CSCI 2270 – Data Structures

Recitation 2, Summer 2020

Objectives:

1. Structs

Why struct?
There are some instances wherein you need more than one variable to represent a complete object. As a student of CU, you should provide name, email, birthday, address to the college.

You could write code as follows:

```
std::string name;
std::string email;
int birthday;
std::string address;
```

But, the problem with this is you have 4 independent variables that are not grouped.

What is a struct?
C++ allows us to declare an aggregated(grouped) user-defined data type. This user-defined data type can, in turn, hold multiple variables of different data types (imagine an array that holds multiple values of different data types).

```
struct student
```

```
{
        std::string name;
        std::string email;
        int birthday;
        std::string address;
};
```

2. Pointer to a struct

We can make pointers to store addresses of any type of variables, even struct variables.

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

struct Distance
{
        int feet ;
        int inch ;
};

int main()
{
        Distance d;
        // declare a pointer to Distance variable
        Distance* ptr;
        d.feet=8;
        d.inch=6;

        //store the address of d in p
        ptr = &d;
        cout<<"Distance="<< ptr->feet << "ft"<< ptr->inch << "inches";
        return 0;
}
```

Output

Why don't you try this one yourself?

You may be wondering about '−>'. Recall to access members of a struct variable we use '.' operator (e.g. d.foot = 8). When we have a pointer to a struct variable to use the member variables we will use −> operator.

3. Pass-by-value vs Pass-by-pointer vs Pass-by-reference

Pass by Value

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

void add2 (int num)
{
    num = num + 2;
}

int main ()
{
    int a = 10;
    add2(a);
    cout << a;
}
```

What do you think the output will be? 12?
To your surprise, it will be just 10. When we pass a variable as an argument to a function in the system stack the function creates a local copy of the variable and performs the operation on that local copy. The caller function (main) has no knowledge of that local copy. Hence the change is local to the callee function (add2).

Pass by Pointers

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

void add2 (int * num)
{
    *num = *num + 2;
}

int main ()
{
    int a = 10;
    add2( &a );
    cout << a;
}
```

In this case, we are passing the address of a. The function will again create a local copy of the address. However, since both of these addresses are the same, they will refer to the variable a. Hence changing the value at the pointer will change the value of a and that change will be persisted.

Pass by Reference

```
#include <iostream>
using namespace std;
void add2 (int &num)
{
    num = num + 2 ;
}

int main ()
{
    int a = 10 ;
    add2( a );
    cout << a;
}
```

In C++, we can pass parameters to a function either by pointers or by reference. When you pass a parameter by reference, the parameter inside the function is an alias to the variable you passed from the outside. On the other hand, when you pass a variable by a pointer, you take the address of the variable and pass the address into the function.

The syntax for passing a variable by reference is pretty straightforward. In the function definition/declaration, you put the '&' sign before those variables that you wish to pass by reference.

For example:

void addition(int a, int &b)

In the above function, the variable a is passed by value while the variable b is passed by reference.

Now examine the following code and try to figure out what will happen?

```
void add2 (int a[], int len)
{
    for (int i=0; i<len; i++)
    a[i]+= 2;
}
int main ()
{
    int a[] = { 1 , 2 , 3 };
    add2( a, 3 );
    for ( int i=0;i< 3;i++)
      cout << a[i] << endl ;
}
```

The output will be

```
3
4
5
```

This is because by default the variable corresponding to an array is a pointer to the first element of the array. So, when we modify the array values inside the function **add2**, we are essentially modifying the values at some address in the memory. That's why those changes are reflected in the main function as well. This can be considered as a special property of arrays.

Is the previous one similar/different to the one given below?

```
void add2 ( int *a, int len)
{
    for ( int i=0;i<len;i++)
    a[i]+= 2 ;
}

int main ()
{
    int a[] = { 1 , 2 , 3 };
    add2( &a[ 0 ], 3 );
    for ( int i=0; i<3; i++)
      cout << a[i] << endl ;
}
```

The output will once again be

```
3
4
5
```

In this example, the **add2** function expects the parameter "a" to be a pointer (based on function definition). So, this case is similar to the pass by pointer case that we discussed earlier. Hence, the changes made to the variable "a" in **add2** are also reflected in the main function.

Tip!
Pass By Value
If I send a fax to someone to sign, he creates a local copy i.e prints it and then signs. The person makes changes to his local copy that he printed out(The signature won't be reflected in my original copy). He has to scan it and send it back, for me to see his signed document.

Pass by reference
Now consider I send my address to the person who is required to sign. The person comes to my place and then signs. In this case, the person is modifying my original document.

4. Pass Pointer by Reference

As mentioned earlier, in C++, when we pass a parameter by reference, the parameter inside the function is an alias to the variable we passed from the outside. So, any changes that are made to that alias inside the function are also reflected outside that function.

What happens when the parameter that you passed by reference is a pointer?

Well, we know that a pointer is a variable that stores the address of another variable. If a pointer parameter is passed by reference to a function, then any change that we make to that pointer variable inside that function will also be reflected outside that function.

How do you pass a pointer by reference?

1) To pass a parameter by reference, we use the following syntax in function definition/declaration:

void square(int &x)   // variable x is passed by reference

2) To pass a pointer to a function, we use the following syntax in function definition/declaration:

void square(int* x)   // variable x is a pointer

3) Therefore, to pass a pointer variable by reference, we use the following syntax in function definition/declaration:

void square(int *& x)   // variable x is a pointer that is being passed by reference

What do you think will be the output for the following lines of code?

```
int temp = 100;
// change ref to ptr
void func(int *& x)
{
    x = &temp;
}

int main ()
{
    int var=3;
    int *ptr_to_var= &var;
    cout << "Before :" << *ptr_to_var << endl;
    func(ptr_to_var);
    cout << "After :" << *ptr_to_var << endl;
    return 0;
}
```

The output will be

```
Before : 3
After : 100
```

This happened because, inside the function **func,** the pointer variable (ptr_to_var) was made to point to the variable temp, i.e. it now stores the address of the variable "temp" and not of the variable "var". Since the pointer variable was passed by reference, these changes were then also reflected in the main function.

5. Dynamic Memory

What is static memory allocation?
When we declare variables, we are preparing the variables that will be used. This way a compiler can know that the variable is an important part of the program. So, the compiler allocates spaces to all these variables during compilation of the program. **The variables allocated using static memory allocation will be stored in *STACK*.**

But wait, what if the input is not known during compilation? Say, you want to pass input in the command line arguments (During runtime). The size of the input is not known beforehand during compilation.

The need for dynamic memory!!
We suffer in terms of inefficient storage use and lack or excess of slots to enter data. This is when dynamic memory play an important role. This method allows us to create storage blocks or room for variables during run time. Now there is no wastage. **The variables allocated using dynamic memory allocation will be stored in *HEAP*.**

Tip!
Fine dining restaurant vs drive thru
Most of the times we reserve space at a fine dining restaurant by calling them up or reserving online. (The space is wasted if you do not show up even after reserving). The restaurant can also not accommodate more than its capacity. (Static memory)

Consider McDonalds drive-thru, you don't reserve space, you will somehow find a spot in the queue and just manage to grab your order. Here you are not reserving any seats beforehand, but still you can manage to get a meal. The restaurant can serve multiple customers even if they have not reserved. (Dynamic memory)

Dynamic memory allocation example.

```
int main()
{
   // Dynamic memory allocation
   int *ptr1 = new int;
   int *ptr2 = new int[10];
}
```

6. Freeing Memory

Once a programmer allocates memory dynamically as shown in the previous section, it is the responsibility of the programmer to delete/free the memory which is created. If not deleted/freed, even though the variable/array is still not used, the memory will continue to be occupied. This causes a **memory leak.**

To delete a variable **ptr1** allocated dynamically

```
delete ptr1;
```

To delete an array ptr2 allocated dynamically

```
delete [] ptr2;
```

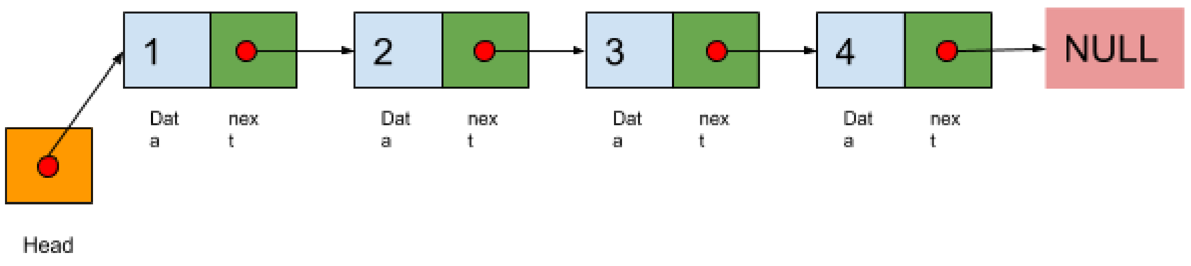
7. Introduction to Linked List

A linked list is a data structure that stores a list, where each element in the list points to the next element.

Let's elaborate:

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Node** − Each Node of a linked list can store data and a link.
- **Link** − Each Node of a linked list contains a link to the next Node (unit of Linked List), often called 'next' in code.
- **Linked List** − A Linked List contains a connection link from the first link called Head. Every Link after the head points to another Node (unit of Linked List). The last node points to NULL.



In code, each node of the linked list will be represented by a class or struct. These will, at a minimum, contain two pieces of information - the data that node contains, and a pointer to the next node. Example code to create these is below:

```
class Node
{
public:
        int data;
        Node *next;
};
```

```
struct node
{
   int data;
   Node *next;
};
```

8. Insertion in a linked list

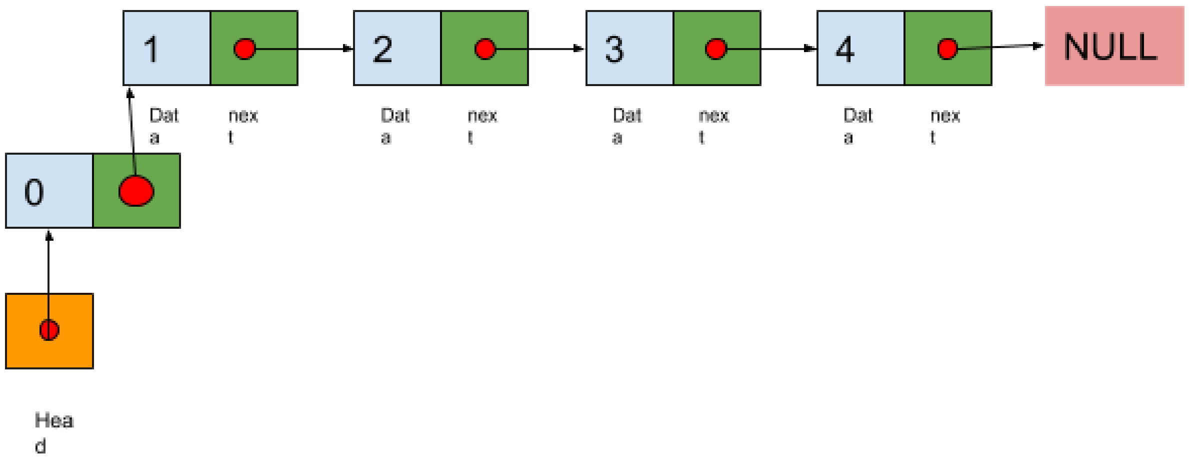
Adding a new node in a linked list is a multi-step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it must be inserted.

Scenarios in insertion

1. Insertion at the start.
2. Insertion at a given position.
3. Insertion at the end.

I) Inserting at the start of the Linked List.

Now we will insert an element at the start of the list.

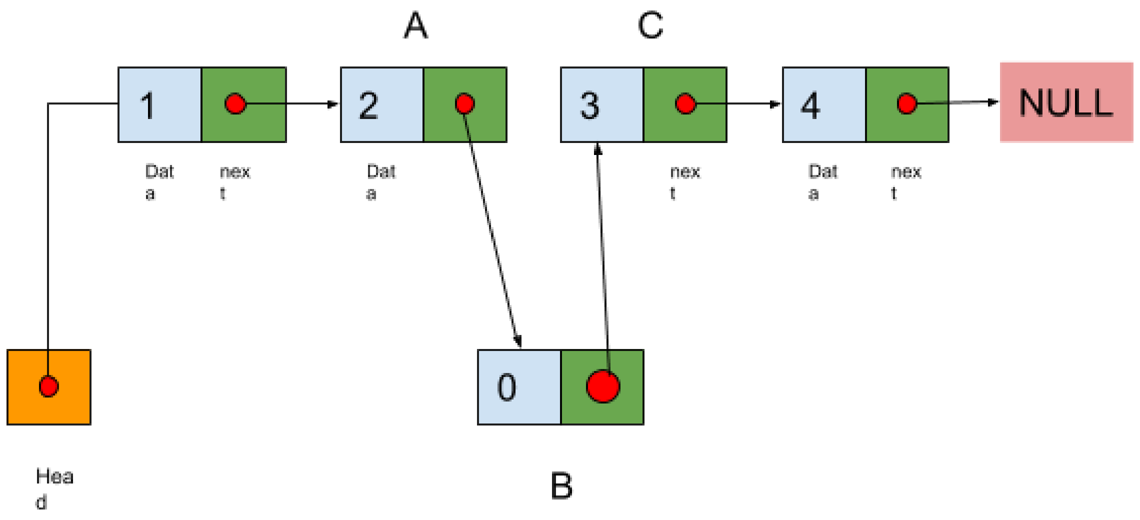


Create a new node,

a. Update the next pointer of that node to start of the list. (Value of the head pointer)
b. Update head pointer to new node.

II) Insertion at a given position in the Linked List

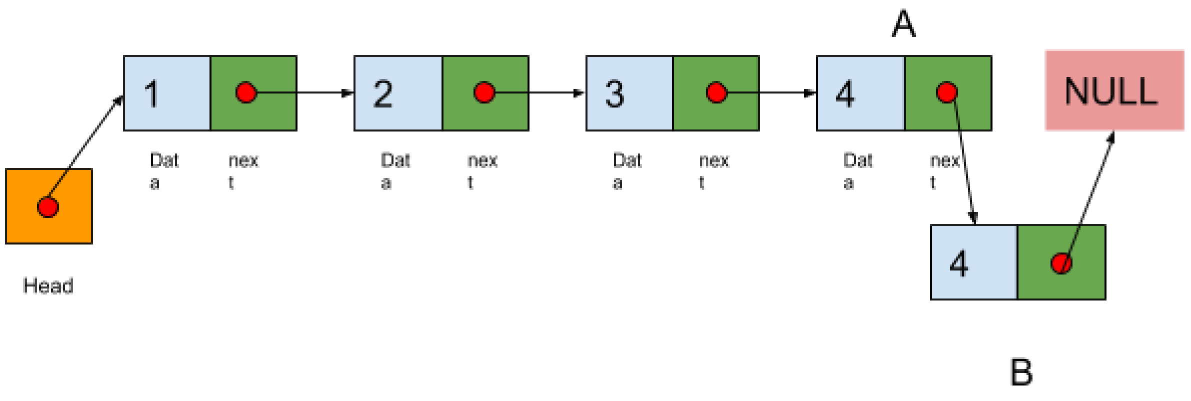
For example, let us insert a new node at position 2.



Create a new node (B).

a. Count and traverse until the node previous to given position i.e A.
b. Store the A’s next pointer value in a temporary variable.
c. Update the next pointer of A to the address of new node.
d. Copy the temporary variable’s value to B’s next pointer.
e. Now B’s next pointer points to the address of the node C.

III) Insertion at the end of a Linked List



Create a new node B.

a. Traverse till the node whose next pointer points to NULL. (A)
b. Update the next pointer of A to B's address.
c. Point B's next pointer to NULL.

9. Traversing a Linked List and printing elements in it

To print a list, we need to traverse through all the nodes in the list until we encounter the last node. When a node points to NULL we know that it is the last node.

```
node = root;
while ( node != NULL )
 {
    cout << node->value << endl;
    node = node->next;
}
```

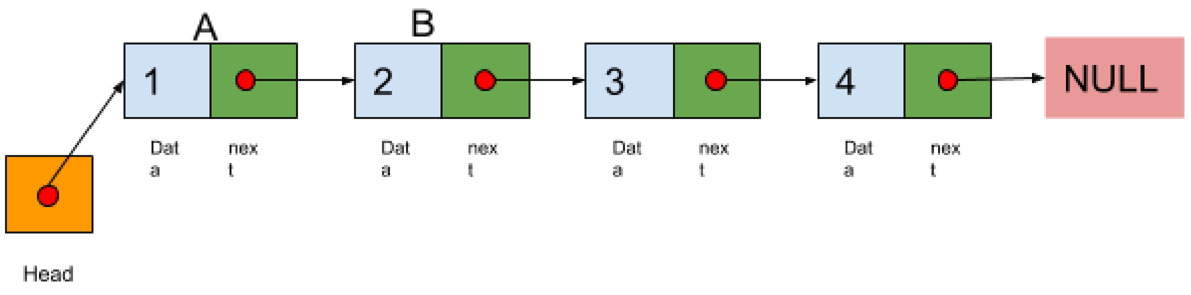
10. Deletion from a Linked List

Deleting a node in the linked list is a multi-step activity. Let's call the node to be deleted as 'A' and the node 'A' is pointing to as 'B'.

- First, the position of the node to be deleted ('A') must be found.
- The next step is to point the node pointing to 'A', to point to 'B'.
- The last step is to free the memory held by 'A'.

I) Deletion of the first node in Linked List

a. Given below is the linked list representation before the deletion of the first node



b. Steps followed to delete the first node ('A') having value '1'.

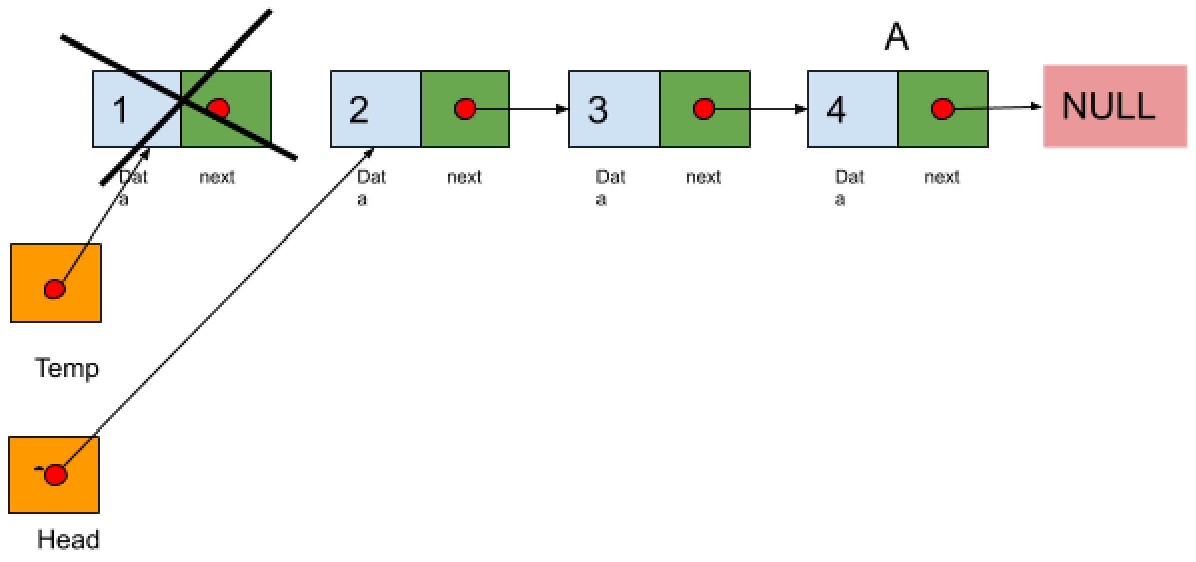
i. Create a variable **temp** having a reference copy of the head node.

ii. Point the head node from 'A' to 'B'

iii. Head is now pointing to 'B'. So, the Linked List's first element now is 'B' with the value'2'

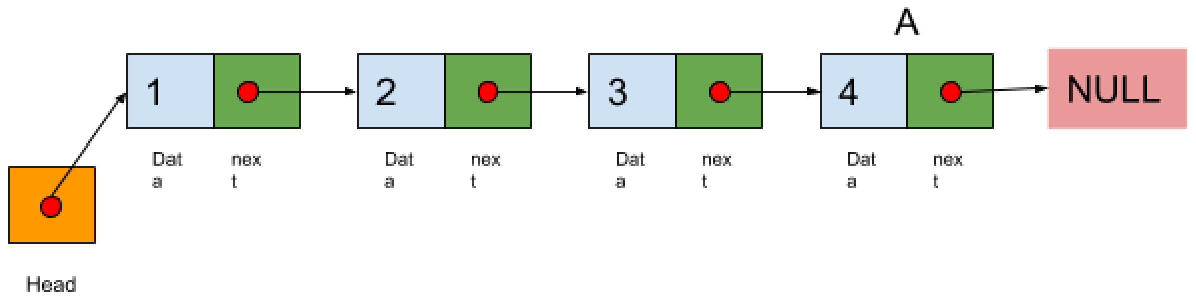
iv. Free the node 'A' pointed to by **temp**.

c. Linked list representation after deletion.



II) Deletion of the last node in Linked List

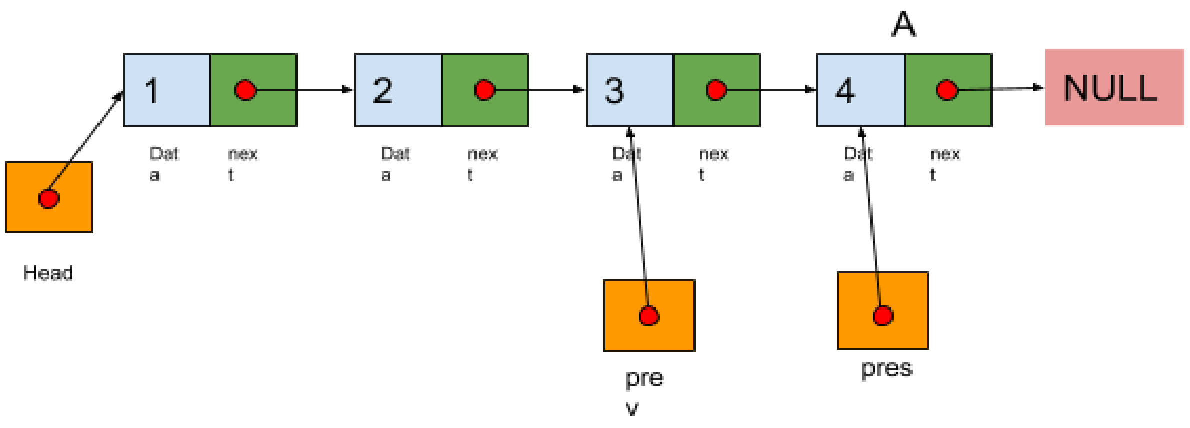
a. Given below is a linked list representation before deletion of the last node



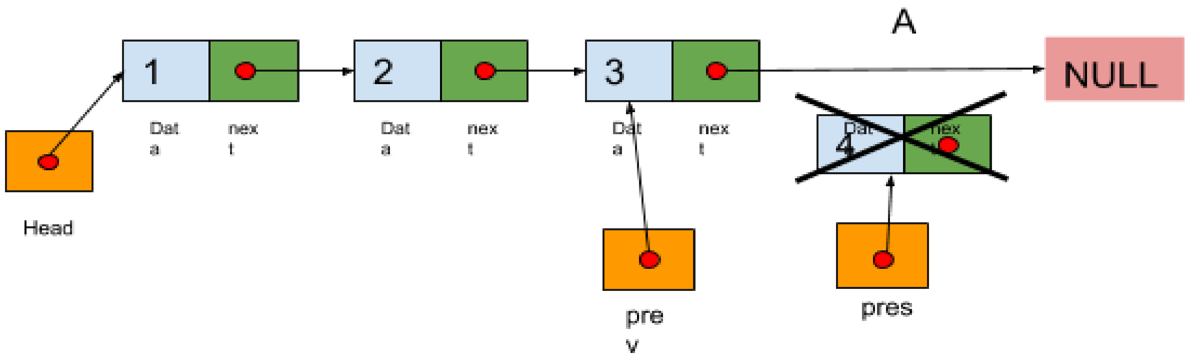
b. Steps followed to delete the last node (‘A’) having value ‘4’.

i. Create a variable **prev** having a reference copy of the head node.
ii. Create a variable **pres** having a reference copy of the next node after the head.
iii. Traverse the list until **pres** is pointing to the last node ‘A’.
   prev will be pointing to the second last node now.
iv. Make **prev** point to **NULL.**
v. Free the node ‘A’ pointed to by **pres.**

c. Linked list representation after **prev** and **pres** have completed traversing.



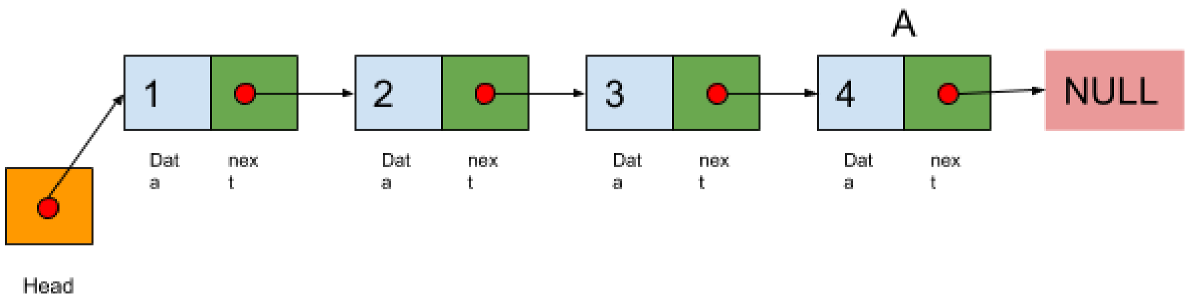
d. Linked list representation after deletion of the node ‘A’ and pointing prev to NULL.



III) Deletion of the complete linked list

The deletion of a linked list involves iteration over the complete linked list and deleting (freeing) every node in the linked list.

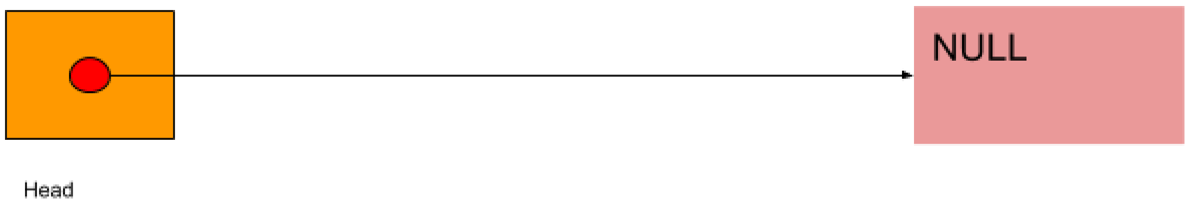
a. Given below is a linked list representation before deletion of the last node



b. Steps followed to delete every node in the linked list.

i. Create a variable **prev** having a reference copy of the head node.
ii. Create a variable **pres** having a reference copy of the next node after the head.
iii. While traversing the list, at each step delete/free the memory pointed to by **prev.**
iv. Now, point **prev** to the **pres** and point **pres** to the next node after **pres (ie. pres->next)**.
v. Traverse the list until **pres** is pointing to the **NULL**.
prev will be pointing to the second last node now.
vi. Free the memory pointed to by **prev.** Now every element in the linked list is deleted/freed.

c. Linked list representation after deletion of all the nodes.



11. Exercise

Open your exercise file and complete the TODOs in the C++ program which does the following:
1. It will read from a text file. Each line of the file contains a number. Provide the file name as a command line argument. The number of lines in the file is not known.
2. Create an array dynamically of a capacity (say 10) and store each number as you read from the file.
3. If you exhaust the array but haven't reached the end of the file yet, then dynamically resize/double the array and keep on adding elements to it.