LISTS
(USING LINKED LISTS)

iACADEMY
SCHOOL OF COMPUTING • SCHOOL OF BUSINESS • SCHOOL OF DESIGN

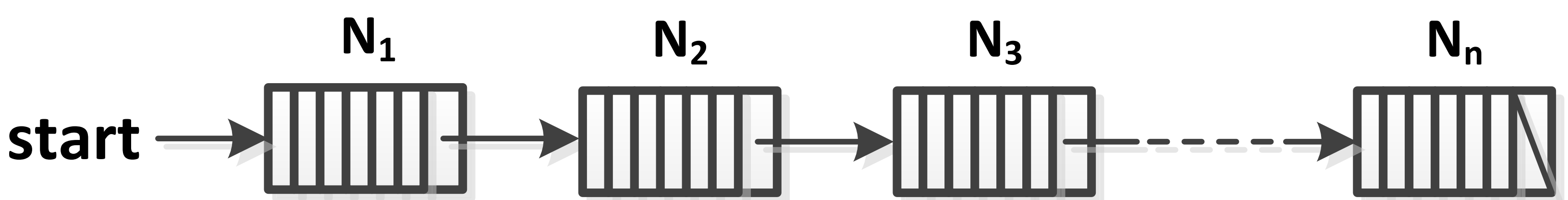
SCHOOL OF COMPUTING
BENNETT TANYAG

LINKED LIST IMPLEMENTATION OF LISTS

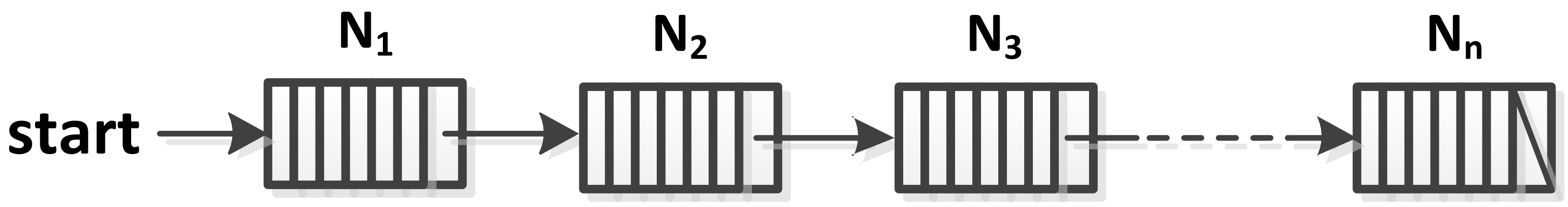
- A ***singly linked list***, or simply ***linked list***, consists of ordered ***nodes*** wherein each node represents an item in the list:

$$N_1, N_2, \ldots, N_n$$

Each linked list has a pointer that points to the first node in the list (node N_1).



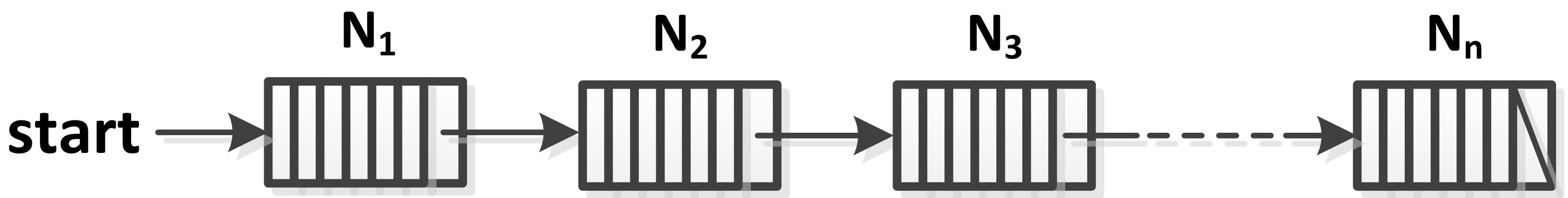
LINKED LIST IMPLEMENTATION OF LISTS



Each node is a structure with several fields (for the data of each item in the list).

The last field of a node will be a pointer that points to the next or succeeding node in the list. This is how sequencing in a list is achieved. This field is often called the ***address field***. The address field of node N_i contains the address of the following node in the list N_{i+1}.

LINKED LIST IMPLEMENTATION OF LISTS

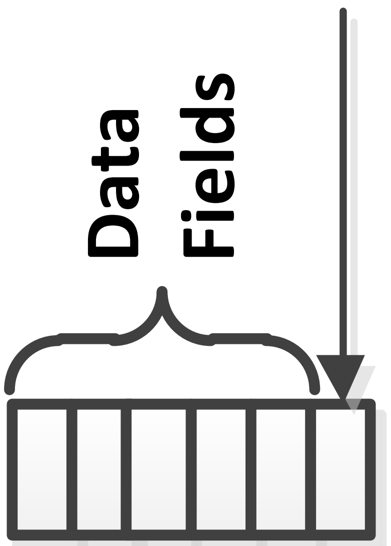


In other words, the address field of N_i points to N_{i+1}. For example, the address field of node N_2 should point to node N_3.

The address field of the last node, N_n, contains a special value called *NULL* which means that the address field of the last node does not point to anything.

A pointer (in this example, *start*) will point to the first element or item in the list. This allows the program to easily locate the first node.

LINKED LIST IMPLEMENTATION OF LISTS



Sample Definition of a Node:

```
struct NODE
{
    char name[50];
    char address[100];
    int age;
    float weight;
    float height;
    NODE *next;
};
```

LINKED LIST IMPLEMENTATION OF LISTS

- Example (List of Integers)

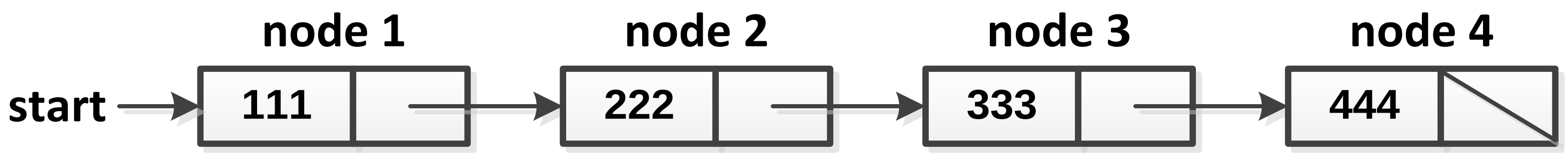
  To simplify discussions, assume that the list is simply a list of integers.

  So the definition of a node is as follows:

```
struct NODE
{
    int value;
    NODE *next;
};
```

LINKED LIST IMPLEMENTATION OF LISTS

- Example (List of Integers):



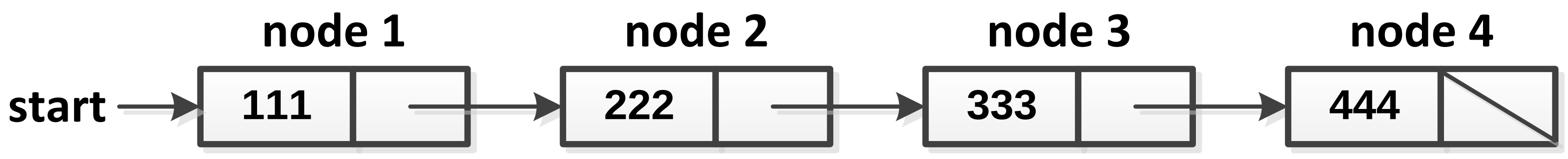
Some important points:

1. This is an example of a linked list consisting of 4 nodes representing a list of integers.

2. The pointer variable *start* contains the address of the first node (*start* points to node 1) to indicate the first integer in the list.

LINKED LIST IMPLEMENTATION OF LISTS

- Example (List of Integers):

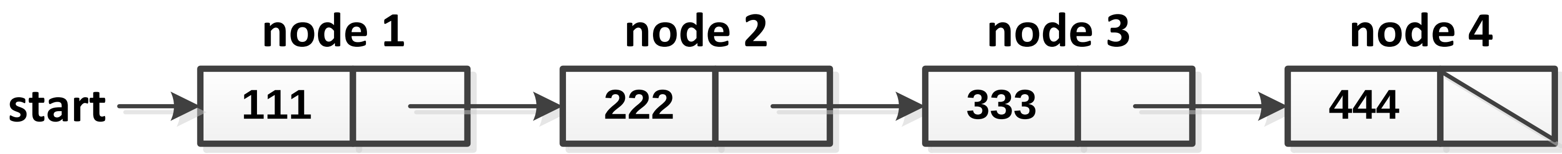


3. Each node has two fields. The first field contains the value of the integer.

4. The second field is the address field. As discussed earlier, the address field of a particular node contains the address of the node after it. In other words, the address field of a particular node points to the next node in the list.

LINKED LIST IMPLEMENTATION OF LISTS

- Example (List of Integers):



5. The address field of the last node has the value *NULL* to indicate that it is not pointing to any other node and that it is the last node of the list.

6. This list represents the sequence: 111, 222, 333, 444

7. An ***empty list*** is a linked list that does not contain any node. Since there is no first node, *start* is equal to *NULL* (*start* is not pointing to anything).

LINKED LIST IMPLEMENTATION OF LISTS

- Sample Program to Create the Linked List of Integers:

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

struct NODE
{
	int value;
	NODE *next;
};
```

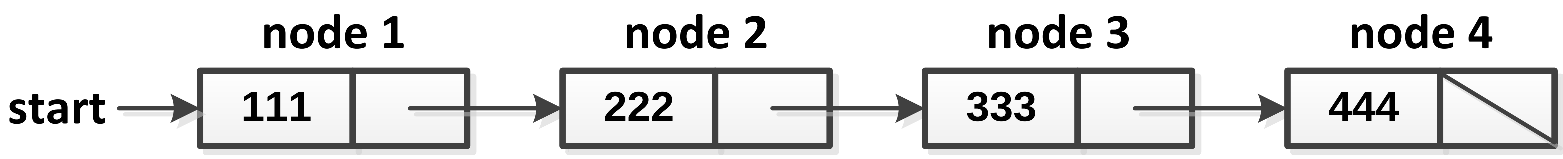
LINKED LIST IMPLEMENTATION OF LISTS

```
main()
{
    NODE node1, node2, node3, node4, *start;

    node1.value = 111;
    node2.value = 222;
    node3.value = 333;
    node4.value = 444;

    start = &node1;
    node1.next =&node2;
    node2.next =&node3;
    node3.next =&node4;
    node4.next = NULL;
}
```

This program is not practical since it is static (additional nodes cannot be created while the program is running).

One of the advantages of linked lists is that programs can create additional nodes during run-time (to be discussed later).

TRAVERSING A LINKED LIST

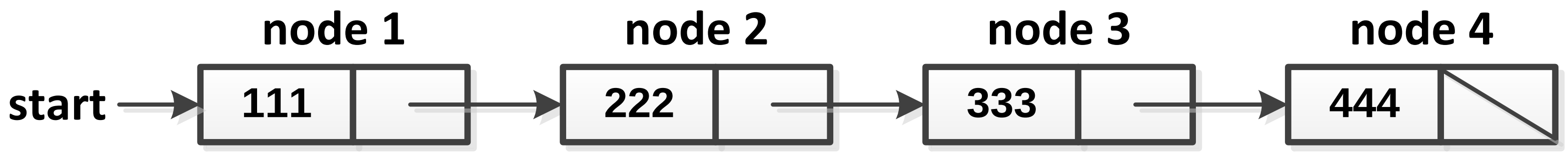


Traversing a linked list means visiting each and every node in the list and perhaps performing a necessary operation (such as printing or displaying the contents of the node).

To traverse a linked list, it is necessary to use a pointer (not *start*) and use it to point at each node in the list one at a time.

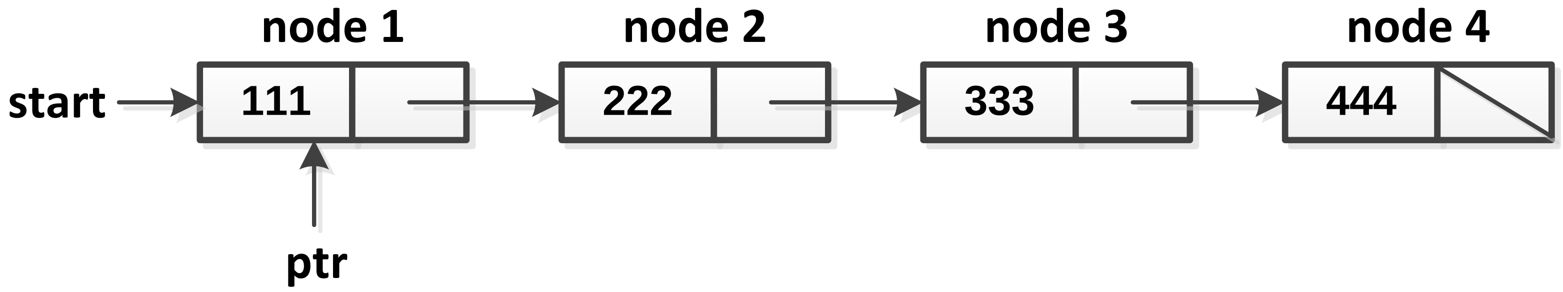
Always remember that the pointer *start* should not be moved (it should always point to the first item in the list).

TRAVERSING A LINKED LIST

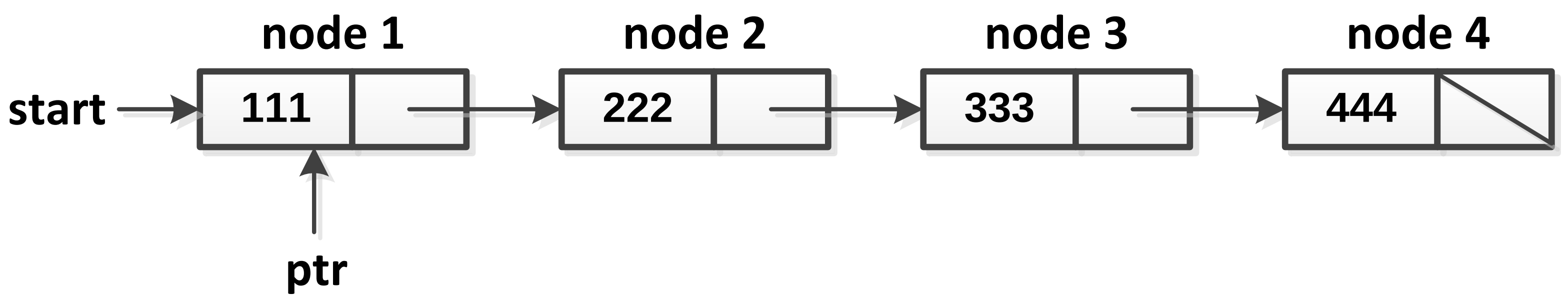


First, let the pointer *ptr* point to the first node (*ptr* should point to the same thing *start* is pointing at). This is done by executing:

ptr = start;



TRAVERSING A LINKED LIST



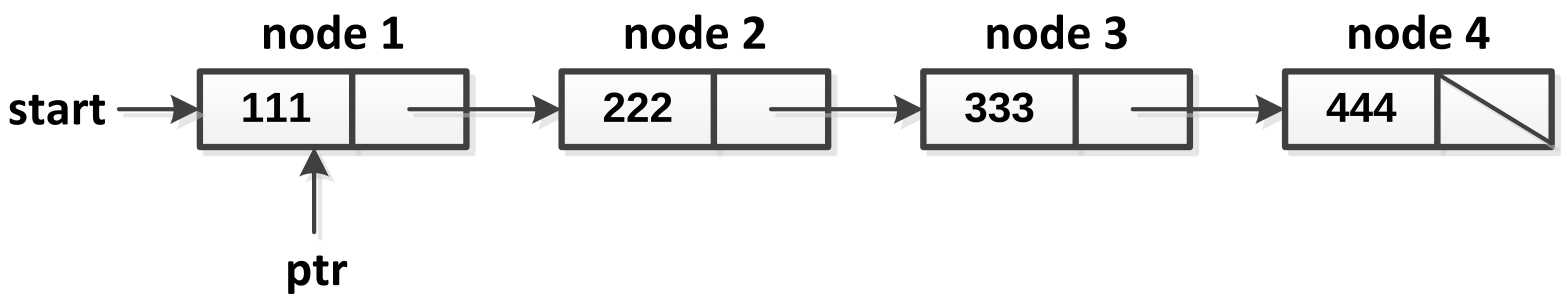
To print the data of that node, execute:

cout << "The value of the node is " << (*ptr).value << ".";

or it may be written as:

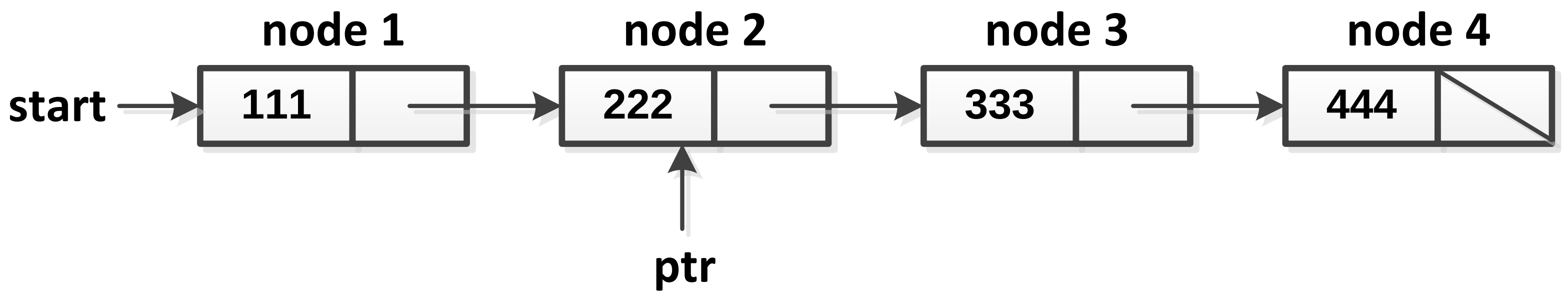
cout << "The value of the node is " << ptr -> value << ".";

TRAVERSING A LINKED LIST

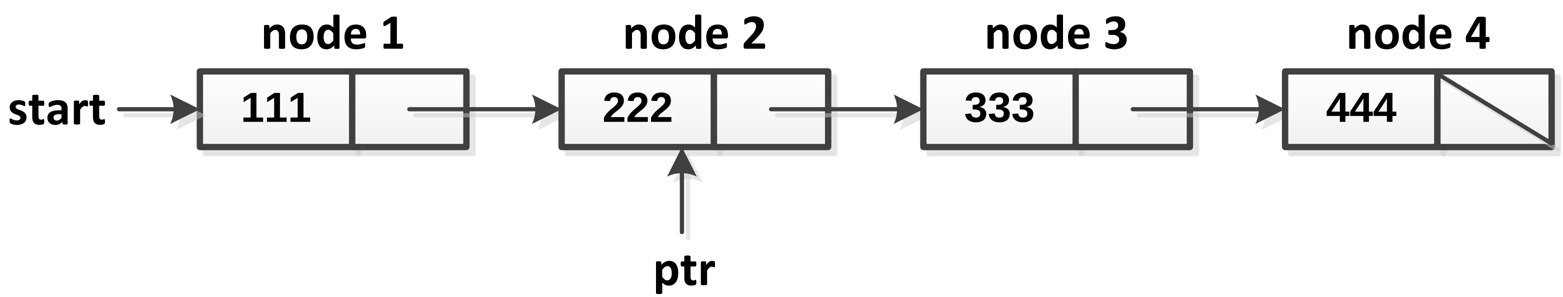


Then let the pointer *ptr* point to the second node in the list. This is done by executing:

ptr = ptr -> next;



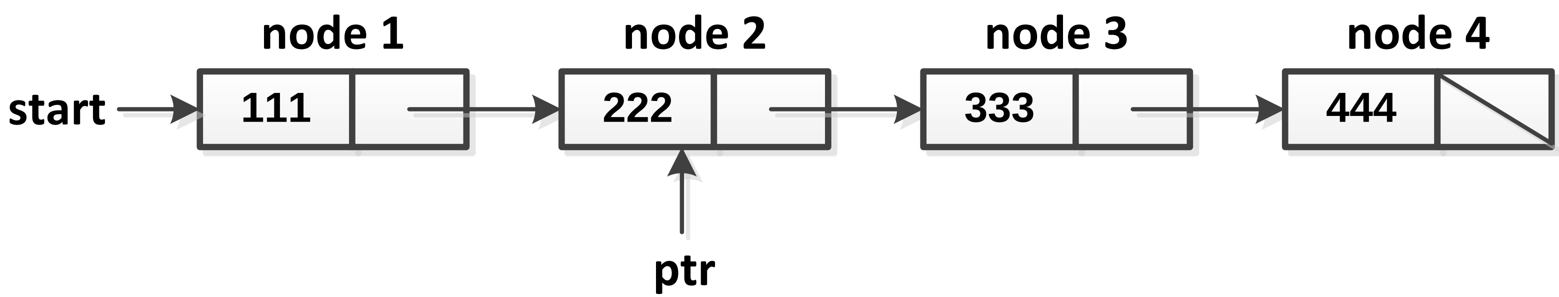
TRAVERSING A LINKED LIST



To print the data of that node, execute again:

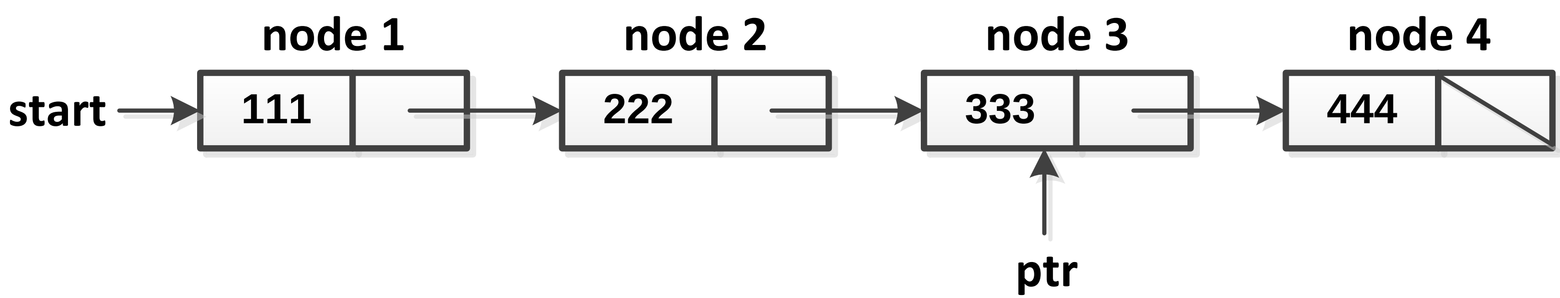
```
cout << "The value of the node is " << ptr -> value << ".";
```

TRAVERSING A LINKED LIST

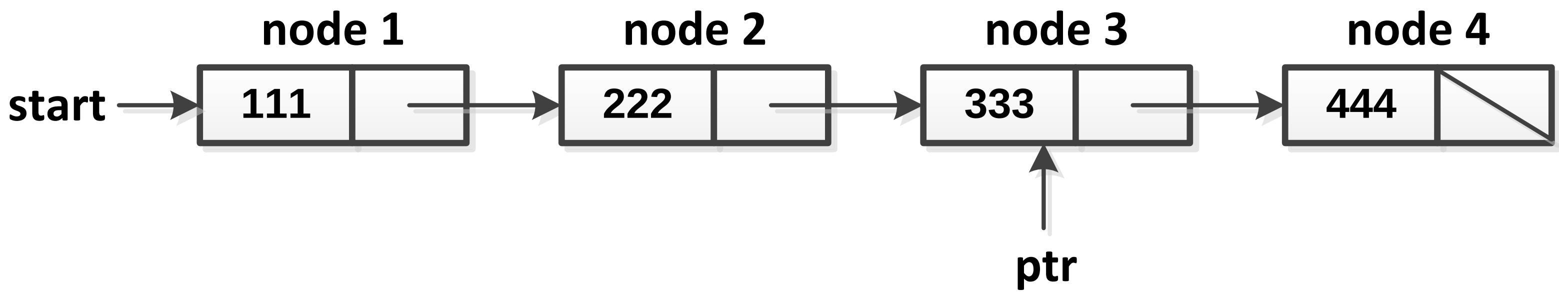


Then let the pointer *ptr* point to the third node in the list. This is done by executing:

ptr = ptr -> next;



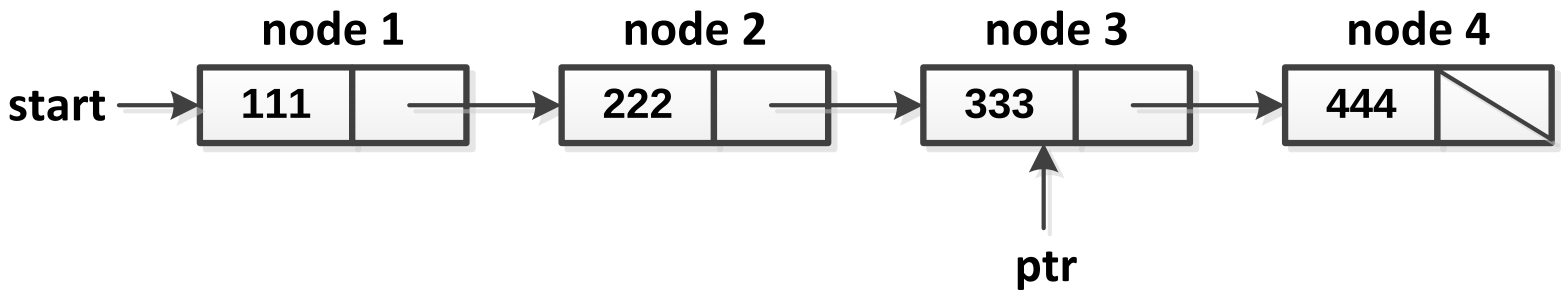
TRAVERSING A LINKED LIST



To print the data of that node, execute again:

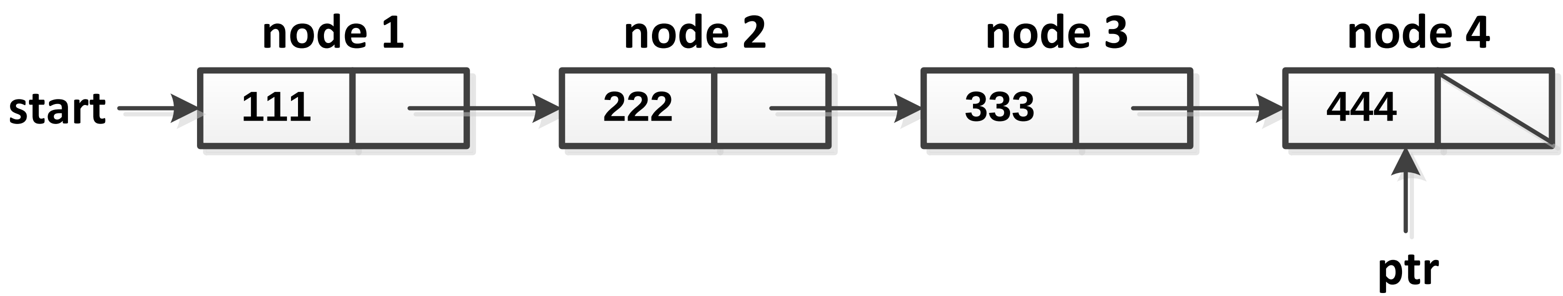
```
cout << "The value of the node is " << ptr -> value << ".";
```

TRAVERSING A LINKED LIST

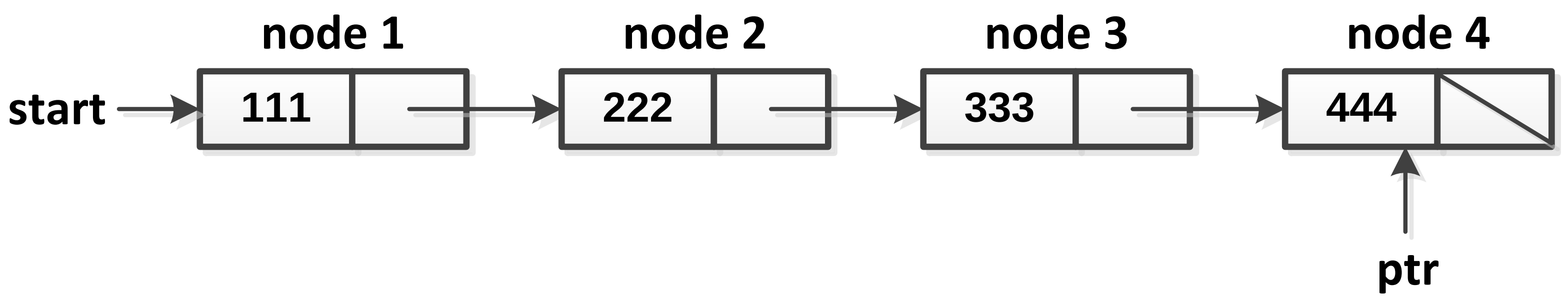


Then let the pointer *ptr* point to the fourth node in the list. This is done by executing:

ptr = ptr -> next;



TRAVERSING A LINKED LIST



To print the data of that node, execute again:

```
cout << "The value of the node is " << ptr -> value << ".";
```

TRAVERSING A LINKED LIST

Summary of the algorithm for traversing the list:

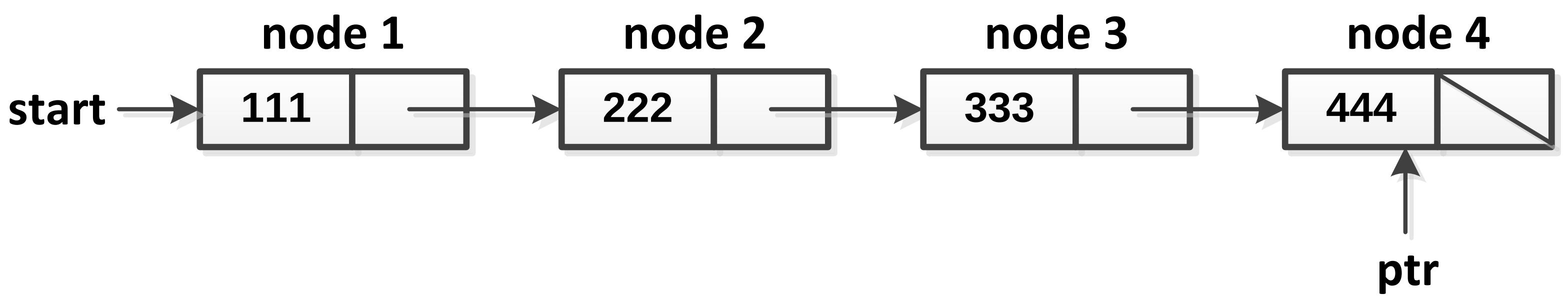
1. Let *ptr* point to the first node.

2. Repeat the following:

```
cout << "\nThe value …
ptr = ptr -> next;
```

until the last node is visited (printed).

TRAVERSING A LINKED LIST



Take note that if the pointer *ptr* is pointing to the last node and the instruction ***ptr = ptr -> next;*** is executed, *ptr* will become equal to *NULL*. This means that there are no more nodes to be traversed so the loop should be terminated.

So while *ptr* is not equal to *NULL*, the loop should continue.

TRAVERSING A LINKED LIST

Program segment for traversing the list:

```
ptr = start;

while (ptr != NULL)
{
    cout << "The value of the node is " <<
                                ptr -> value << ".";
    ptr = ptr -> next;
}
```

TRAVERSING A LINKED LIST

- Sample Program for Linked List (with printing of nodes)

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

struct NODE
{
    int value;
    NODE *next;
};
```

iACADEMY
SCHOOL OF COMPUTING • SCHOOL OF BUSINESS • SCHOOL OF DESIGN

TRAVERSING A LINKED LIST

```
main()
{

    NODE node1, node2, node3, node4, *start;
    int ctr;

    node1.value = 111;
    node2.value = 222;
    node3.value = 333;
    node4.value = 444;

    start = &node1;
    node1.next =&node2;
    node2.next =&node3;
    node3.next =&node4;
    node4.next = NULL;
```

iACADEMY
SCHOOL OF COMPUTING • SCHOOL OF BUSINESS • SCHOOL OF DESIGN

TRAVERSING A LINKED LIST

```
ptr = start;
ctr = 1;

while (ptr != NULL)
{
    cout << "\nValue of node " << ctr << " is "
                                          << ptr -> value <<".";
    ++ctr;
    ptr = ptr->next;
}
}
```

THE malloc() FUNCTION

- When a program declares or defines a variable, C++ will allocate main memory storage to that variable and assigns a name to that storage.

- For example:

```
int x;
```

allocates four bytes of memory storage and the name *x* is assigned or associated to that storage.

THE malloc() FUNCTION

- This kind of storage allocation is called ***compile-time storage allocation*** since the system knows at compile time the type and amount of storage required.

- It is also possible to allocate storage to variables when the program is already running. This kind of storage allocation is called ***run-time storage allocation***.

- The *malloc()* function allocates an amount of storage specified by the programmer and returns the address of the storage allocated. If the function cannot find any available storage in memory, it will return *NULL*.

THE malloc() FUNCTION

- To allocate one *int* cell, first define a pointer to an integer:

  int *p;

  then invoke the *malloc()* function

  p = (int*) malloc(sizeof (int));

  where the argument *sizeof(int)* is the number of bytes necessary to store one *int*. Remember, *sizeof()* is a C++ function that returns the number of bytes needed to store any data type.

  malloc() returns a value which is the address of the storage allocated and is stored into *p*.

THE malloc() FUNCTION

- Sample Program Using Run-time Allocation

```
#include <iostream>
#include <cstdlib>
using namespace std;

main()
{
    int  *p1;
    char *p2;

    p1 = (int*) malloc(sizeof(int));
    p2 = (char*) malloc(sizeof(char));
```

THE malloc() FUNCTION

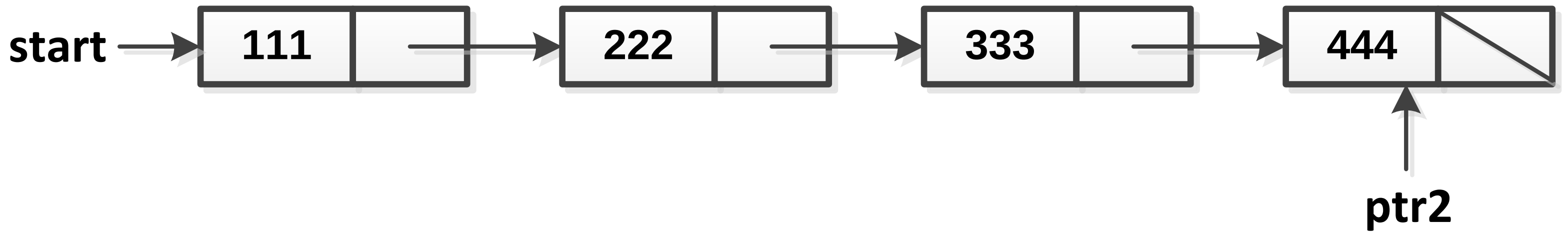
```
    cout << "Enter an integer: ";
    cin >> *p1;

    cout << "Enter a character: ";
    cin >> *p2;

    cout << "\n\nThe integer you entered is " << *p1 << ".";
    cout << "\n\nThe character you entered is " << *p2 << ".";
}
```

iACADEMY
SCHOOL OF COMPUTING • SCHOOL OF BUSINESS • SCHOOL OF DESIGN

CREATING A LINKED LIST USING malloc()

- Assume that there is already a linked list:

- To add a new node at the rear of the list, there must be a pointer pointing to the last node of the list

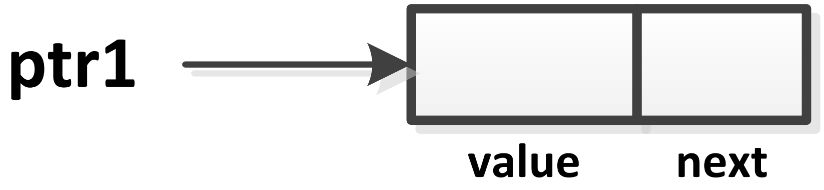


CREATING A LINKED LIST USING malloc()

- To add a new node, first create a new node by executing:

```
ptr1 = (NODE*) malloc(sizeof(NODE));
```

This will create a new node and it will be pointed to by the pointer *ptr1*.



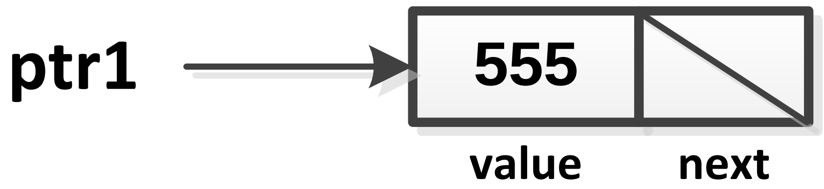
CREATING A LINKED LIST USING malloc()

- The next step is to put values in the fields of the new node. To put data in the *value* field, execute

  ptr1 -> value = 555; (or ask the user to input the data)

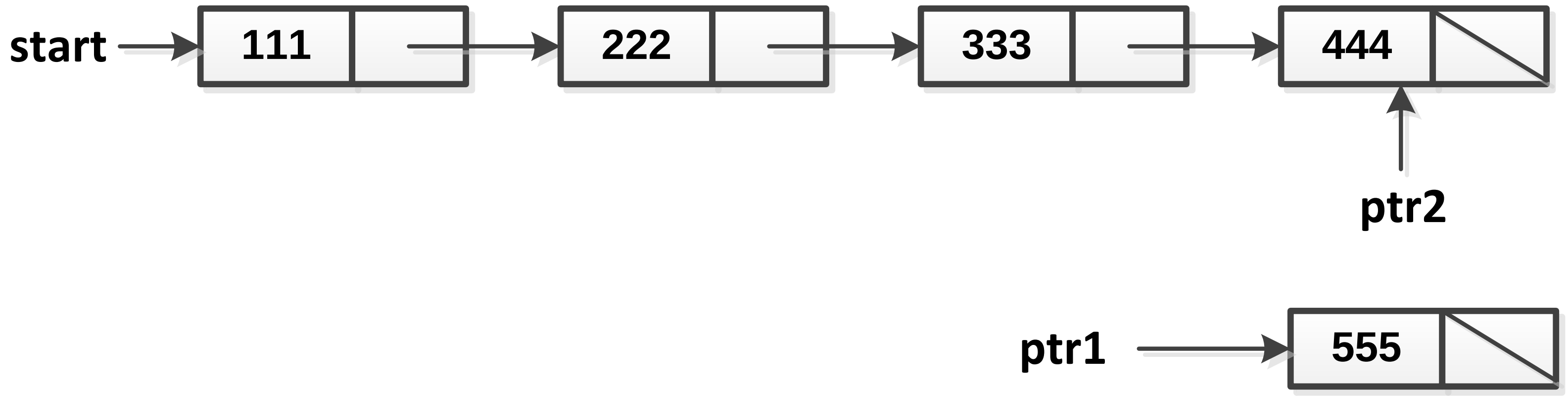
  The *next* field should be made equal to *NULL* since the new node will become the last node in the list. This is done by executing:

  ptr1 -> next = NULL;



CREATING A LINKED LIST USING malloc()

- The last step is to attach the new node to the list:



This is done by executing:

ptr2 -> next = ptr1;
ptr2 = ptr2->next; (just in case another node will be added)

CREATING A LINKED LIST USING malloc()

- Take note that if the newly created node happens to be the first node to be created (the list is empty or *start = NULL*) , then make sure to make *start* and *ptr2* point to that node:

  This is done by executing:

```
ptr1 = (NODE*) malloc(sizeof(NODE));

if (start == NULL)
{
    start = ptr1;
    ptr2 = ptr1;
}
```

CREATING A LINKED LIST USING malloc()

- Sample Program for Creating a Linked List (and then printing the nodes)

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

struct NODE
{
       int value;
       NODE *next;
};

main()
{
    NODE *start, *ptr1, *ptr2;
    int i, number_of_nodes;

    start = NULL;
    cout << "Enter the number of nodes: ";
    cin >> number_of_nodes;
```

CREATING A LINKED LIST USING malloc()

```
for (i = 1; i <= number_of_nodes;  ++i)
{
  ptr1 = (NODE*) malloc(sizeof(NODE));
  cout << "\nEnter the value of node " << i << ": ";
  cin >> ptr1 -> value;
  ptr1 -> next = NULL;

  if (start == NULL)
  {
      start = ptr1;
      ptr2 = ptr1;
   }
   else
   {
      ptr2 -> next = ptr1;
      ptr2 = ptr2 -> next;
   }
}
```

CREATING A LINKED LIST USING malloc()

```
    ptr1 = start;

    if (ptr1 == NULL)
      cout << "The list is empty!";
    else
    {
        i = 1;
        while (ptr1 != NULL)
       {
           cout << "\nThe value of node " << i << " is " << ptr1 -> value <<".";
           ++i;
           ptr1 = ptr1->next;
       }
  }
}
```

SOME OPERATIONS ON LINKED LISTS

- For the following programs for the operations on linked lists, assume the following global declaration:

```
struct NODE {
    int value;
    NODE *next;
};
```

and the following local declaration in *main()*:

```
NODE *start;
```

SOME OPERATIONS ON LINKED LISTS

- Since some functions for the operations on linked lists need to access the list itself, it should be able to access the pointer variable *start*. Therefore, it must be passed on to these functions as an argument.

- For example, the function *list_empty()* needs to access *start* to be able to determine if the list is empty (by checking if *start* = *NULL*). It must therefore be defined as:

int list_empty (NODE *p)

- In *main ()*, this function can be invoked by:

list_empty (start);

It should be noted that *list_empty()* does not have to modify the value of *start*.

SOME OPERATIONS ON LINKED LISTS

- Function to Determine if a Linked List is Empty

```
int list_empty (NODE *p)
{
  if (p == NULL)
    return (1);
  else
    return (0);
}
```

Take note that there is no need for a function to determine if a linked list is full.

SOME OPERATIONS ON LINKED LISTS

- Function to Print the Nodes of a Linked List

```
void print_nodes (NODE *p)
{
   int ctr;

   if (list_empty(p) == 1)
      cout << "\nThe List is Empty!";
   else
   {
      ctr = 1;
      while (p != NULL)
      {
         cout << "\nThe value of node " << ctr << " is " << p -> value <<".";
         ++ctr;
         p = p -> next;
      }
   }
}
```

To invoke this function at *main*():

print_nodes (start);

SOME OPERATIONS ON LINKED LISTS

- Function to Count the Number of Nodes in a Linked List

```
int count_nodes (NODE *p)
{
    int ctr;

    ctr = 0;
    while (p != NULL)
    {
        ++ctr;
        p = p -> next;
    }
    return (ctr);
}
```

To invoke this function at *main()*:

```
count = count_nodes (start);
```

SOME OPERATIONS ON LINKED LISTS

- Function to Locate a Node in a Linked List and Print its Position (assuming nodes are unique)

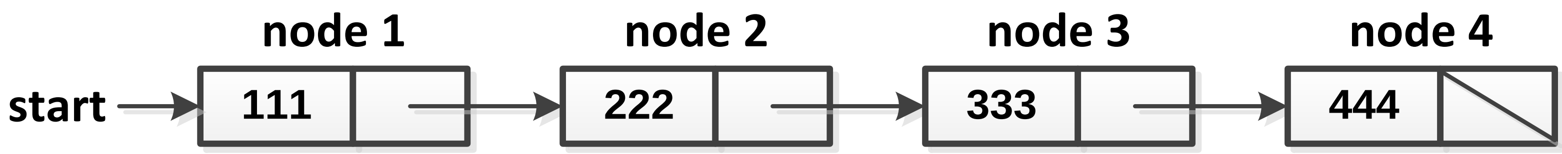
```
void locate_node (NODE *p, int search_value)
{
   int ctr;
   if (list_empty(p) == 1)
      cout << "\nThe List is Empty!";
   else
   {
      ctr = 1;
      while (p != NULL && p -> value != search_value)
      {
         p = p->next;
         ++ctr;
      }
      if (p != NULL)
         cout << "Node requested is Node " << ctr << ".";
      else
         cout << "Node does not exist.";
   }
}
```

To invoke this function at *main*():

locate_node (start, key);

INSERTING A NODE IN A LINKED LIST (AT THE REAR)

- Assume that there is already a linked list:



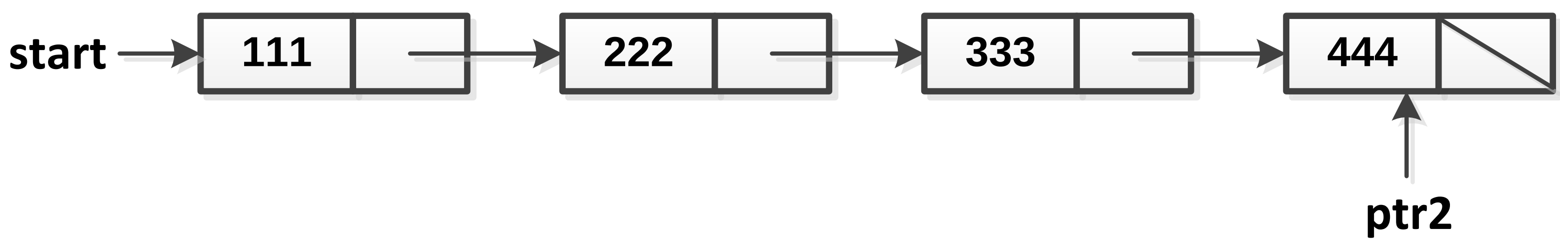
- The first step is to create a new node and enter the necessary values (assume the pointer *ptr1* will point to the new node and the variable *data* contains the value of this node):

```
ptr1 = (NODE*) malloc(sizeof(NODE));
ptr1 -> value = data;
ptr1 -> next = NULL;
```

iACADEMY
SCHOOL OF COMPUTING • SCHOOL OF BUSINESS • SCHOOL OF DESIGN

INSERTING A NODE IN A LINKED LIST (AT THE REAR)

- Next is to have a pointer (*ptr2*) point to the last element in the list (node *n*).

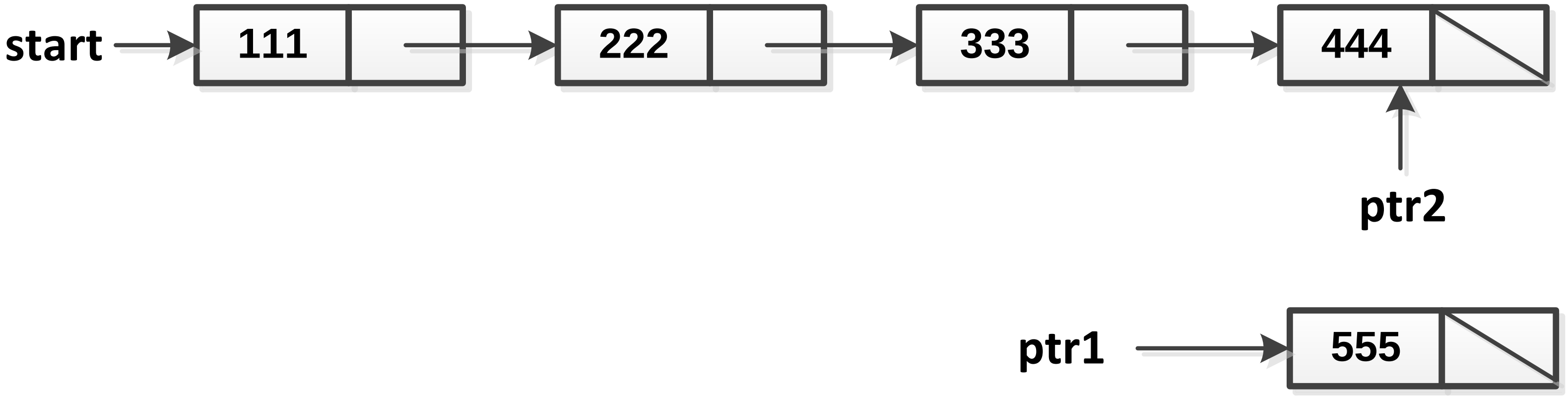


This is done by executing:

```
ptr2 = start;
while (ptr2 -> next != NULL)
    ptr2 = ptr2 -> next;
```

INSERTING A NODE IN A LINKED LIST (AT THE REAR)

- The last step is to attach the new node to the list:



This is done by executing:

```
ptr2 -> next = ptr1;
```

INSERTING A NODE IN A LINKED LIST (AT THE REAR)

- If the list is empty (*start* = *NULL*), then the pointer *start* should point to the new node.

- This would mean that the function to insert/add a node would have to modify the value of *start* which is most likely a local variable to *main()* or to some other function.

- For the *add_item()* function to do this, the address of *start* must be passed on to the function. Consequently, the function must have a pointer to *start* to store this address (a pointer to another pointer):

void add_item (…., NODE **ptr_start, …)

- In *main ()*, the *add_item()* function can be invoked by:

add_item (…., &start, …);

INSERTING A NODE IN A LINKED LIST (AT THE REAR)

- Function to Insert/Add a Node in a Linked List (at the rear)

```
void add_node (NODE **ptr_start,  int data)
{
    NODE *ptr1, *ptr2;
    ptr1 = (NODE*) malloc(sizeof(NODE));
    ptr1 -> value = data;
    ptr1 -> next = NULL;
    if (*ptr_start == NULL)
        *ptr_start = ptr1;
    else
    {
        ptr2 = *ptr_start;
        while (ptr2 -> next != NULL)
            ptr2 = ptr2 -> next;
        ptr2 -> next = ptr1;
    }
}
```

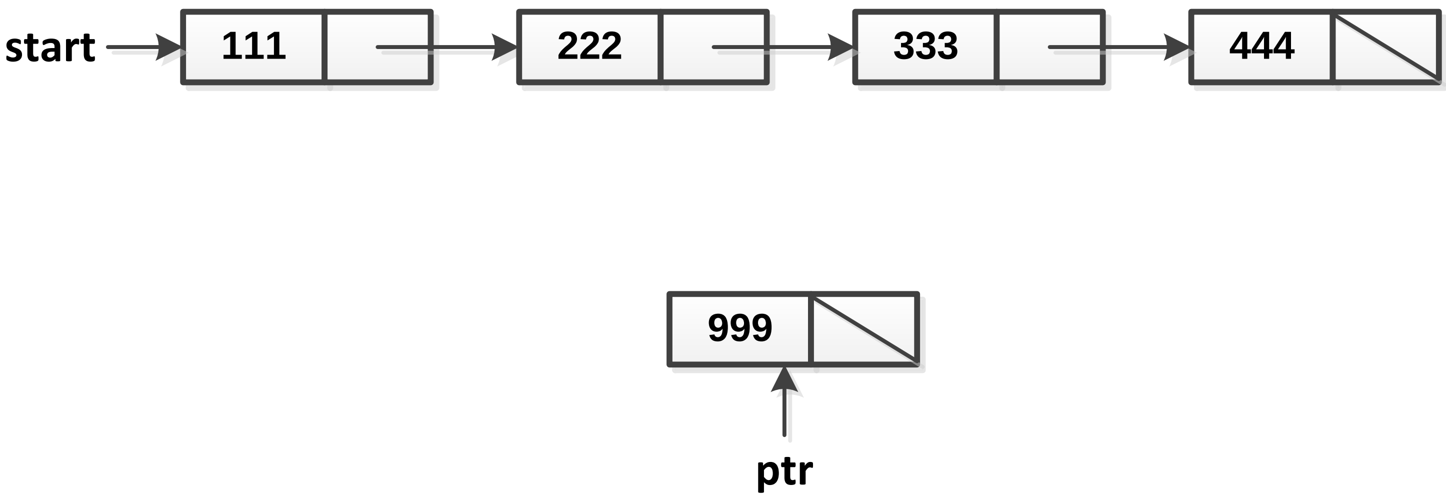
To invoke this function at *main*():

```
add_node (&start, add_data);
```

INSERTING A NODE WITHIN A LINKED LIST

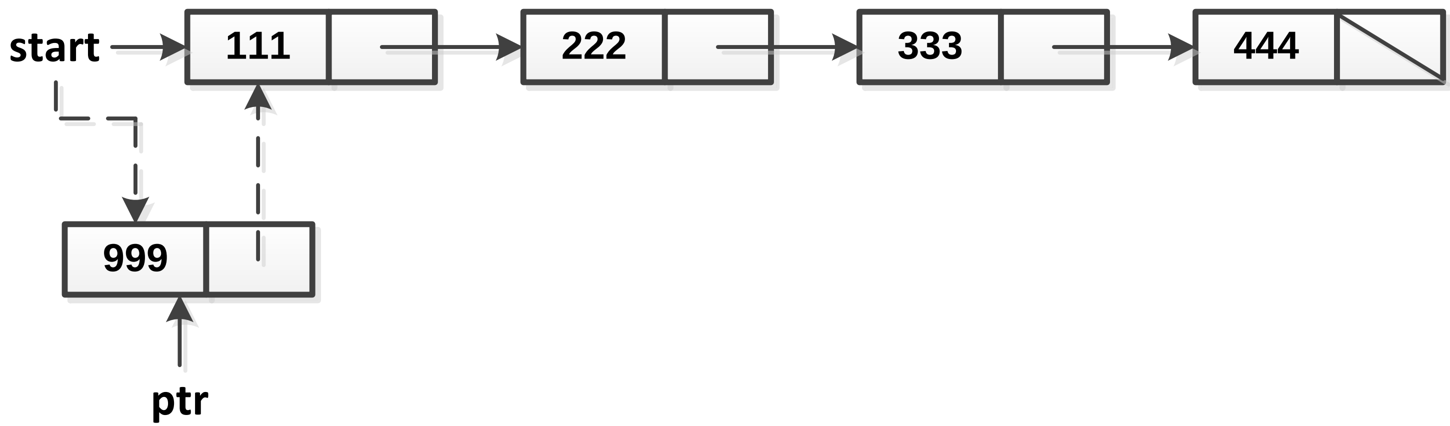
- Assume the node will be inserted at position *pos*. The first step is to create a new node and enter the necessary values (assume the pointer *ptr* will point to the new node and the variable *add_value* contains the value of this node):

```
ptr = (NODE*) malloc(sizeof(NODE));
ptr -> value = add_value;
ptr -> next = NULL;
```



INSERTING A NODE WITHIN A LINKED LIST

- If *pos* = 1 (the new node will become the first node in the list):



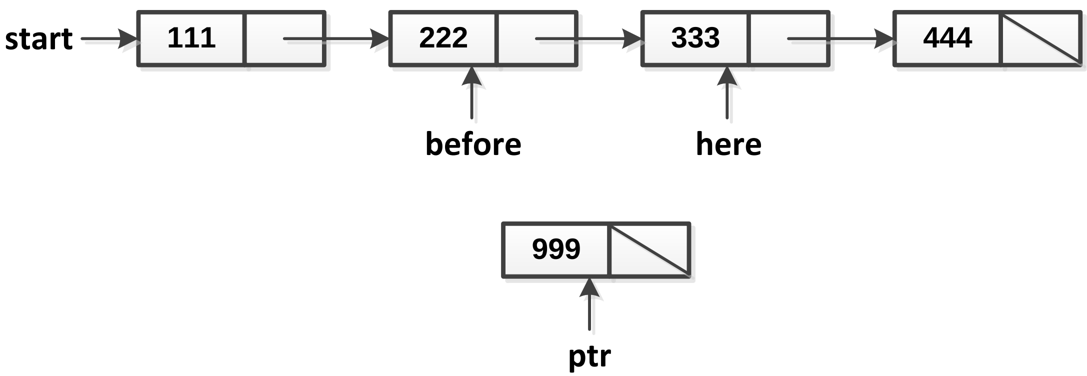
This is accomplished by the following code:

```
ptr -> next = start;
start = ptr;
```

INSERTING A NODE WITHIN A LINKED LIST

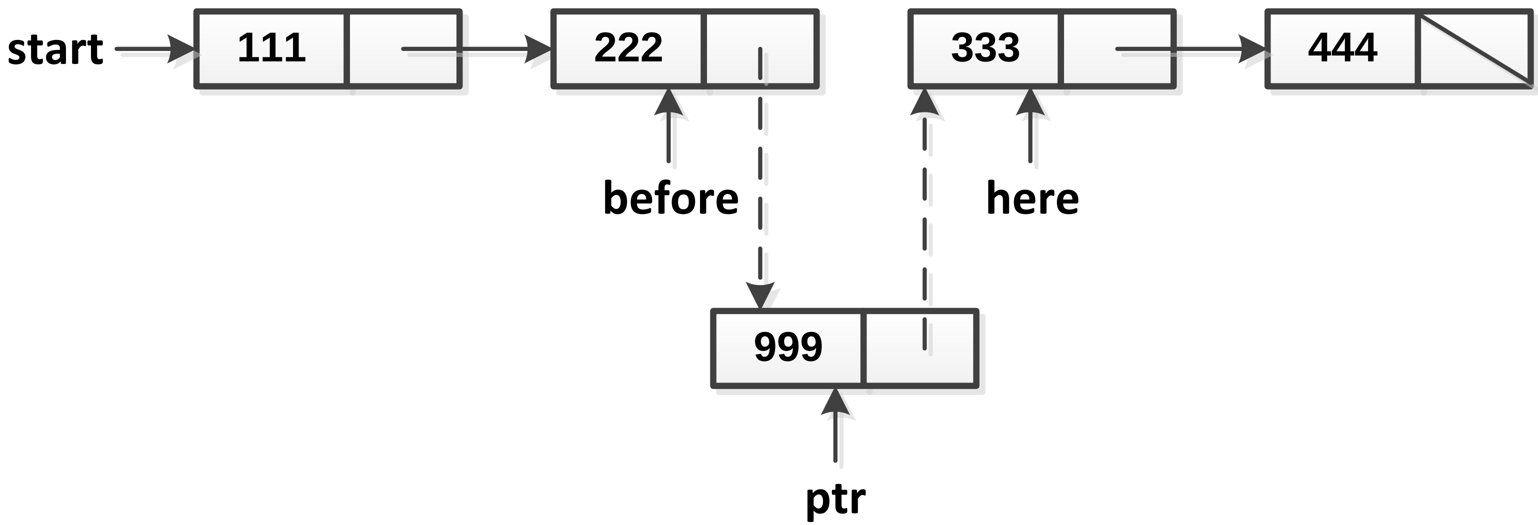
- If *pos* > 1, then the next step is to have two pointers (*before* and *here*) to point to the nodes before and at the position where the node is to be inserted. In other words, *before* should point to the node at position *pos* – 1 and *here* should point to the node at position *pos*.

Assume *pos* = 3




```
before = start;
here = start -> next;
ctr = 1;
while (ctr <= pos-2)
{
    before = before -> next;
    here = here -> next;
    ++ctr;
}
```

INSERTING A NODE WITHIN A LINKED LIST



- To insert the node (pointed to by *ptr*) in the list, execute:

before -> next = ptr;
ptr -> next = here;

INSERTING A NODE WITHIN IN LINKED LIST

- Function to Insert a Node Within a Linked List

```
void insert_node (NODE **ptr_start, int add_value, int pos)
{
    NODE *ptr, *before, *here;
    int ctr;

    ptr = (NODE*) malloc(sizeof(NODE));
    ptr -> value = add_value;
    ptr -> next = NULL;

    if (pos == 1)
    {
        ptr -> next = *ptr_start;
        *ptr_start = ptr;
    }
```

To invoke this function at *main()*:

```
insert_node (&start, add_data, position);
```

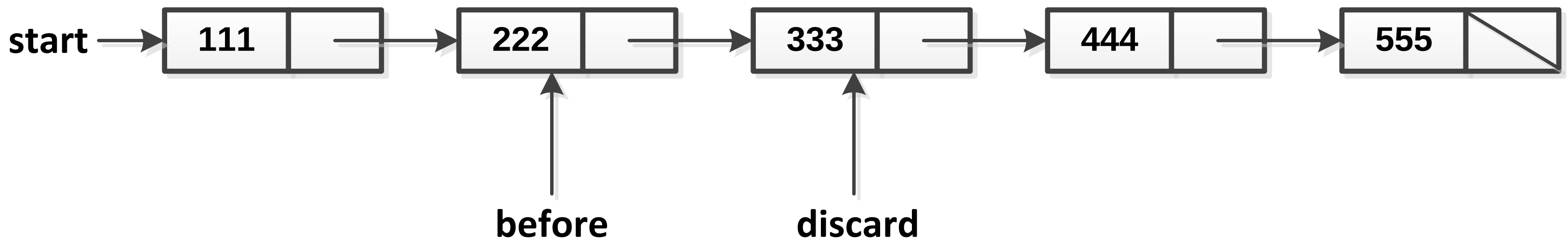
INSERTING A NODE WITHIN IN LINKED LIST

```
else
{
    before = *ptr_start;
    here = (*ptr_start) -> next;
    ctr = 1;
    while (ctr <= pos-2)
    {
        before = before -> next;
        here = here -> next;
        ++ctr;
    }
    before -> next = ptr;
    ptr -> next = here;
}
cout << "\nItem Successfully Added!";
}
```

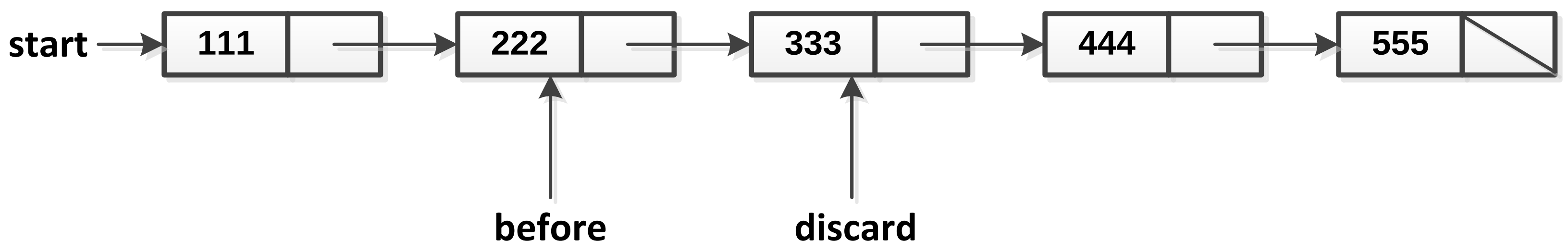
DELETING A NODE IN A LINKED LIST

- Assume that the value of the node to be deleted is stored in a variable called *delete_value*.

- There should be a pointer (*discard*) to point to the node to be deleted and another pointer (*before*) to point to the node before the node to be deleted.

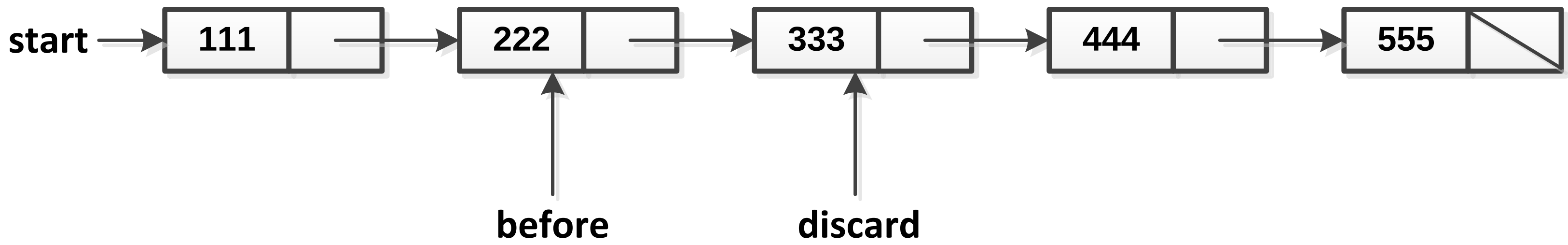


DELETING A NODE IN A LINKED LIST



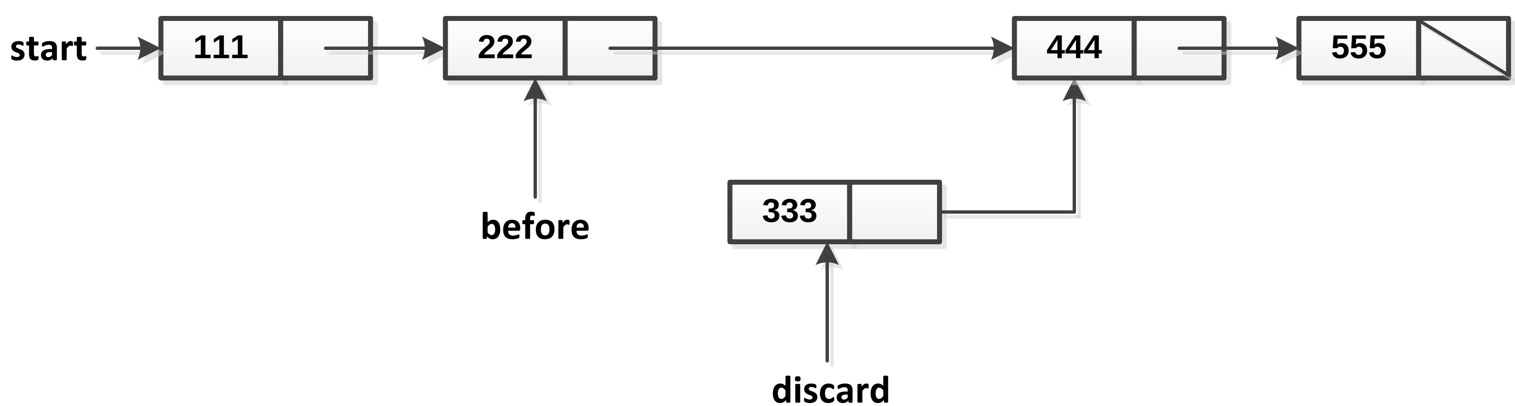

```
discard = start;
while (discard != NULL && discard->value != delete_value)
{
      before = discard;
      discard = discard->next;
}
```

DELETING A NODE IN A LINKED LIST



- To remove the node (pointed to by *discard*) from the list, execute:

before -> next = discard -> next;



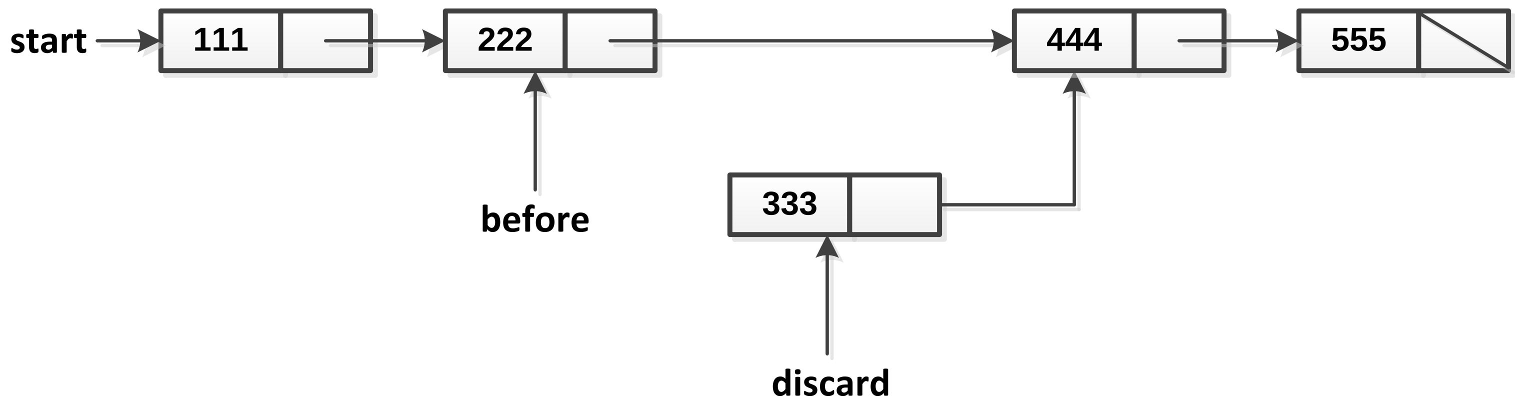
DELETING A NODE IN A LINKED LIST

- Deleting a node from a linked list actually means removing it from the list. In the example given, the node pointed to by discard still exists but it is no longer part of the list.

- Although the node containing 333 is no longer part of the linked list, it has not been destroyed. Unless the program explicitly eliminates the node, it will remain in storage and will waste precious memory space.

- C++ has a built-in function that eliminates dynamic variables to free some storage for other purposes. This function is known as *free()*. It frees the memory occupied by the node pointed to by *discard*. Note that the pointer *discard* still exists, only the node pointed to by it was destroyed.

DELETING A NODE IN A LINKED LIST



- To "destroy" the node (pointed to by *discard*) from the list, execute:

```
before -> next = discard -> next;
free(discard);
```

DELETING A NODE IN A LINKED LIST

- Some important points:

1. After making *discard* = *start* and before moving the pointer *discard*, check first if the node to be deleted is the first node (the one pointed to by *start*). If it is, let *start* point to the second node and delete the designated node. If not, then begin moving *discard* to the desired node.

2. After the *while (discard != NULL && discard->value != delete_value)* loop finishes executing and *discard* = *NULL*; this means the node to be deleted does not exist.

DELETING A NODE IN A LINKED LIST

- Function to Delete a Node in a Linked List

```
void delete_node (NODE **ptr_start, int delete_value)
{

   NODE *discard, *before;

   discard = *ptr_start;

   if (list_empty(*ptr_start) == 1)
      cout << "\nThe List is Empty!";
  else
  {
      if ((*ptr_start) -> value == delete_value)
      {
          *ptr_start = (*ptr_start) -> next;
          free(discard);
          cout << "\nItem Successfully Deleted!";
      }
```

To invoke this function at *main()*:

```
delete_node (&start, delete_data);
```

DELETING A NODE IN A LINKED LIST

```
            else
            {
                while (discard != NULL && discard -> value != delete_value)
                {
                    before = discard;
                    discard = discard -> next;
                }

                if (discard == NULL)
                    cout << "\nNode To Be Deleted Does Not Exist!";
                else
                {
                    before -> next=discard -> next;
                    free(discard);
                    cout << "\nItem Successfully Deleted!";
                }
            }
        }
    }
```

To invoke this function at *main()*:

```
delete_node (&start, delete_data);
```

iACADEMY
SCHOOL OF COMPUTING • SCHOOL OF BUSINESS • SCHOOL OF DESIGN

LINKED LISTS VERSUS ARRAYS

- Advantages of Using Linked Lists in Implementing Lists:
  - Faster/Easier in inserting/deleting nodes. Unlike in arrays wherein several elements have to be moved every time a single element is inserted or deleted in the array.
  - Dynamic size. Theoretically, using linked lists does not put any limit on the number of items the list can have. The only limitation is the size of the physical memory of the machine.

LINKED LISTS VERSUS ARRAYS

- Disadvantages of Using Linked Lists in Implementing Lists:
  - More memory space is needed. As mentioned before, an array of 1,000 *char* elements requires only 1,000 bytes. While a linked list of 1,000 *char* elements requires 16,000 bytes.
  - Direct access to a node is not allowed. The program has to access nodes sequentially starting from the first node.

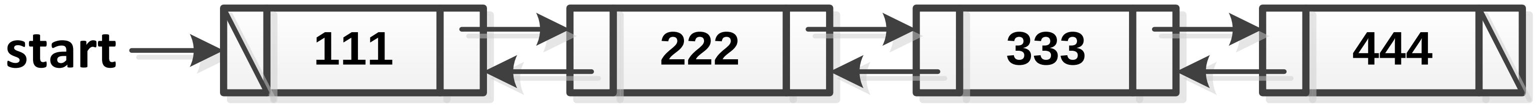
DOUBLY LINKED LISTS

- Disadvantages of singly linked lists:

  1. A program cannot traverse such a list backwards.

  2. A program cannot delete a node from the list given only a pointer to that node (there is a need for the *before* pointer).

- In a doubly linked list, each node contains two pointers. One pointer points to the node's predecessor while the other points to its successor.



DOUBLY LINKED LISTS

- Sample declaration:

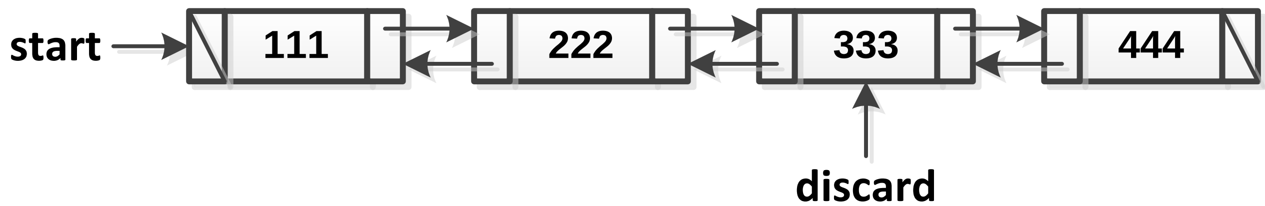
```
typedef struct node {
        int data;
        struct node *prev;
        struct node *next;
        } NODE;
```

The *prev* field points to the previous node in the list while the *next* field points to the next node in the list.

DOUBLY LINKED LISTS

- Deleting a Node

  Assume that a pointer *discard* points to the node to be deleted.



Algorithm to delete a node pointed to by *discard*:

```
discard -> next -> prev = discard -> prev;
discard -> prev-next   = discard -> next;
free (discard);
```

DOUBLY LINKED LISTS

- Function to Delete a Node in a Doubly Linked List

```
void delete_node (NODE **ptr_start,  int delete_value)
{
    NODE *discard, *before;

    discard = *ptr_start;

    if ((*ptr_start) -> value == delete_value)
    {
        *ptr_start = (*ptr_start) -> next;
        (*ptr_start) -> prev = NULL;
        free(discard);
    }
```

iACADEMY
SCHOOL OF COMPUTING • SCHOOL OF BUSINESS • SCHOOL OF DESIGN

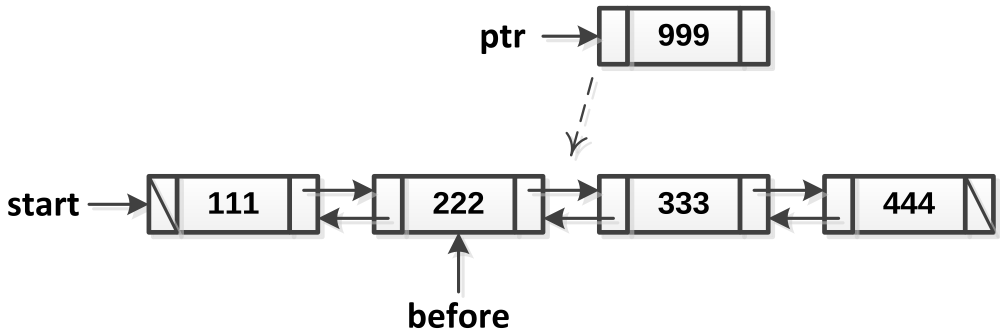
DOUBLY LINKED LISTS

```
else
{
    while (discard != NULL && discard -> value != delete_value)
        discard = discard -> next;

    if (discard == NULL)
        cout << "\nNode to be deleted does not exist!";
    else
    {
     discard -> next -> prev = discard -> prev;
     discard -> prev -> next   = discard -> next;
     free(discard);
    }
}
}
```

DOUBLY LINKED LISTS

- Inserting a Node

  Assume that a pointer *before* points to the node that precedes the node to be inserted and that pointer new points to the node to be inserted.



Algorithm to insert a node pointed:

```
ptr -> next = before -> next;
ptr -> prev = before;
before -> next-> prev = ptr;
before -> next = ptr;
```