there is no node in list that has same content as target node.**
- **Do nothing if obj is already in the list.**

The following PowerPoint presentation illustrates the dynamics of function addAfter ( ).

C:\Folderes from D\
_CoursesTaught\CS2

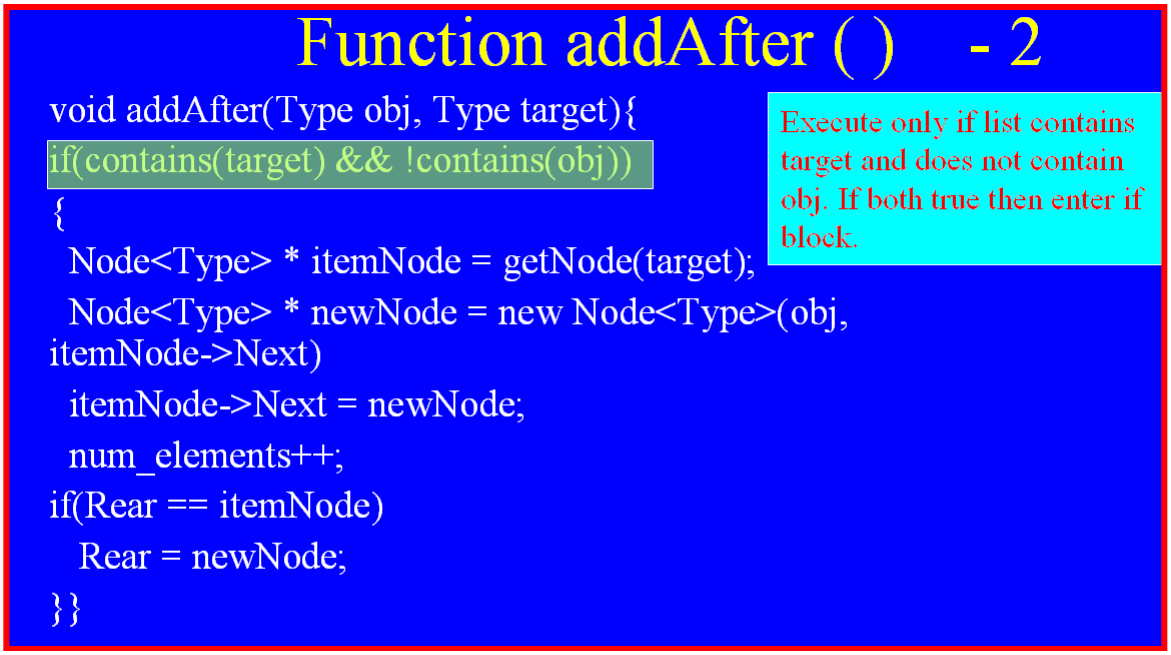


FIG. 6.16A

Function first checks to see that list contains the target and does not contain the obj. If either condition is false then nothing is done and obj is not added (Figure 6.16A). If block is entered then. getNode function is called to get the a pointer to target node that exists in the list (Figure 6.16B).

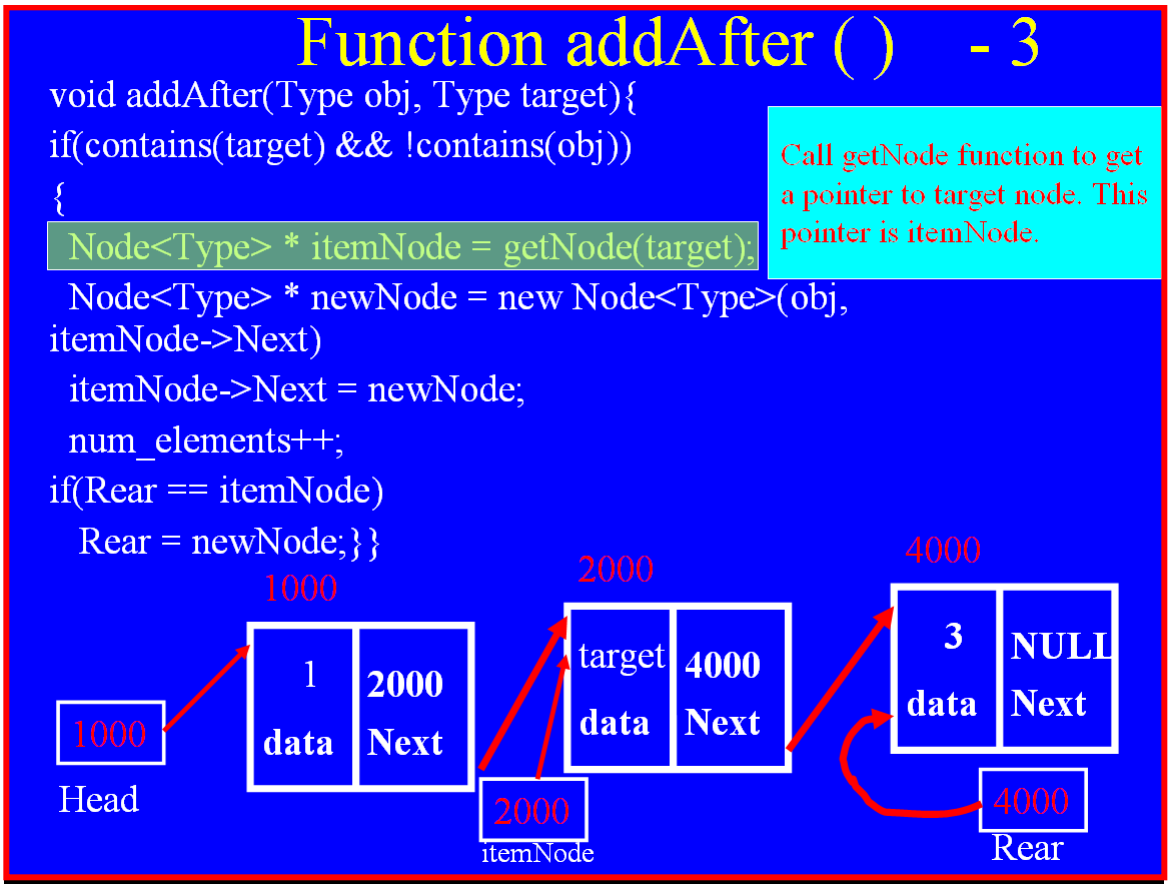


FIG. 6.16B

itemNode is the pointer to target node. In Figure 6.16B the itemNode stores address 2000, pointing to target node. Then we create a newNode that would be added after the target node (Figure 6.16C).

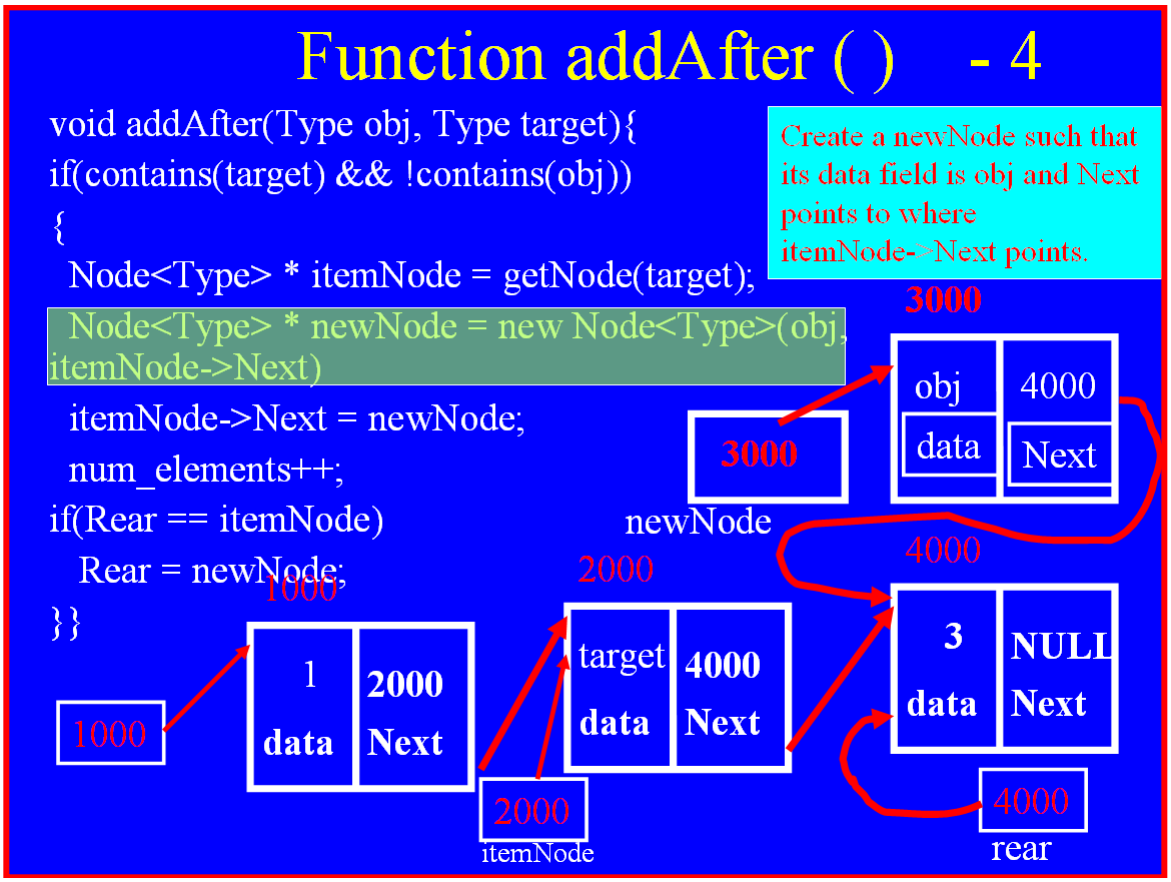


FIG. 6.16C

Clearly understand the constructor call made in highlighted portion of code in Figure 6.16C. newNode is a pointer to a freshly created node whose data field is obj, and Next field is pointing to the pointee of pointer itemNode->Next. This attaches the newNode to point to the node that is currently after the target node. Now we just need to detach the target node from its following node and attach it to newNode (Figure 6.16D).

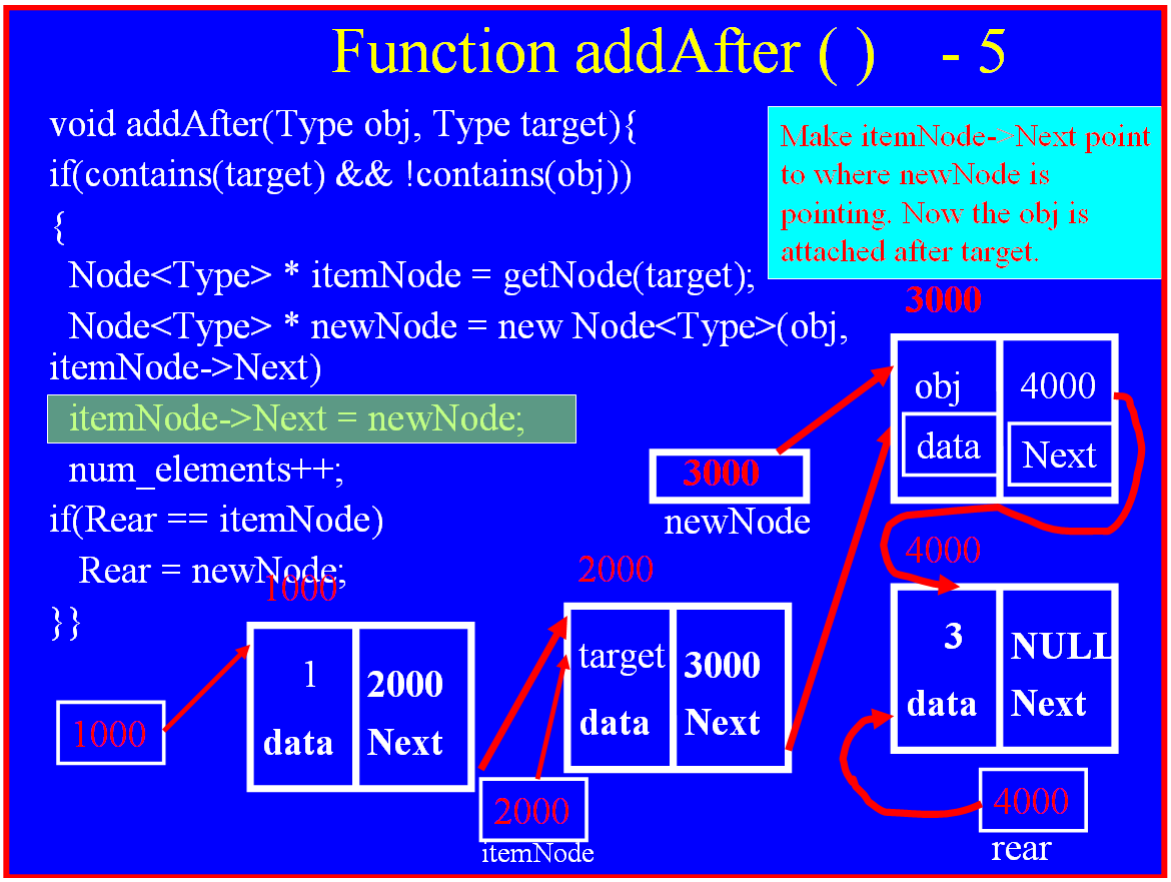


FIG. 6.16D

This is done simply by setting pointer itemNode->Next to point to the pointee of newNode. Thus itemNode->Next now stores address 3000 and task of attaching the node with data value obj after the target node in the list is completed. Next task is to increase the num_elements or number of elements in the list by one.

There is a situation where target node may actually be the last node in the list. In this case the picture of list would look similar to the Figure 6.16E.

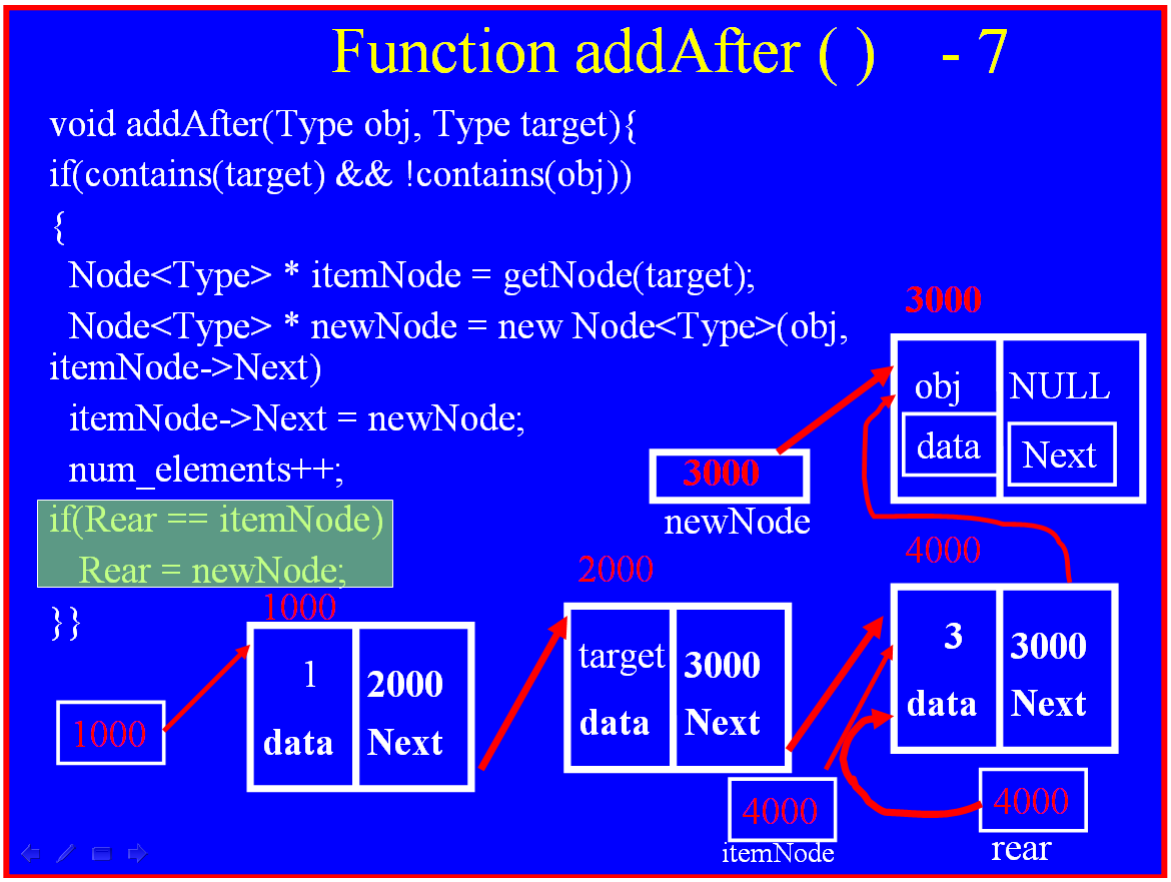


FIG. 6.16E

In this case Rear is pointing to pointee of itemNode. Thus to correct the location of Rear pointer, the Rear is made to point to the pointee of newNode which is the fresh last node in the list (Figure 6.16F).

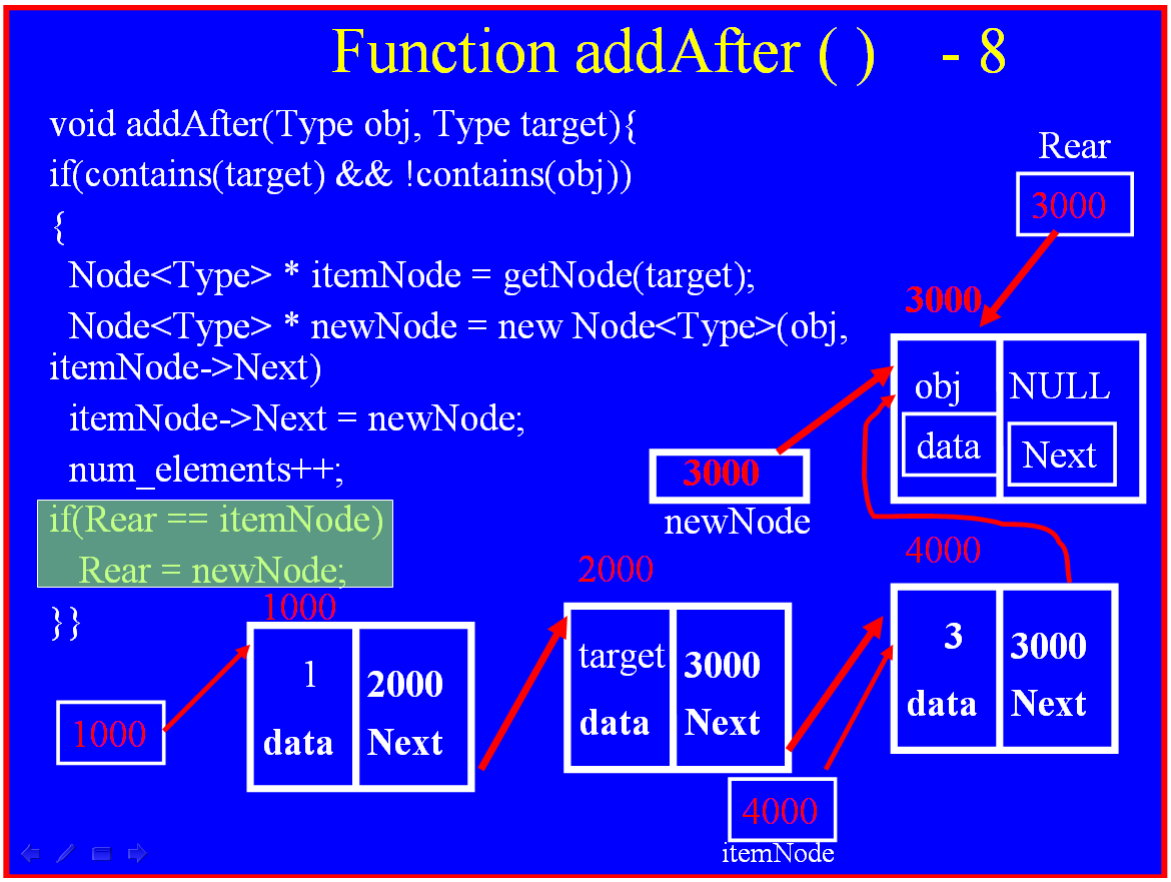


FIG. 6.16F

Function addBefore (Type obj, Type target)

The function addBefore ( ) adds the item obj, before the node with target. Similar to function addAfter ( ) the purpose of function is as follows:

- **Add object obj before the first occurrence of item target in the list (scanning from Head pointer towards Rear pointer).**
- **Do nothing if there is no node in list that has same content as target node.**
- **Do nothing if obj is already in the list.**

The PowerPoint presentation below shows how the function addBefore ( ) works.

C:\Folderes from D\
_CoursesTaught\CS2

We first take the case that target node is first node in the list (Figure 6.17A).

```
void addBefore(Type obj, Type target){
if(contains(target) && !contains(obj)){
  Node<Type> * itemNode = getNode(target);
  Node<Type> * newNode = new
Node<Type>(obj, itemNode);
  if(Head == itemNode)
    Head = newNode;
  else{
    Node<Type> * beforeNode =
nodeBefore(itemNode);
    beforeNode->Next = newNode;}
num_elements++;}}
```

We first take the case where target node is the first node in list. The obj is not in list so if block executes.

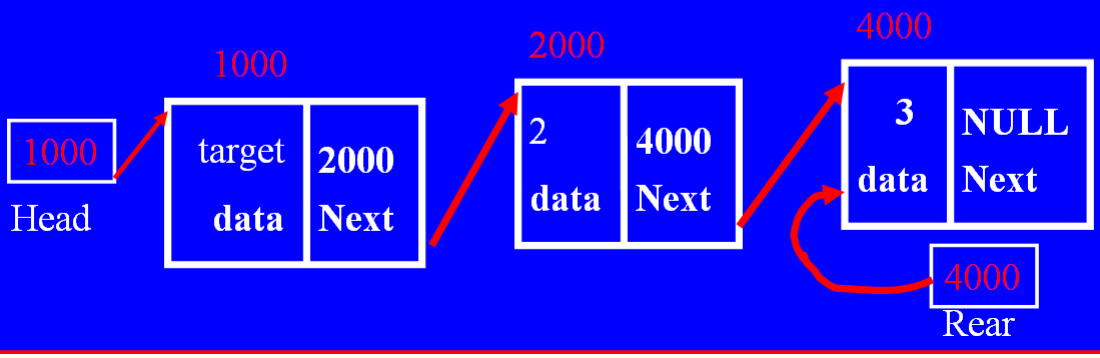


FIG. 6.17A

Since obj is not in list the if block is executed. Then we get the pointer itemNode which points to target node (Figure 6.17B). This is facilitated by using function getNode.

```
void addBefore(Type obj, Type target){
if(contains(target) && !contains(obj)){
 Node<Type> * itemNode = getNode(target);
  Node<Type> * newNode = new
Node<Type>(obj, itemNode);
  if(Head == itemNode)
     Head = newNode;
  else{
    Node<Type> * beforeNode =
nodeBefore(itemNode);
    beforeNode->Next = newNode;}
num_elements++;}}
```

We get the pointer itemNode that points to node with target value.

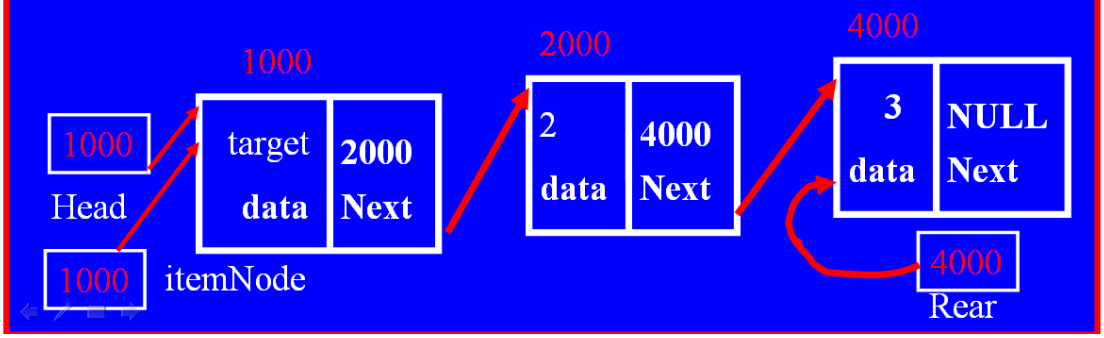


FIG. 6.17B

We create a newNode whose data field is obj and Next field points to target (Figure 6.17C).

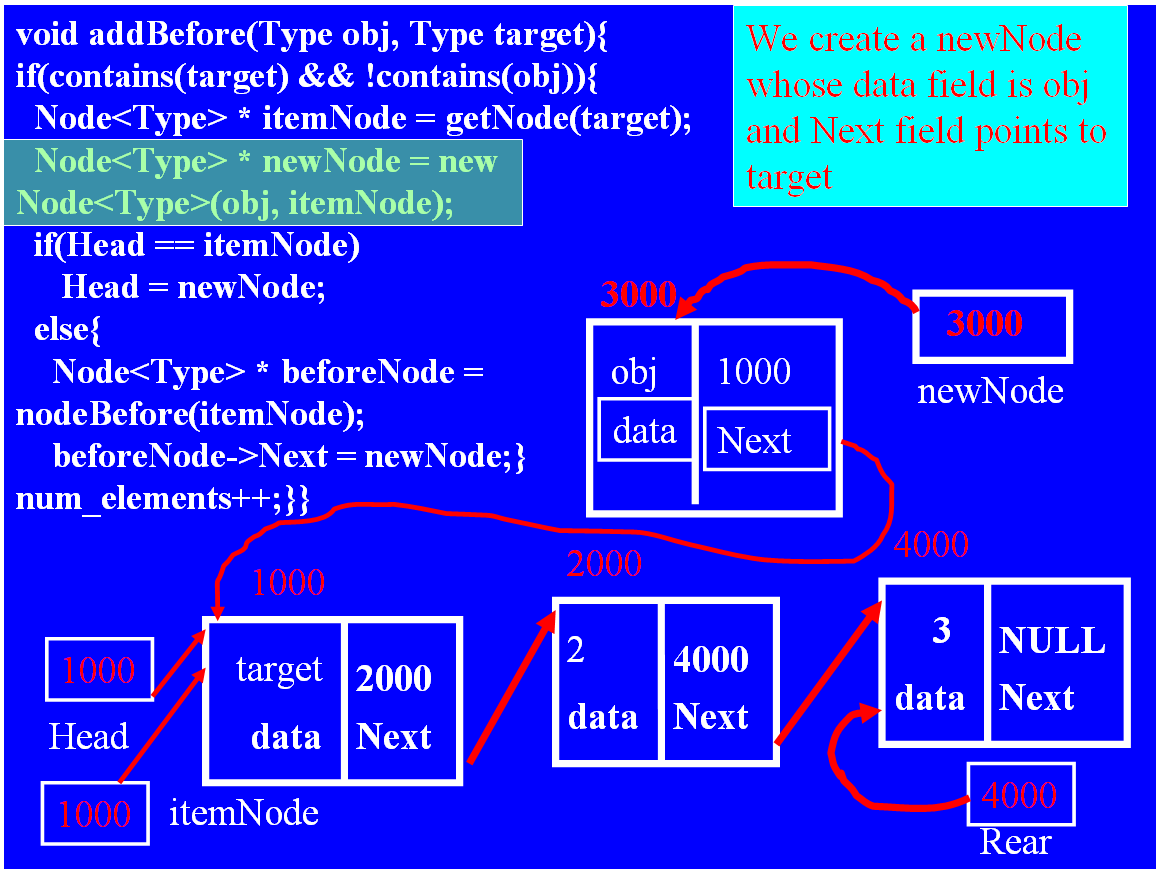


FIG. 6.17C

Since this newNode must be the head node in the list, we simply make Head pointer point to where the newNode is pointing (Figure 6.17D).

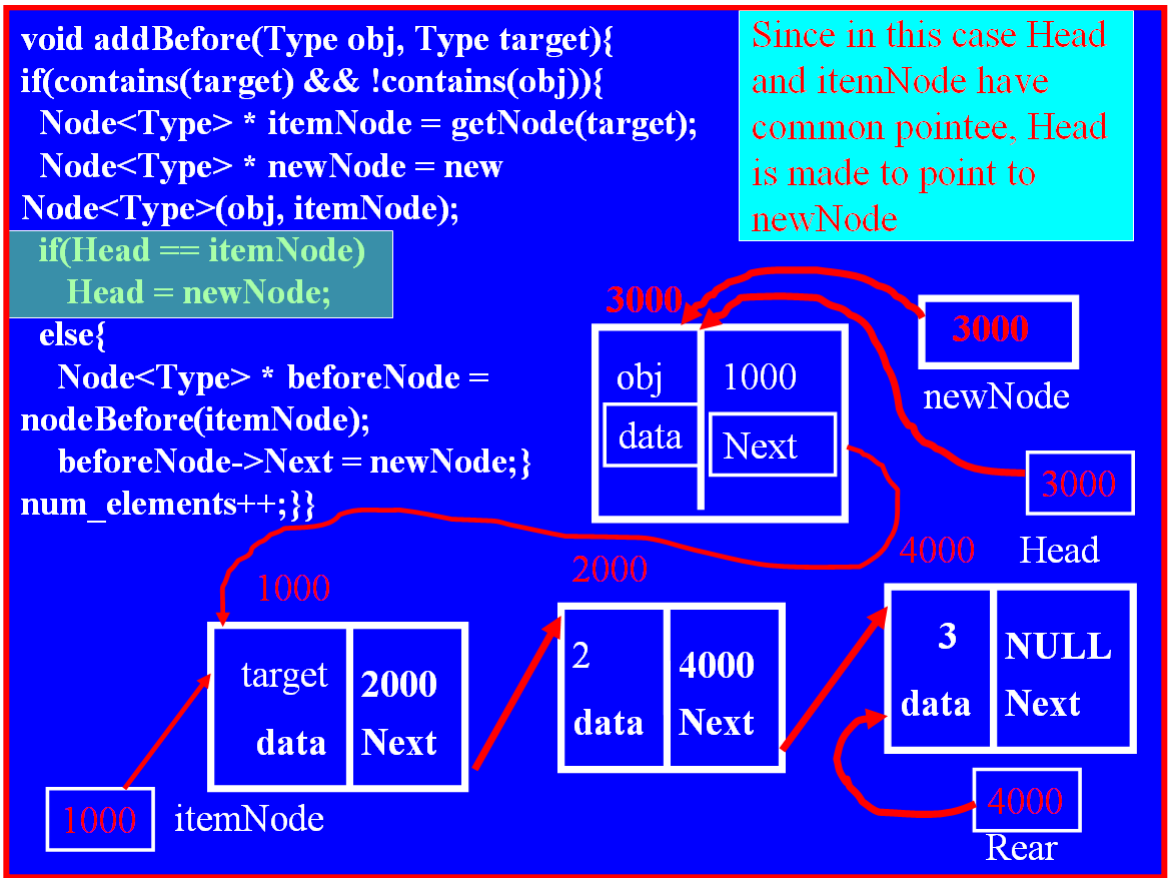


FIG. 6.17D

This completes the first case where the target node was the first node in the list and obj was to be made the head node. Next we consider the case where target node is any node other than the head node in the list (Figure 6.17E). In this case else block is executed.

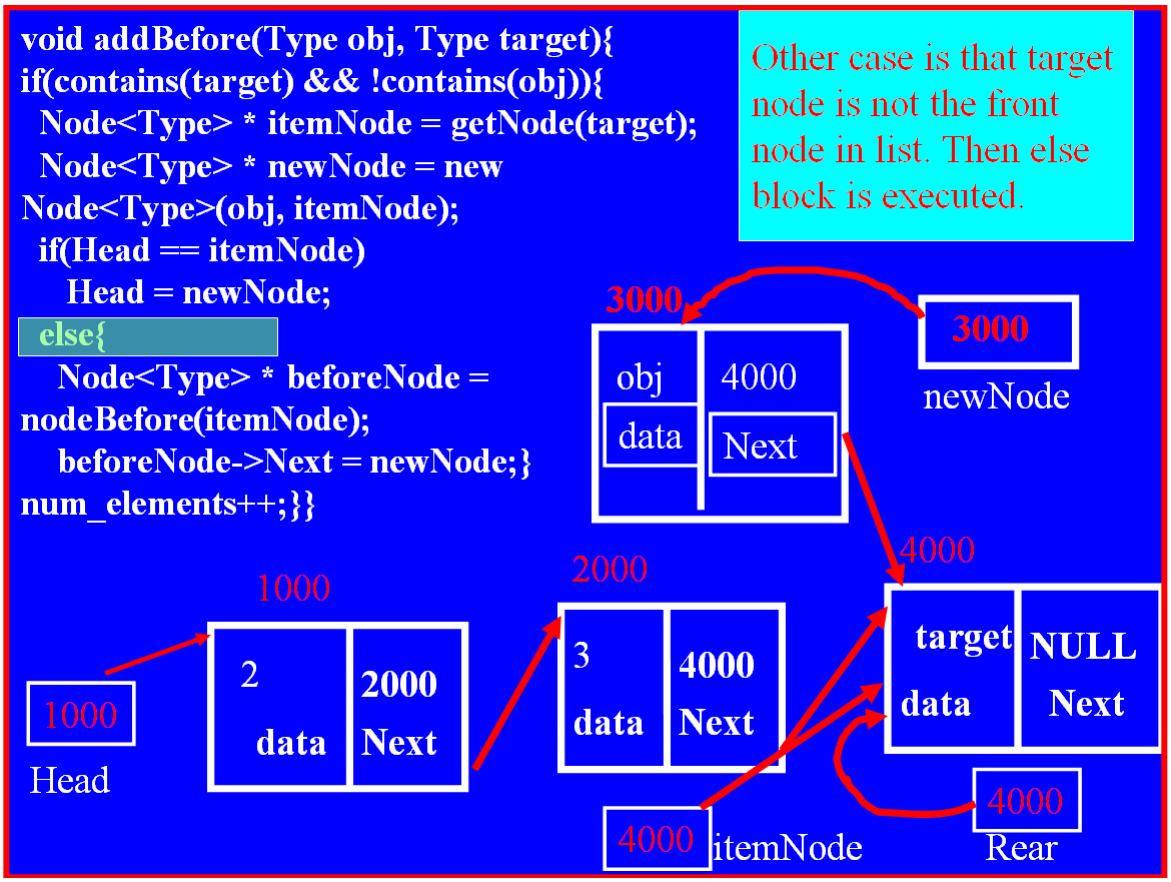


FIG. 6.17E

In this case we use the function beforeNode to get a pointer to the node before the target node (Figure 6.17F).

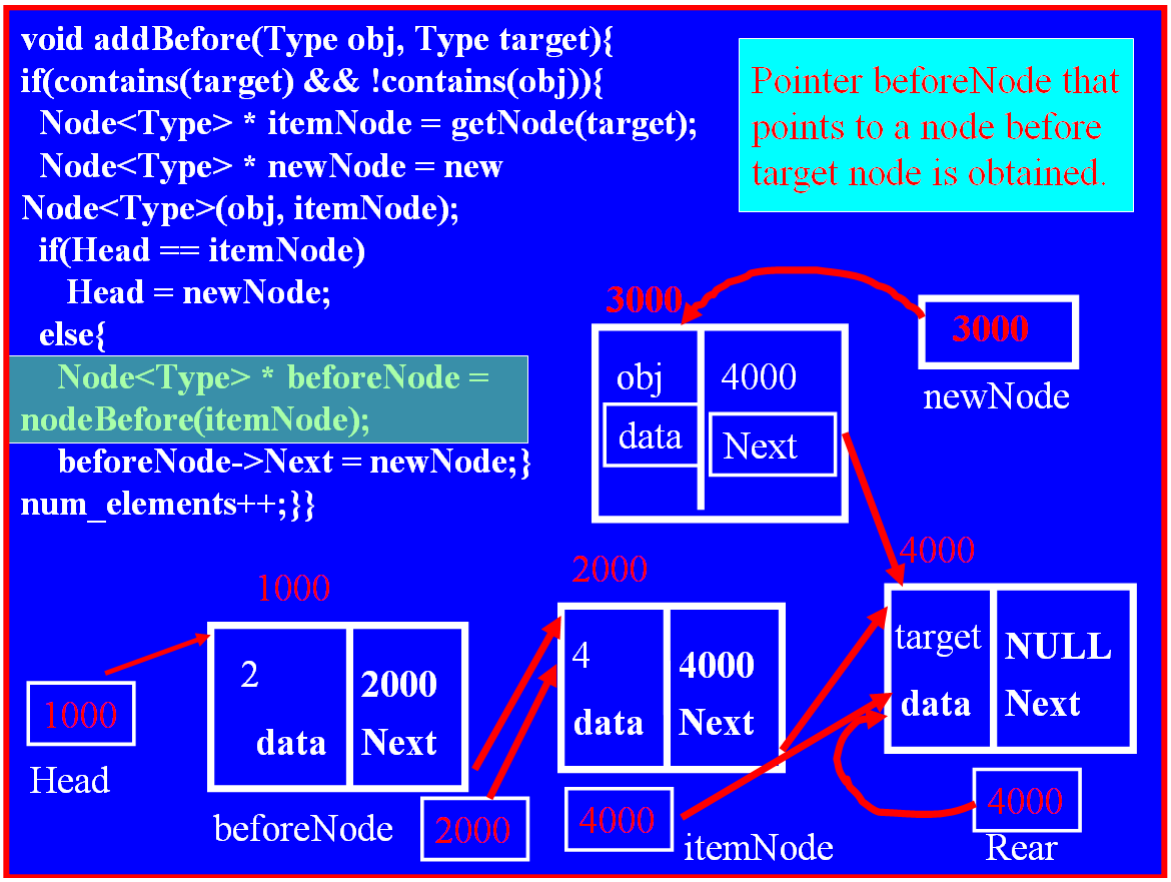


FIG. 6.17F

To attach the newNode to its proper place in the list, all we have to do is to make beforeNode->Next point to where pointer newNode is pointing (Figure 6.17G). This breaks the linkage of beforeNode from target node and establishes a linkage with newNode.

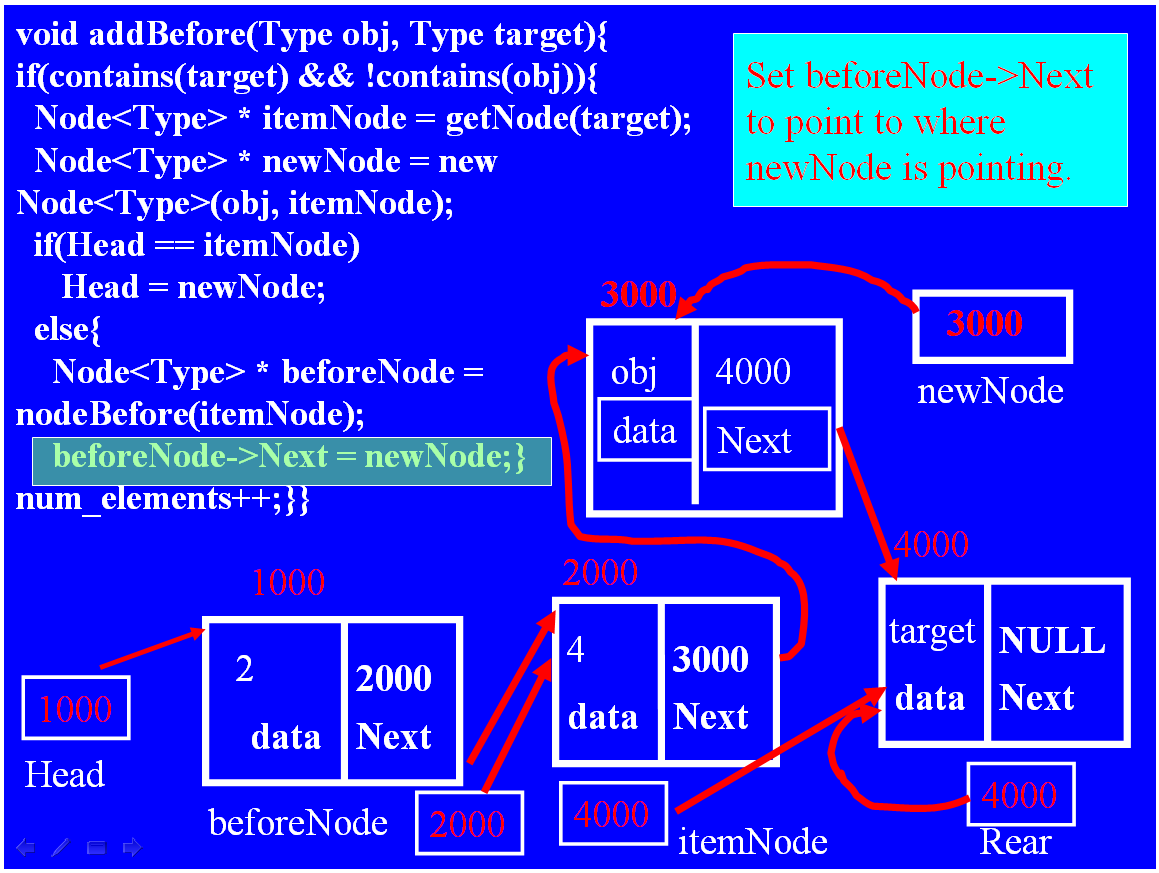


FIG 6.17G

Finally, the number of elements in the list is incremented by one and function execution completes.

Other SinglyLinkedList Functions

The logic of function destroy is similar to the logic of destroyList function discussed in Listing 6.1. The logic of all other functions that are not discussed here is easy enough to understand by reading the source code. The details of unimplemented functions which are required to be completed as laboratory exercise are provided in a separate manual.

Testing of Singly Linked List class

The listing to test SinglyLinkedList build the list, removes elements from list, destroys the list etc. One set of test code is shown in Listing 6.4 below:

```
#include "LinkedListInterface.h"

void testIntegers();
void testStrings();
int main()
{
```

```
      {
            testIntegers();
            testStrings();
      }
      cout<<"The value of memory counter = "<<counter<<endl;
      return 0;
}
////////////////////////////////////////////////////
SinglyLinkedList<string> List;
      List.addFront("john");
      List.addFront("Mary");
      List.addFront("Rob");
      List.addFront("Johnson");
      List.addFront("Tyron");
      List.addFront("Nancy");
      List.addFront("Bill");
      List.addFront("Betty");
      cout<<List;
      cout<<"List size = "<<List.size()<<endl;
      List.addRear("Monica");
      List.addRear("Hamlet");
      List.addRear("Portia");
      List.addRear("Cordelia");
      List.addRear("Goneril");
      List.addRear("Electra");
      List.addRear("Zeus");
      List.addRear("Hera");
      cout<<"Printing linked list front to back.\n";
      cout<<List;
      cout<<"Printing linked list back to front.\n";
      List.printBackward(cout);
      cout<<"Printing linked list front to back, using recursion.\n";
      List.printForwardRecursively(cout);
      cout<<"List size = "<<List.size()<<endl;
      cout<<List.getFront()<<endl;
      cout<<List.getRear()<<endl;
      List.destroy();
      cout<<"List Size = "<<List.size()<<endl;

}



////////////////////////////////////////////////////////////
```

```
void testIntegers()
{
          SinglyLinkedList<int> List;
     cout<<List;
     List.addFront(7);
     List.addFront(6);
     List.addFront(5);
     List.addFront(4);
     List.addFront(3);
     List.addFront(2);
     List.addFront(1);
     List.addFront(0);
     cout<<List;
     cout<<"List size = "<<List.size()<<endl;
     List.addRear(8);
     List.addRear(9);
     List.addRear(10);
     List.addRear(11);
     List.addRear(13);
     List.addRear(12);
     List.addRear(14);
     List.addRear(15);
     cout<<"Printing linked list front to back.\n";
     cout<<List;
     cout<<"Printing linked list back to front.\n";
     List.printBackward(cout);
     cout<<"Printing linked list front to back, using recursion.\n";
     List.printForwardRecursively(cout);
     cout<<"List size = "<<List.size()<<endl;
     cout<<List.getFront()<<endl;
     cout<<List.getRear()<<endl;
     List.destroy();
     cout<<"List Size = "<<List.size()<<endl;
}
//Listing 6.4
```

Figure 6.18 shows the output from Listing 6.4.

FIG. 6.18

The structure of test functions testStrings and testIntegers is identical. In both functions, first the items are added to the linked list using the addFront function and list is then printed. Inside function testIntegers addFront function adds integers 7, 6, 5, 4, 3, 2, 1, and 0 to the list. Because of use of addFront the 7 becomes the last node and when we print the list in forward order the printed list is 0 1 2 3 4 5 6 7 (first line of output in Figure 6.18). List size is then printed (second output line in Figure 6.18). Then numbers 8 to 15 are added to the list using addRear function. This allows build up of list in ascending order. List is then printed in ascending and descending orders using functions: overloaded insertion operator, printBackward, and printForwardRecursively. Functions getFront and getRear are then tested and finally list is destroyed by an explicit call to function destroy. Similar test is repeated with function testStrings. Finally the value of counter zero indicates that program has no memory leaks.

Summary Of Some Facts

1. **Linked Lists are dynamic structures, which are better suited for frequent additions and removals of list items.**

1. **Simplest form of linked list would have a head node. Each node in the list stores a pointer to its next member. The tail node stores a null pointer.**
2. **A singly linked list class has its node pointing only to the node after them. This allows the list iteration only by forward movement in the list. This movement restriction comes with an advantage that the number of links to be made or broken when a node is added or removed is kept to minimum.**
3. **In addition to the head node pointing to the first node (called Head), a pointer to the last node (called Rear) may be added to the singly linked list for added functionality.**
4. **Functions can be added to the Singly Linked List class to search the list for a certain key, navigate through the list using an Iterator, adding or removing the nodes before and after a target node – and helper functions to facilitate the above.**
5. **In later algorithmic analysis we would show that for searching for a key, the linked lists can work no faster than a Big O of 1 (O(N)), whereas arrays can work faster than that having a Big O of $\log_2 N$ or O ($\log_2 N$).**

Reference:

1. http://www.cs.hope.edu/~alganim/jvall/index.html

Appendix 6A

```
#pragma once
#include <iostream>
using namespace std;

class Node
{
public:
	void * data_ptr;
	Node * Next;
	int flag;
	/**
	*Default/explicit constructor for class Node
	*/
	Node(void * init_data=NULL, int init_flag = 0, Node * Next1 = NULL )
	{
		data_ptr = init_data;
		Next = Next1;
		flag = init_flag;
	}
	/**
	*Overloaded insertion operator  << for class Node.
	*/
	friend ostream& operator << (ostream& out, const Node& IL)
	{
		if(IL.flag == 1)
		{
			int * iptr = (int*)IL.data_ptr;

			out<<*iptr<< "    ";
		}
		else if(IL.flag == 2)
		{
			float * fptr = (float*)IL.data_ptr;

			out<<*fptr<< "    ";
		}
		return out;
	}
	/**
	*Overloaded extraction operator for class Node.
	*/
	friend istream& operator >>(istream& in, Node& IL)
	{
		if(IL.flag == 1)
		{
```

```
                        int * iptr = (int*)IL.data_ptr;
                        in>>*iptr;
                }
                return in;
        }
        virtual ~Node()
        { }
};
//Listing A6A.1
```

The listing below gives the main function and other classes.

```
#include "Node.h"
#include <string>
using namespace std;

static int count = 0;
class Student
{
private:
        string name;
        float gpa;
public:
        Student(string init_name="",float init_gpa=0.0f):
          name(init_name),gpa(init_gpa){ }
   friend istream & operator >>(istream & in , Student& Std)
        {
                in>>Std.name;
                in>>Std.gpa;
                return in;
        }
        friend ostream& operator<<(ostream& out, const Student& Std)
        {
                out<<Std.name<<"  "<<Std.gpa<<endl;
                return out;
        }
};
void createList(Node * Head);
void printList(Node * Head);
void destroyList(Node * Head);
int main( )
{
        int flag = 0;
        cout<<"Enter 1 to build linked list of Integers:\n"
                <<"Enter 2 to build linked list of floats:\n"
                <<"Enter 3 to build linked list of chars:\n"
                <<"Enter 4 to build linked list of string\n"
```

```
            <<"Enter 5 to build linked list of students:";

        cin>>flag;

        if(flag == 1)
        {
                Node * Head = new Node(0,1,0);
                count++;
                createList(Head);
                printList(Head);
                destroyList(Head);
        }
        else if(flag == 2)
        {
                Node * Head = new Node(0,2,0);
                count++;
                createList(Head);
                printList(Head);
                destroyList(Head);
        }
        else if(flag == 3)
        {
                Node * Head = new Node(0,3,0);
                count++;
                createList(Head);
                printList(Head);
                destroyList(Head);
        }
        else if(flag == 4)
        {
                Node * Head = new Node(0,4,0);
                count++;
                createList(Head);
                printList(Head);
                destroyList(Head);
        }
        else if(flag == 5)
        {
                Node * Head = new Node(0,5,0);
                count++;
                createList(Head);
                cout<<"Printing first names and gpa of students in linked list:\n";
                cout<<"NAME  "<<"GPA"<<endl;
                printList(Head);
                destroyList(Head);
        }
```

```
        cout<<"The value of memory counter = "<<count<<endl;
        return 0;
}
//////////////////////////////////////////////
void createList(Node * Head)
{
        bool done = false;
        if(Head->flag == 1)
        {
                //List to be built is int list
                Node * IHead = Head;

                while(!done)
                {
                        int temp = 0;
                        cout<<"Enter an integer to add to linked list: ";
                        cin>>temp;
                Node * TNode = new Node((void*)new int(temp),1,IHead->Next);
                        count++;
                        count++;
                        IHead->Next = TNode;
                        cout<<"More data? Enter 0 to continue, enter 1 to exit: ";
                        cin>>done;
                }
        }
        else if(Head->flag == 2)
        {
                //List to be built is float list
                Node * IHead = Head;

                while(!done)
                {
                        float temp = 0;
                        cout<<"Enter a float to add to linked list: ";
                        cin>>temp;
                Node * TNode = new Node((void*)new float(temp),1,IHead->Next);
                        count++;
                        count++;
                        IHead->Next = TNode;
                        cout<<"More data? Enter 0 to continue, enter 1 to exit: ";
                        cin>>done;
                }
        }
```

```
else if(Head->flag == 3)
{
        //List to be built is character list
        Node * IHead = Head;

        while(!done)
        {
                char temp = ' ';
                cout<<"Enter a char to add to linked list: ";
                cin>>temp;
        Node * TNode = new Node((void*)new char(temp),1,IHead->Next);
                count++;
                count++;
                IHead->Next = TNode;
                cout<<"More data? Enter 0 to continue, enter 1 to exit: ";
                cin>>done;
        }
}
else if(Head->flag == 4)
{
        //List to be built is string list
        Node * IHead = Head;

        while(!done)
        {
                string temp = "";
                cout<<"Enter a string to add to linked list: ";
                cin>>temp;
        Node * TNode = new Node((void*)new string(temp),1,IHead->Next);
                count++;
                count++;
                IHead->Next = TNode;
                cout<<"More data? Enter 0 to continue, enter 1 to exit: ";
                cin>>done;
        }
}
else if(Head->flag == 5)
{
        //List to be built is Student list
        Node * IHead = Head;

        while(!done)
        {
                string temp = "";
                float temp_gpa = 0.0f;
        cout<<"Enter only first name of student to be added to linked list: ";
```

```
                        cin>>temp;
                        cout<<"Enter a gpa of student to be added to linked list: ";
                        cin>>temp_gpa;
Node * TNode = new Node((void*)new Student(temp,temp_gpa),1,IHead->Next);
                        count++;
                        count++;
                        IHead->Next = TNode;
                        cout<<"More data? Enter 0 to continue, enter 1 to exit: ";
                        cin>>done;
                }
        }
}
//////////////////////////////////////////////
void printList(Node * Head)
{
        Node * Iter;
        if(Head->flag == 1)
        {
                Node * IHead = Head;
                Iter = IHead->Next;

                while(Iter != NULL)
                {
                        cout<< *((int*)Iter->data_ptr)<<"  ";
                        Iter = Iter->Next;
                }
        }
        else if(Head->flag == 2)
        {
                Node * IHead = Head;
                Iter = IHead->Next;

                while(Iter != NULL)
                {
                        cout<< *((float*)Iter->data_ptr)<<"  ";
                        Iter = Iter->Next;
                }
        }
        else if(Head->flag == 3)
        {
                Node * IHead = Head;
                Iter = IHead->Next;

                while(Iter != NULL)
                {
                        cout<< *((char*)Iter->data_ptr)<<"  ";
```

```
                        Iter = Iter->Next;
                }
        }
        else if(Head->flag == 4)
        {
                Node * IHead = Head;
                Iter = IHead->Next;

                while(Iter != NULL)
                {
                        cout<< *((string*)Iter->data_ptr)<<"  ";
                        Iter = Iter->Next;
                }
        }
        else if(Head->flag == 5)
        {
                Node * IHead = Head;
                Iter = IHead->Next;

                while(Iter != NULL)
                {
                        cout<< *((Student*)Iter->data_ptr)<<"  ";
                        Iter = Iter->Next;
                }
        }



        cout<<endl;
}
//////////////////////////////////////////////////
void destroyList(Node * Head)
{
        Node * Iter;
        Node * IHead = Head;
        Iter = IHead;
        int dummy = 0;

        while(Iter != NULL)
        {
                IHead = IHead->Next;
                if(dummy != 0)
                {
                        delete Iter->data_ptr;
                        count--;
                }
```

```
                delete Iter;
                dummy++;
                count--;
                Iter = IHead;
        }

}
//////////////////////////////////////////////
```