Link Lists

Structure: Review

```
struct student
    {
        string name;
        int rollNumber;
        float cgpa;
    };
```

- Previously, we have used **array of structure** to store the information of 3 students.

```
student input()
{
    student stu;
    cout << "Enter Student Name:" << endl;
    cin.ignore();
    getline(cin, stu.name);
    cout << "Enter Student Roll Number:" << endl;
    cin >> stu.rollNumber;
    cout << "Enter Student CGPA:" << endl;
    cin >> stu.cgpa;
    return stu;
}
```

```
void print(student stu)
{
    cout << stu.name;
    cout << "\t";
    cout << stu.rollNumber;
    cout << "\t";
    cout << stu.cgpa;
    cout << endl;
}
```

```
main()
{
    student total_stu[3];
    for (int i = 0; i < 3; i++)
    {
        total_stu[i] = input();
    }
    for (int i = 0; i < 3; i++)
    {
        print(total_stu[i]);
    }
}
```

Structure: Review

- In this we had specify the size of the array before compilation.

```
struct student
    {
        string name;
        int rollNumber;
        float cgpa;
    };
```

```
student input()
{
    student stu;
    cout << "Enter Student Name:" << endl;
    cin.ignore();
    getline(cin, stu.name);
    cout << "Enter Student Roll Number:" << endl;
    cin >> stu.rollNumber;
    cout << "Enter Student CGPA:" << endl;
    cin >> stu.cgpa;
    return stu;
}
```

```
void print(student stu)
{
    cout << stu.name;
    cout << "\t";
    cout << stu.rollNumber;
    cout << "\t";
    cout << stu.cgpa;
    cout << endl;
}
```

```
main()
{
    student total_stu[3];
    for (int i = 0; i < 3; i++)
    {
        total_stu[i] = input();
    }
    for (int i = 0; i < 3; i++)
    {
        print(total_stu[i]);
    }
}
```

Structure: Dynamic Allocation of Memory

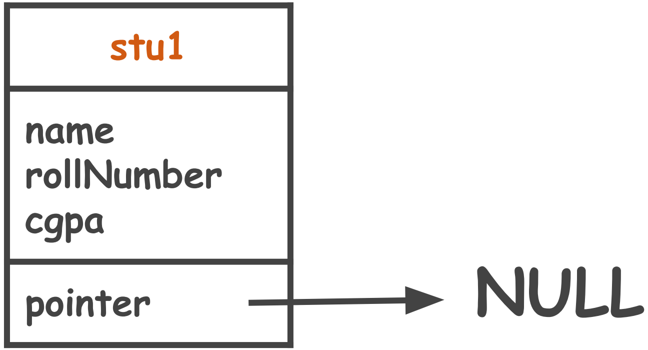
- Is there a way, in which we do not have to specify the size at compilation time, so, we could add the records as we need at the execution time?

Structure: Dynamic Allocation of Memory

- Yes, We can do that with the help of pointers.

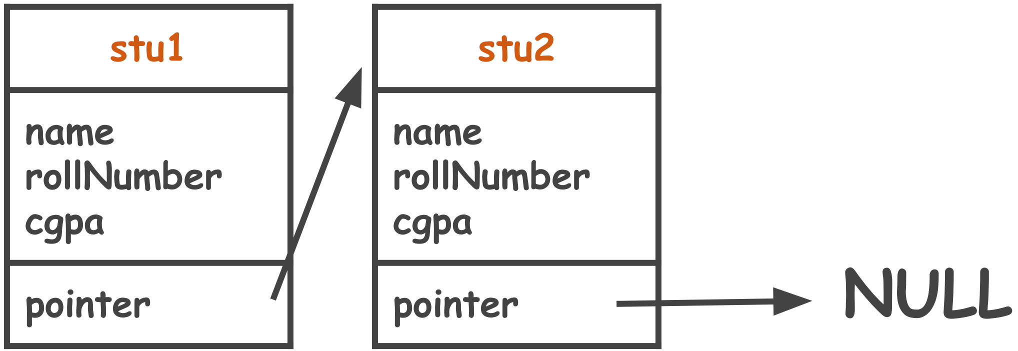
Structure: Dynamic Allocation of Memory

- Yes, We can do that with the help of pointers.



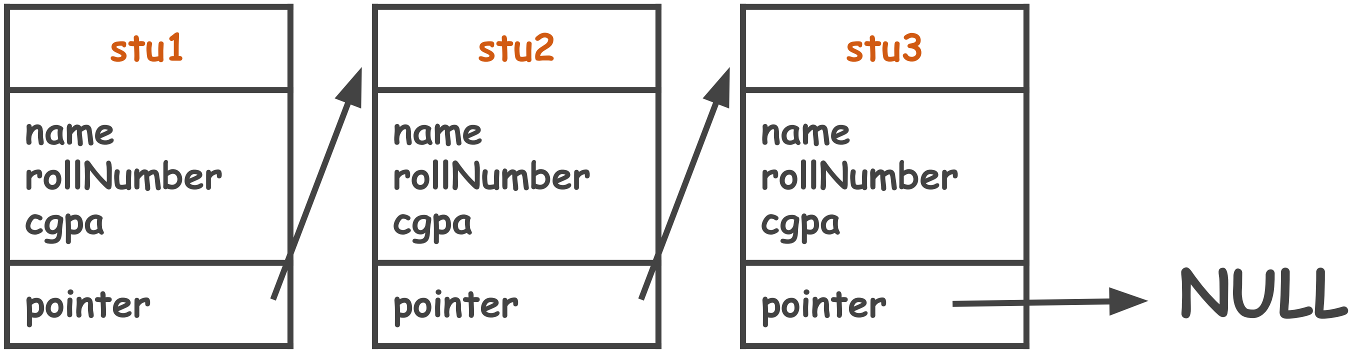
Structure: Dynamic Allocation of Memory

- Yes, We can do that with the help of pointers.



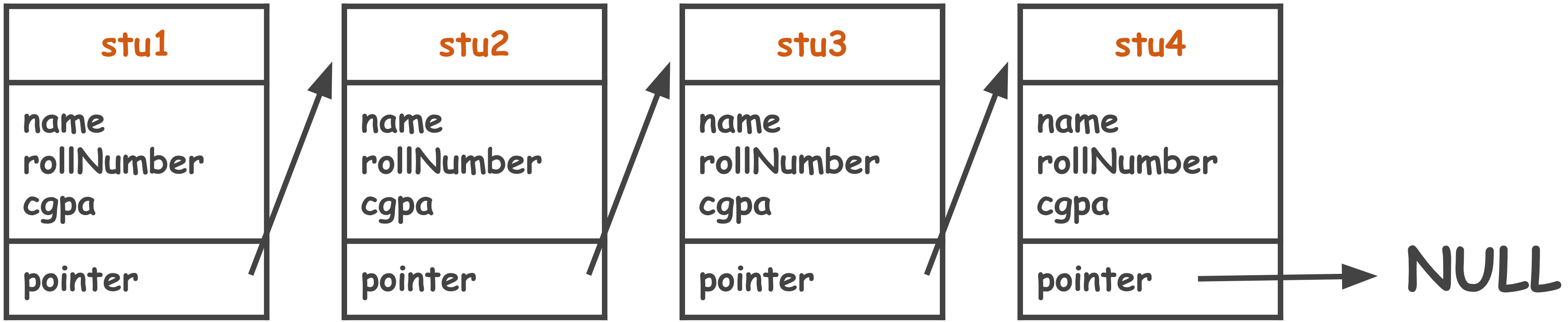
Structure: Dynamic Allocation of Memory

- Yes, We can do that with the help of pointers.



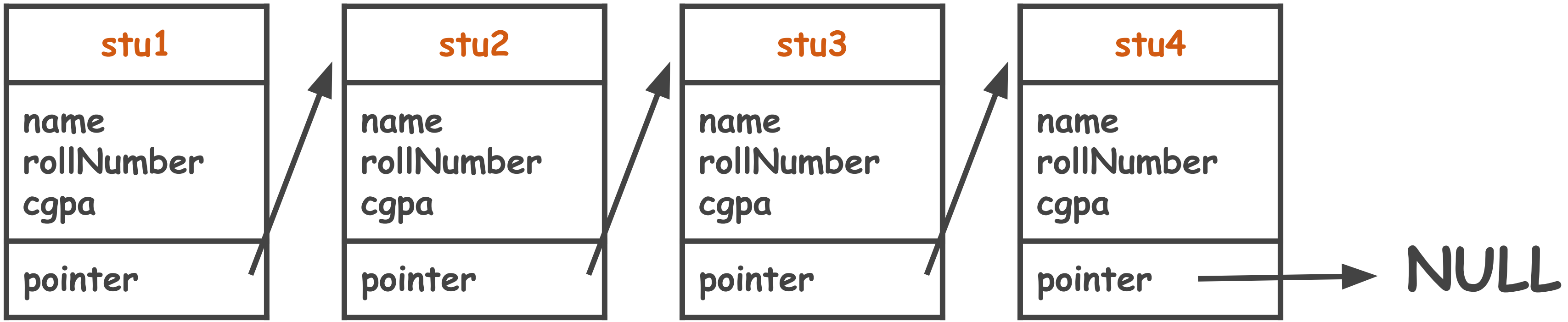
Structure: Dynamic Allocation of Memory

- Yes, We can do that with the help of pointers.



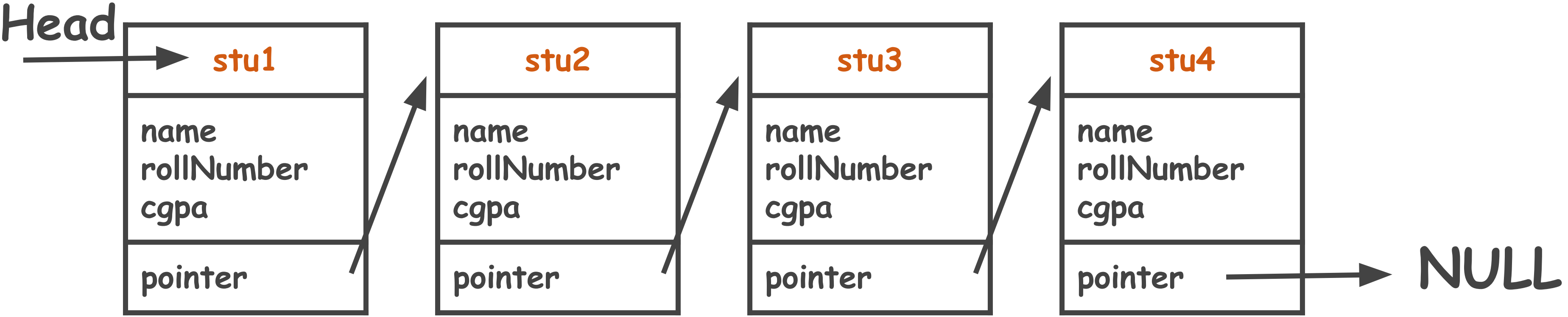
Structure: Dynamic Allocation of Memory

- In order to iterate on this data, we have to keep a pointer that will always point at the start of the data.



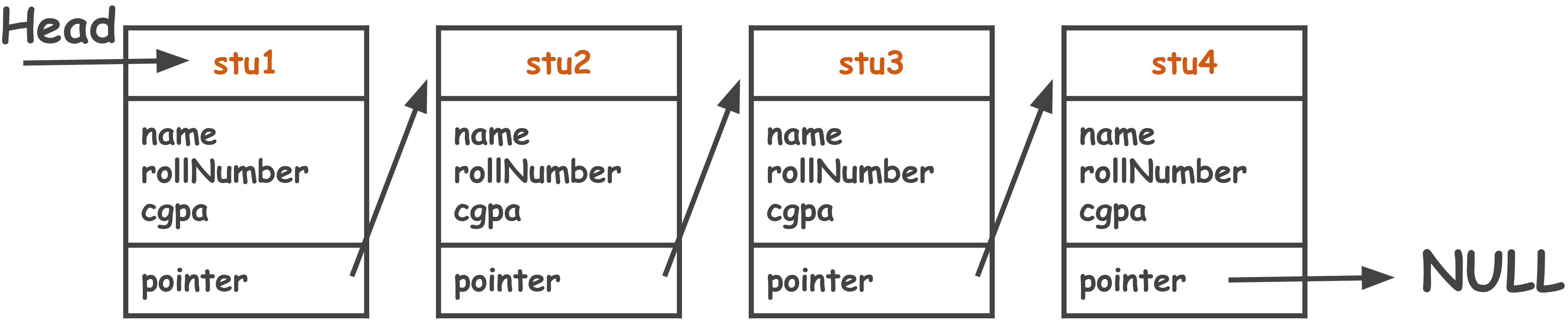
Structure: Dynamic Allocation of Memory

- In order to iterate on this data, we have to keep a pointer that will always point at the start of the data.



Linked List

- This way of storing the information dynamically is called linked list.



Linked List

- We add a pointer at the end of the student structure.

```
struct student
{
    string name;
    int rollNumber;
    float cgpa;
};
```

Previously

```
struct student
{
    string name;
    int rollNumber;
    float cgpa;
    student *next;
};
```

Updated

Linked List

- We add a pointer at the end of the student structure.

```
struct student
{
    string name;
    int rollNumber;
    float cgpa;
};
```

Previously

```
struct student
{
    string name;
    int rollNumber;
    float cgpa;
    student *next;
};
```

Pointer that will point to the next student record

Updated

Linked List

First of all, we will make a pointer that will point towards the start of the Link List. Initially, there is no student record therefore, the starting pointer (let's call it head) will point towards NULL.

```
#include <iostream>
using namespace std;
struct student
{
    string name;
    int rollNumber;
    float cgpa;
    student *next;
};

student *head = NULL;
```

Link List: Take Input

- Lets see how we can take input from the user.

Linked List

We will take input from the user using the function **takeInput()**.

```
#include <iostream>
using namespace std;
struct student
{
    string name;
    int rollNumber;
    float cgpa;
    student *next;
};
student *head = NULL;
void takeInput()
{
    string name;
    int roll;
    float cgpa;
    cout << "Enter Student Name: ";
    cin >> name;
    cout << "Enter Student Roll Number: ";
    cin >> roll;
    cout << "Enter Student CGPA: ";
    cin >> cgpa;
    addRecord(name, roll, cgpa);
}
```

Link List: Insertion at the End

- Lets see how we can insert records at the end of the link list.

Linked List

Now we will make the records dynamically as the user enters the input.

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
}
```

Linked List

Now we will make the records dynamically as the user enters the input.

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
}
```

new is a keyword that allocates the memory dynamically

Linked List

Previously, we had used dot (.) operator to access the elements of the structure.

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
}
```

Linked List

Previously, we had used dot (.) operator to access the elements of the structure.

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
}
```

The Dot(.) operator is used to normally access members of a structure.
The Arrow(->) operator exists to access the members of the structure using pointers.

Linked List

Now, we have to see if the user has entered the first record then we have to set the head pointer at that record.

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
    if (head == NULL)
    {
        head = record;
    }
}
```

Linked List

If the user enters any other record then we have to store at the **end** of the link list.

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
    if (head == NULL)
    {
        head = record;
    }
    else
     {
        student *temp = getLastRecord(head);
        temp->next = record;
     }
}
```

Linked List

If the user enters any other record then we have to store at the **end** of the link list.

For that, we have to find the **last record**.

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
    if (head == NULL)
    {
        head = record;
    }
    else
     {
        student *temp = getLastRecord(head);
        temp->next = record;
     }
}
```

Linked List

If the user enters any other record then we have to store at the **end** of the link list.

```
student *getLastRecord(student *temp)
{
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    return temp;
}
```

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
    if (head == NULL)
    {
        head = record;
    }
    else
     {
        student *temp = getLastRecord(head);
        temp->next = record;
     }
}
```

Linked List

For that we keep iterating on the link list until the last record points towards NULL.

```
student *getLastRecord(student *temp)
{
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    return temp;
}
```

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
    if (head == NULL)
    {
        head = record;
    }
    else
     {
        student *temp = getLastRecord(head);
        temp->next = record;
     }
}
```

Linked List

For that we keep iterating on the link list until the last record points towards NULL.

```
student *getLastRecord(student *temp)
{
    while (temp->next != NULL)
    {
        temp = temp->next;
    }
    return temp;
}
```

```
void addRecord(string n, int roll, float grade)
{
    student *record = new student;
    record->name = n;
    record->rollNumber = roll;
    record->cgpa = grade;
    record->next = NULL;
    if (head == NULL)
    {
        head = record;
    }
    else
     {
        student *temp = getLastRecord(head);
        temp->next = record;
     }
}
```

Then we store the record at the end of the link list.

Link List: Traverse the Records

- Lets see how we can print all the records of the link list.

Linked List

First of all, let's make a **function** to print a single record.

```
void printSingleRecord(student *temp)
{
    cout << "Name: ";
    cout << temp->name << endl;
    cout << "Roll Number: ";
    cout << temp->rollNumber << endl;
    cout << "CGPA: ";
    cout << temp->cgpa << endl;
}
```

Linked List

Now, Let's print all the records of the link list starting from the **head** of the link list.

Until the **temp** pointer reaches the end of the link list keep on printing the records.

```
void printSingleRecord(student *temp)
{
    cout << "Name: ";
    cout << temp->name << endl;
    cout << "Roll Number: ";
    cout << temp->rollNumber << endl;
    cout << "CGPA: ";
    cout << temp->cgpa << endl;
}

void printRecords()
{
    student *temp = head;
    while (temp != NULL)
    {
        printSingleRecord(temp);
        temp = temp->next;
    }
}
```

Link List: Search the Records

- Lets see how we can search a specific item from the link list.

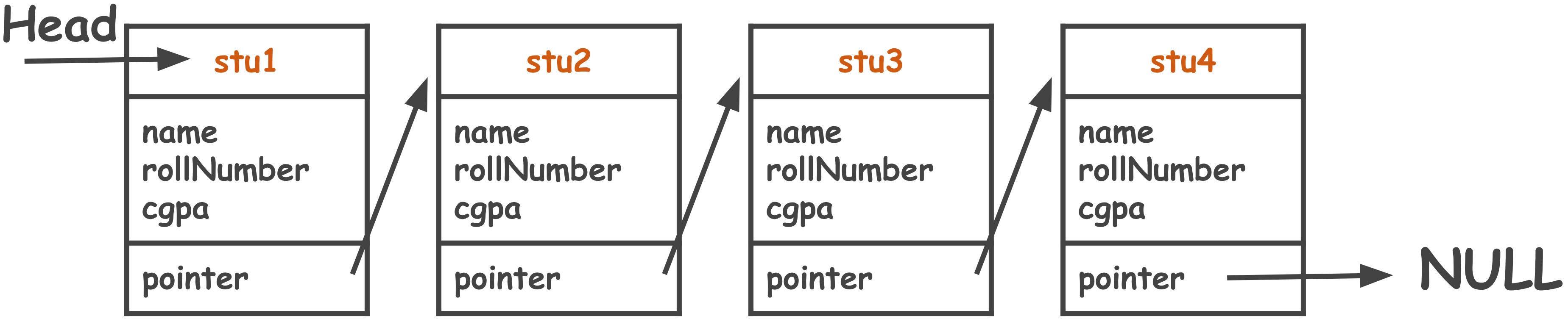
Linked List

Let's find a specific name of the student in the link list.

```
void searchRecord(string n)
{
    student *temp = head;
    bool isFound = false;
    while (temp != NULL)
    {
        if(temp->name == n)
        {
            cout << "Record found" << endl;
            isFound = true;
            break;
        }
        temp = temp->next;
    }
    if(isFound == false)
    {
        cout << "Record not found" << endl;
    }
}
```

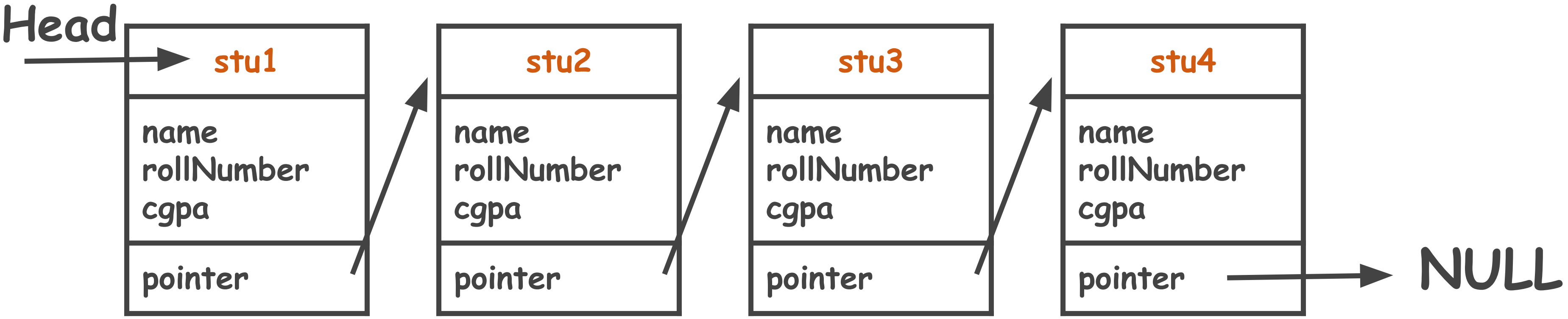
Link List: Delete the Records

- Lets see how we can delete a specific record from the link list.



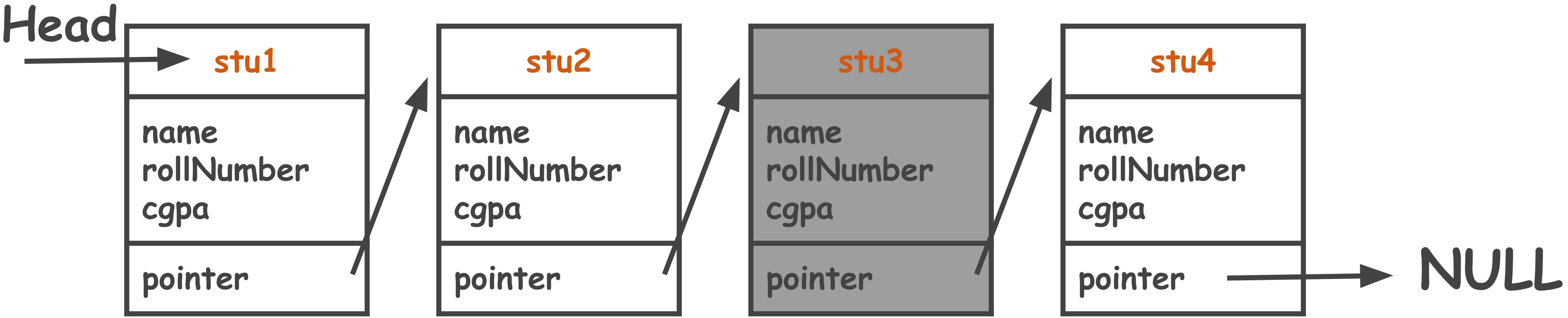
Link List: Delete the Records

- Suppose we want to delete the student with name Ibrahim. We searched the link list and found that he is present at stu3.



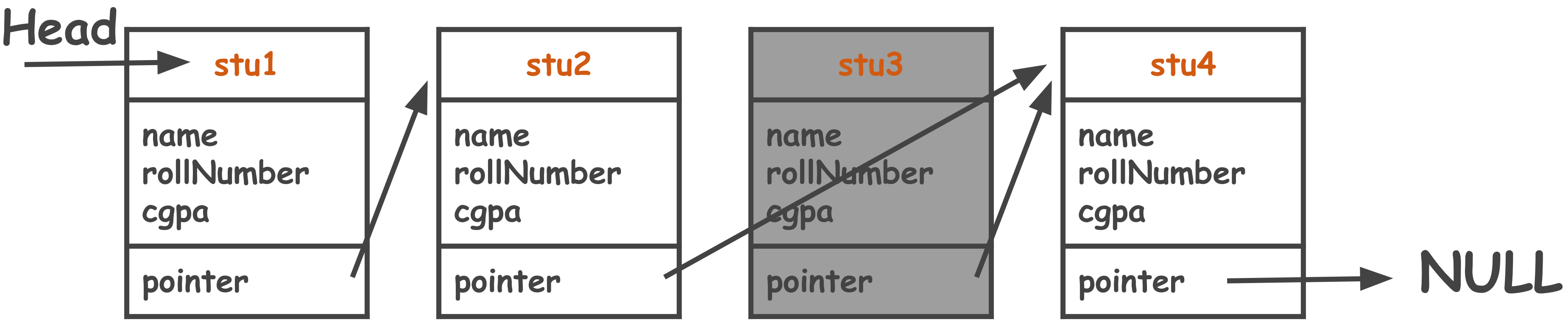
Link List: Delete the Records

- Suppose we want to delete the student with name XYZ. We searched the link list and found that he is present at stu3.



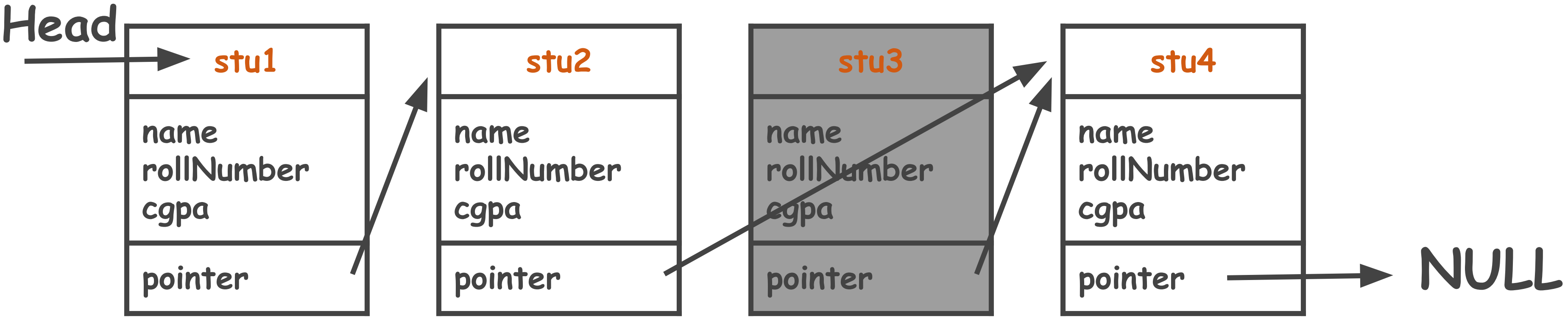
Link List: Delete the Records

- We will just change the pointer of the stu2 so that it starts pointing towards stu4.



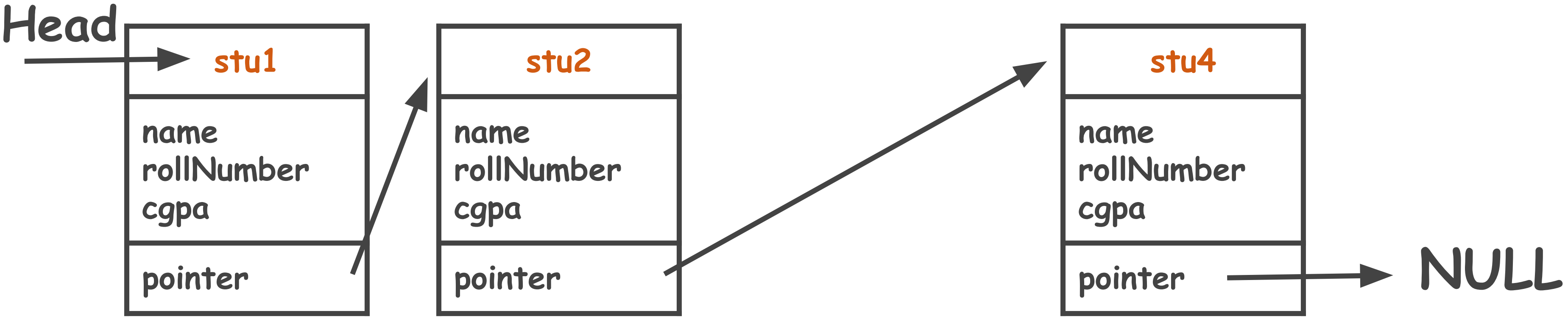
Link List: Delete the Records

- There will no pointer left that will point towards stu3 and we will delete stu3 from the memory using the keyword delete.



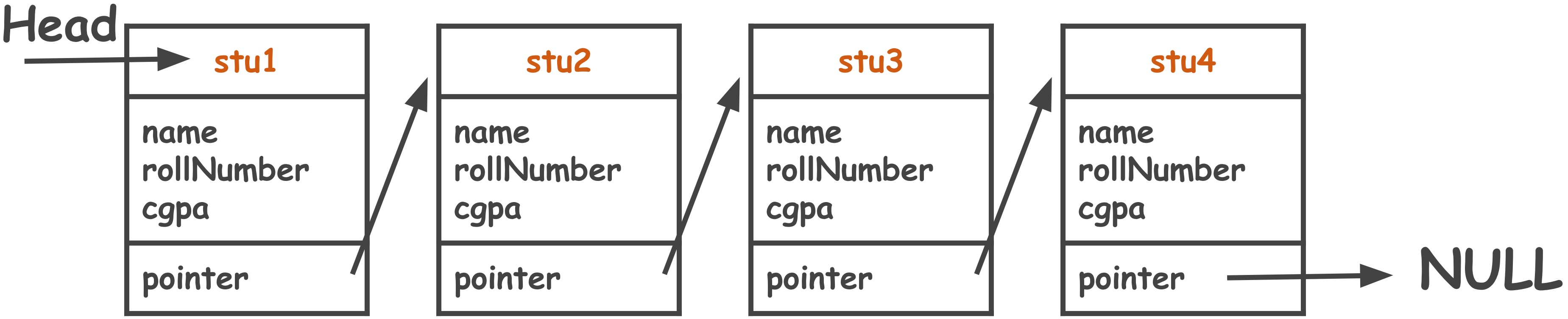
Link List: Delete the Records

- There will no pointer left that will point towards stu3 and we will delete stu3 from the memory using the keyword delete.



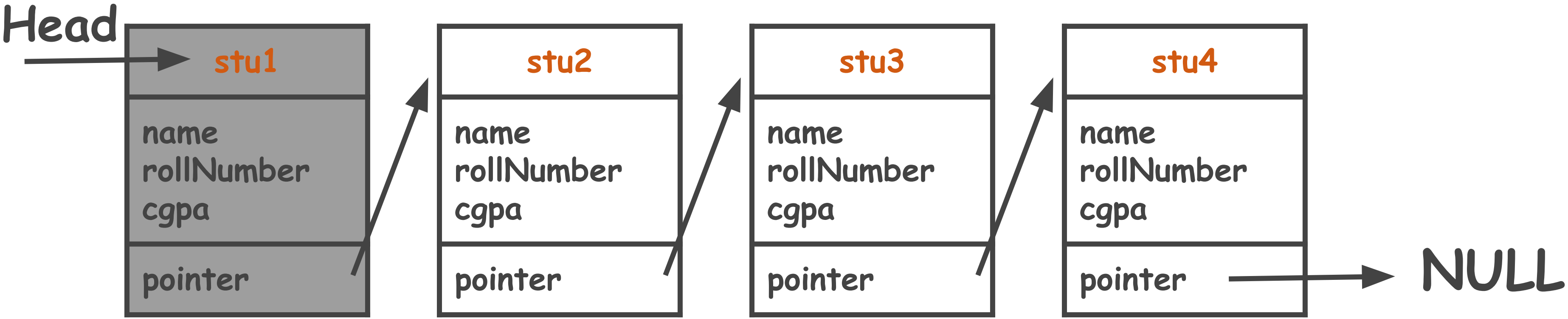
Link List: Delete the 1st Record

- Lets see how we can delete a specific record found at the start of the link list.



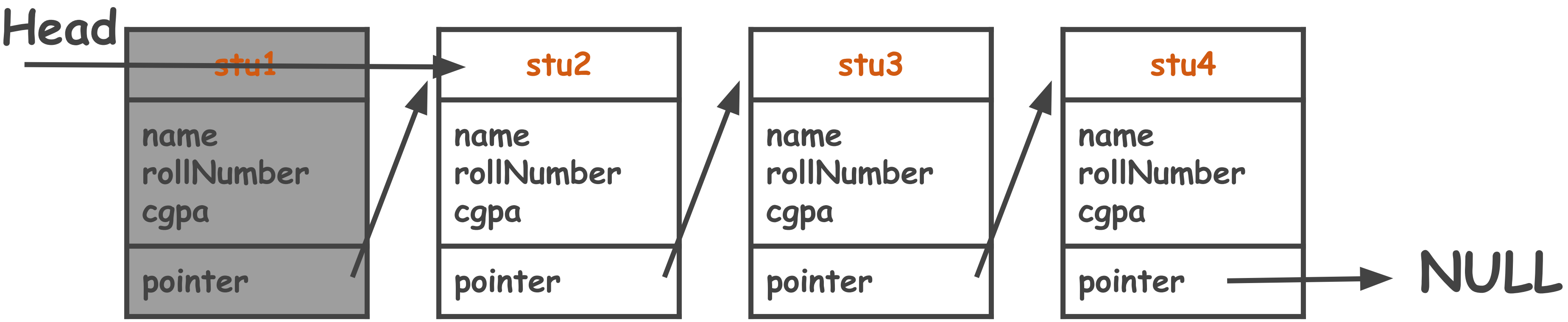
Link List: Delete the 1st Record

- Lets see how we can delete a specific record found at the start of the link list.



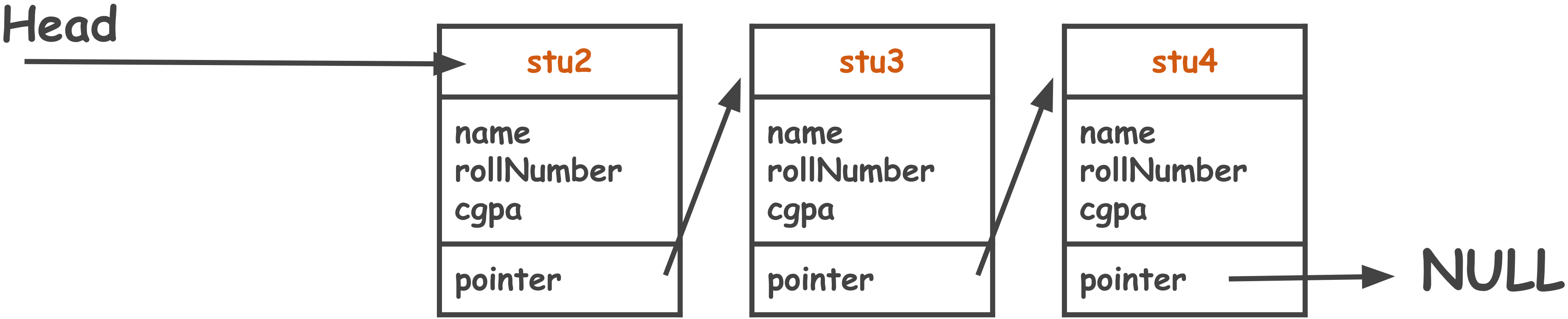
Link List: Delete the 1st Record

- We just change the **head** pointer so that it starts from the next record.



Link List: Delete the 1st Record

- Then delete the first record using the delete keyword.



Linked List

Let's find a specific name of the student in the link list and then delete that record.

```
void deleteRecord(string n)
{
    student *temp = head;
    student *temp1 = head;
    bool isFound = false;
    int count = 0;
    while (temp != NULL){
        count = count + 1;
        if(temp->name == n){
            isFound = true;
            break;
        }
        temp1 = temp;
        temp = temp->next;
    }
    if(isFound == true && count == 1){
        head = head->next;
        delete temp;
    }
    else if(isFound == true && count > 1){
        temp1->next = temp->next;
        delete temp;
    }
}
```

Linked List

Let's find a specific name of the student in the link list and then delete that record.

Here, we check if the current record has the student name that we want to find.

```
void deleteRecord(string n)
{
    student *temp = head;
    student *temp1 = head;
    bool isFound = false;
    int count = 0;
    while (temp != NULL){
        count = count + 1;
        if(temp->name == n){
            isFound = true;
            break;
        }
        temp1 = temp;
        temp = temp->next;
    }
    if(isFound == true && count == 1){
        head = head->next;
        delete temp;
    }
    else if(isFound == true && count > 1){
        temp1->next = temp->next;
        delete temp;
    }
}
```

Linked List

Let's find a specific name of the student in the link list and then delete that record.

Case 1:

If the record is found at the start of the link list

```
void deleteRecord(string n)
{
    student *temp = head;
    student *temp1 = head;
    bool isFound = false;
    int count = 0;
    while (temp != NULL){
        count = count + 1;
        if(temp->name == n){
            isFound = true;
            break;
        }
        temp1 = temp;
        temp = temp->next;
    }
    if(isFound == true && count == 1){
        head = head->next;
        delete temp;
    }
    else if(isFound == true && count > 1){
        temp1->next = temp->next;
        delete temp;
    }
}
```

Linked List

Let's find a specific name of the student in the link list and then delete that record.

Case 2:

If the record is found at any other location of the link list

```
void deleteRecord(string n)
{
    student *temp = head;
    student *temp1 = head;
    bool isFound = false;
    int count = 0;
    while (temp != NULL){
        count = count + 1;
        if(temp->name == n){
            isFound = true;
            break;
        }
        temp1 = temp;
        temp = temp->next;
    }
    if(isFound == true && count == 1){
        head = head->next;
        delete temp;
    }
    else if(isFound == true && count > 1){
        temp1->next = temp->next;
        delete temp;
    }
}
```

Learning Objective

Write a C++ program to store records usings user defined dataTypes (Struct) through link list dynamically.

Conclusion

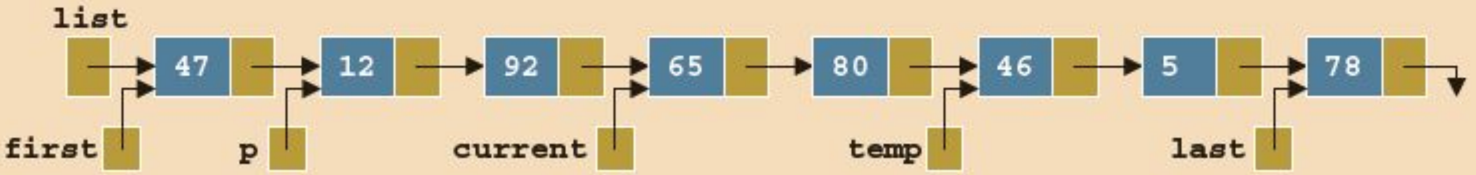
Arrays	Linked lists
Arrays have fixed size	Linked list size is dynamic
Insertion of new element is expensive	Insertion/deletion is easier by just changing the pointer
Random access is allowed using index of the array	Random access not possible as we have to start at the head always
Elements are at contiguous location	Elements have non-contiguous location
No extra space is required for the next pointer	Extra memory space required for next pointer

Self Assessment:

1. Suppose that you have the following definitions:

```
struct nodeType
{
    int info;
    nodeType *link;
};
```

```
nodeType *list, *first, *current, *last, *temp, *trail, *p, *q;
```



Self Assessment:

```
struct nodeType
{
    int info;
    nodeType *link;
};
```

What will be the Output of the following:

a. `cout << p->info;`

b. `q = p->link;`
   `cout << q->info << " " << current->info;`

c. `cout << current->link->info;`

d. `trail = current->link->link;`
   `trail->link = nullptr;`
   `cout << trail->info;`

e. `cout << last->link->info;`

f. `q = current->link; cout << q->link->link->info`

Self Assessment:

```
struct nodeType
{
    int info;
    nodeType *link;
};
```

What is the value of each of the following relational expressions?

a. p->link->link == current
b. first->link->link->info == 92
c. temp->link == 0
d. last->link == nullptr
e. list->link == p
f. p->link->link->link->info == temp->info