LINKED LIST

- Given an ordered-list:
  - What happens when we add/delete an element to the list?

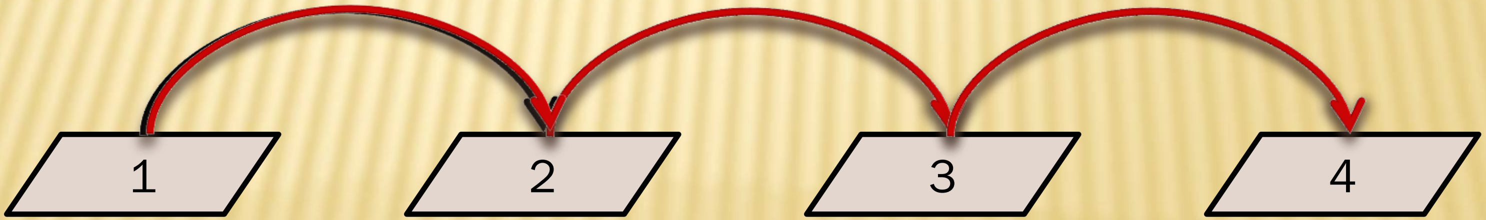
- **In the array implementation, such cases require heavy data movement which is very costly.**

- In most cases, the data is processed in a strictly sequential order.

- Therefore all that is required is the access to **next element.**

- In our previous mapping, the next element happens to be the elemental at the next index in the array.
- If we can some how tell where the next element is stored, we can eliminate this restriction of placing the next element at the next physical location!
- This gives rise to the notion of **logical adjacency** as opposed to **physical adjacency**.

- In that case, in order to access the elements of the list, all we need is a starting point and some mechanism to move from one element to the next.
- In other words, we need a **starting point** and a **link from one element to the next**.



- **We can only travel in the direction of the link.**

- We have a **chain** like structure and we can access the elements in the chain by just following the links in the chain.

- Such an organization is known as a **linked-list.**

CHAINING & LINK LIST

LINKED-LISTS

- This organization rids us from the requirement to maintain the logical as well as physical adjacency.
- Now all we need to maintain is the logical sequences and **two logically adjacent elements need to not be physically next to each other.**

TRAIN & LINK LIST

Engine

Bogies

Starting point of linklist

root → 5 → 8 → 10 → 15 → ∅

data elements

BOGIE

LINKED-LISTS

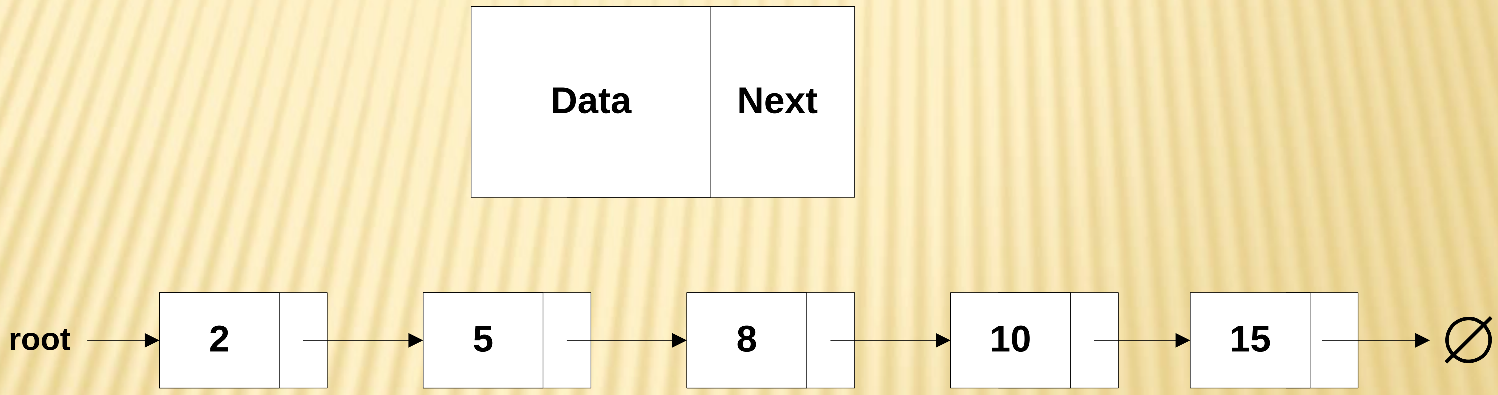
- We can implement this organization by storing the information and handle to the next element with the other data as follows:-

```
struct node {
   node * next;                    // handle to next list element
   data_type  data;
};

node * root;            // starting point - handle to first list element
root = ϕ                // initialize head to ϕ - the end marker
```

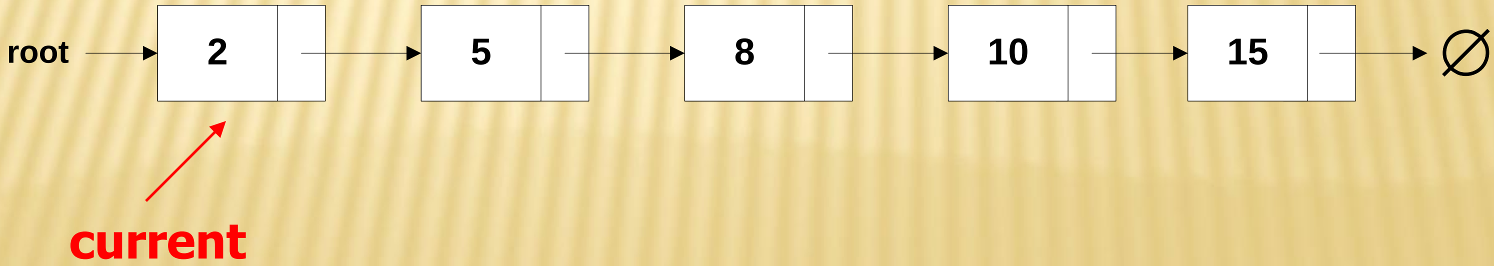
LINKED-LISTS

- A list node and the resulting list looks like as shown below:



LINKED-LISTS – TRAVERSAL

```
void traverse (node * root)
{
    node * current = root;
   while ( current != ϕ ) {
        process data at current;
        current = next field of current element
   }
}
```

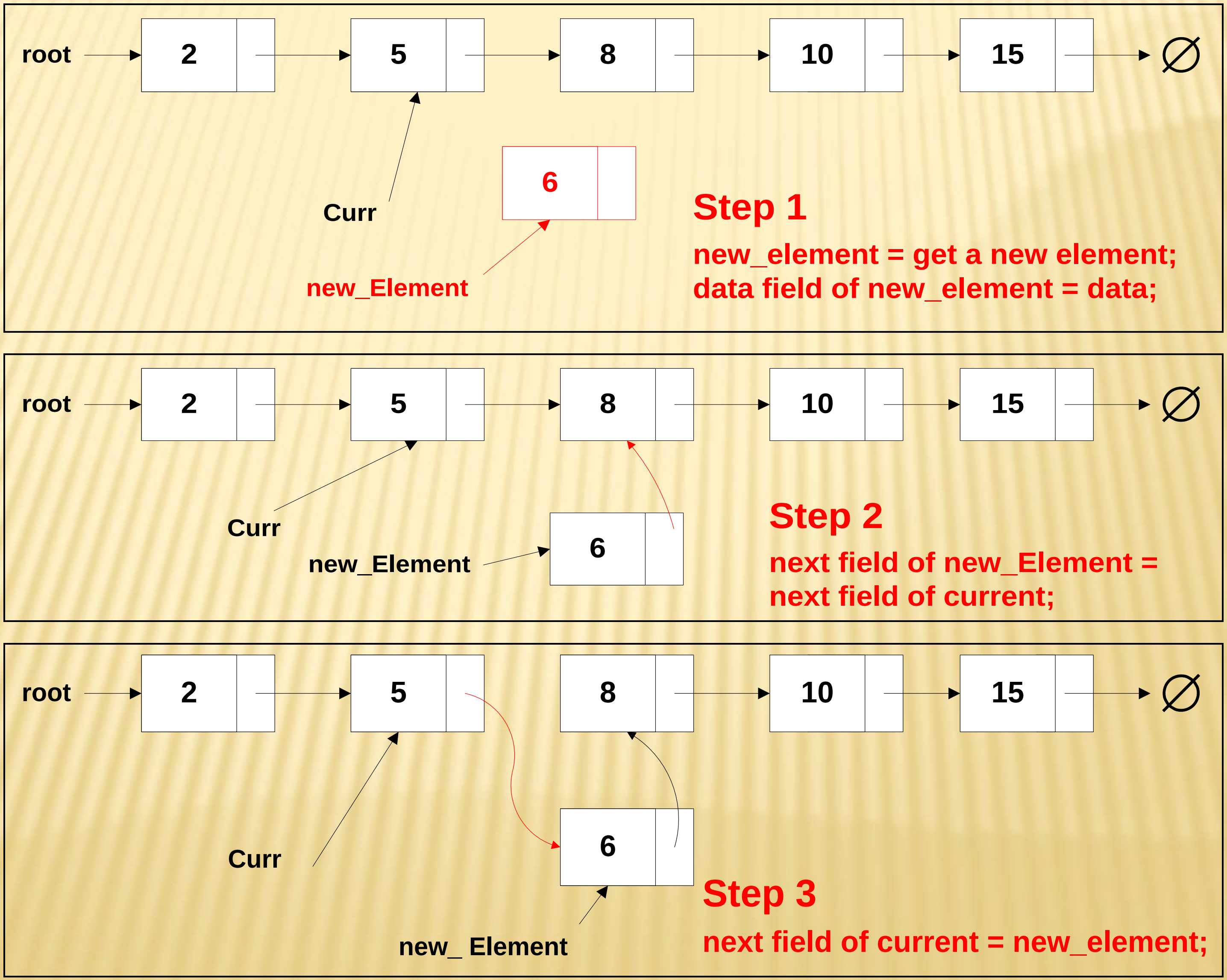


LINKED-LIST
ADD AN ELEMENT AFTER CURRENT ELEMENT

```
void add (node * current, data_type data)
{
    node  * new_element;
    new_element = get a new element;                           //step 1
    data field of new_element = data;                          //step 1

    next field of new_element = next field of current;         //step 2
    next field of current = new_element;                       //step 3
}
```

LINKED-LIST – INSERTION



LINKED-LIST – WARNING: FRAGILE, HANDLE LINKS WITH CARE

- If one link is broken, you loose access to all subsequent elements.
- What happens if you swap the following two lines in add:

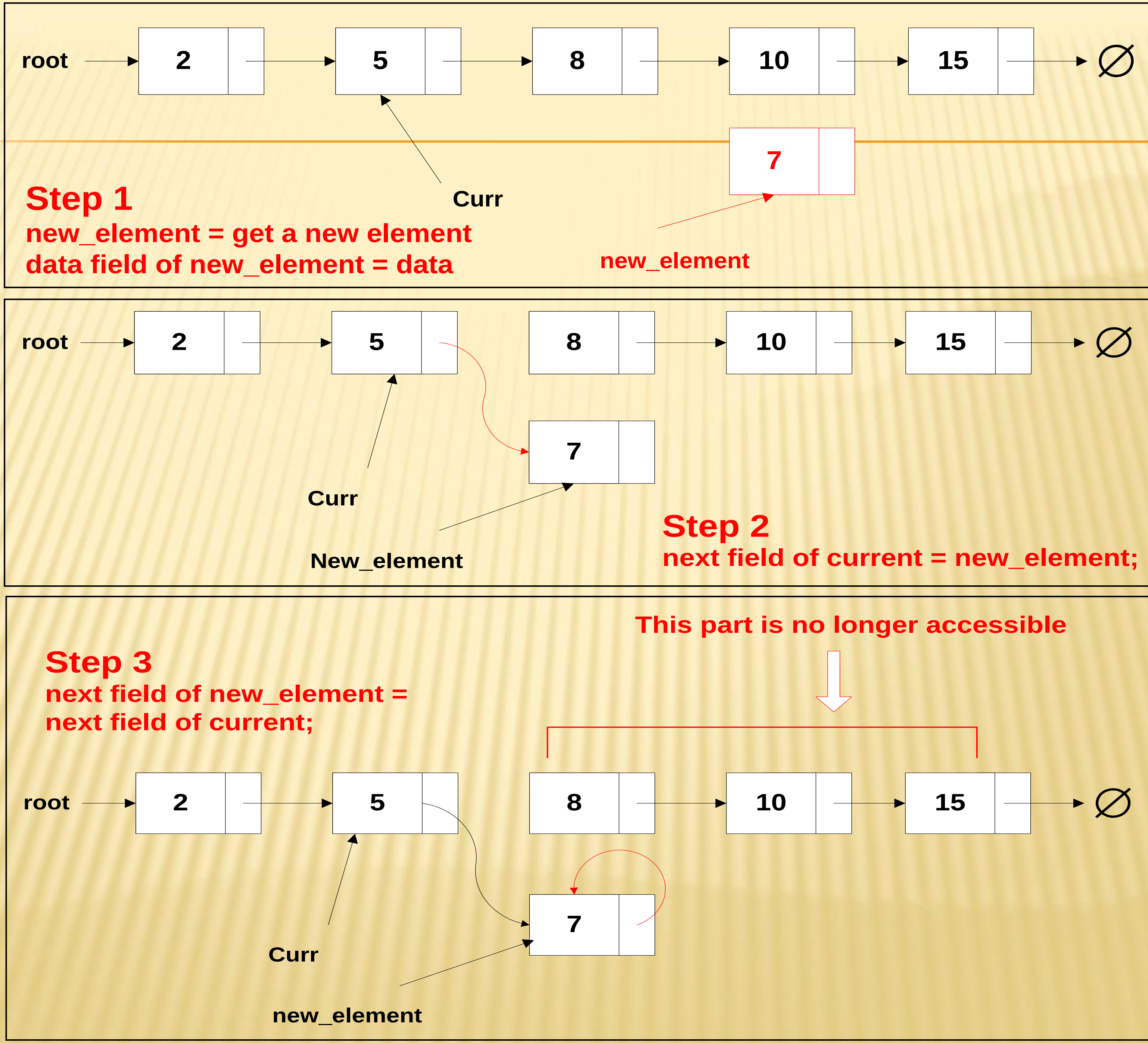
  next field of new_element = next field of current; //step 2

  next field of current = new_element; //step 3

to

  next field of current = new_element; //step 2

  next field of new_element = next field of current; //step 3

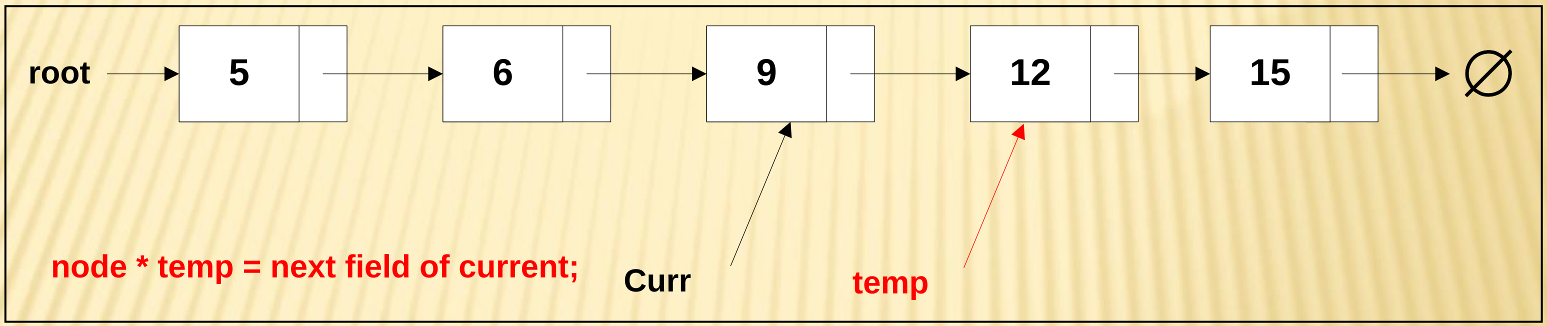


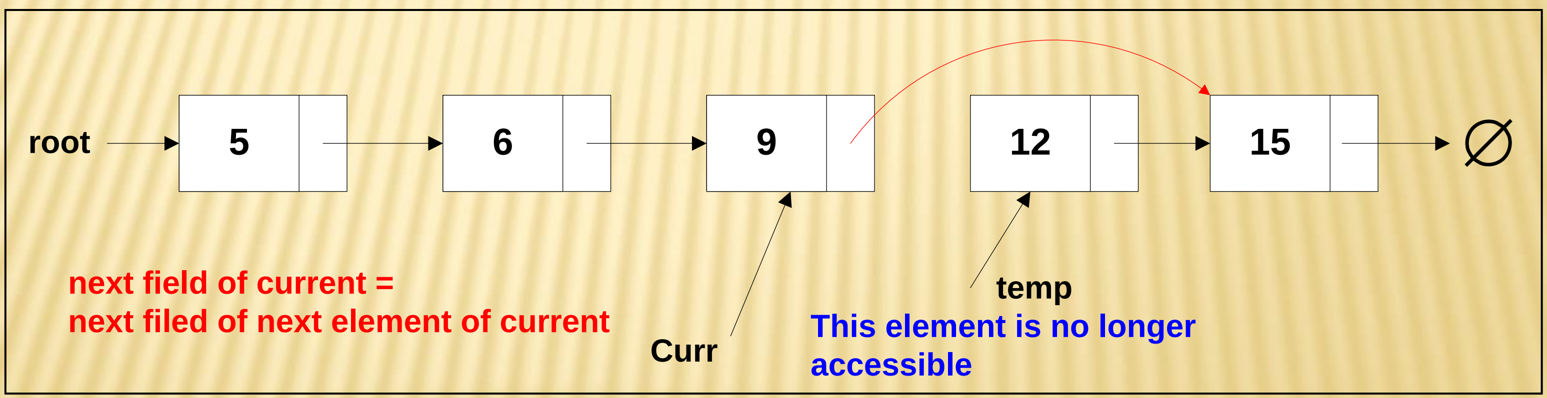
root
2
5
8
10
15
7
Curr
new_element
Step 1
new_element = get a new element
data field of new_element = data
root
2
5
8
10
15
7
Curr
New_element
Step 2
next field of current = new_element;
This part is no longer accessible
Step 3
next field of new_element =
next field of current;
root
2
5
8
10
15
7
Curr
new_element

LINKED-LIST
DELETE AN ELEMENT AFTER CURRENT ELEMENT

```
void delete (node * current)
{
     node * temp = next field of current;
    next field of current = next field of next element of current
                                              // same as next field of temp
    discard temp
}
```

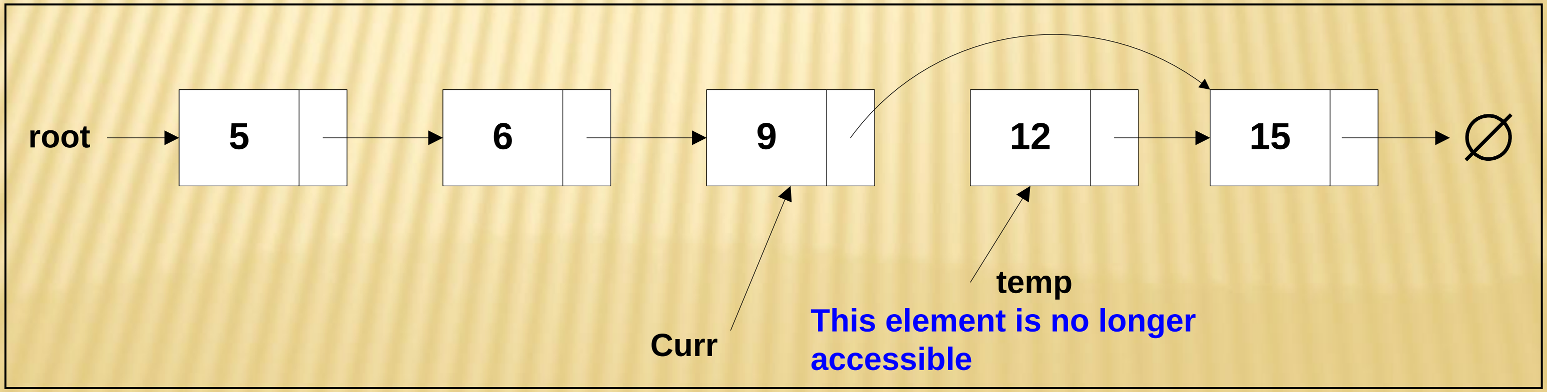
LINKED LIST - DELETION



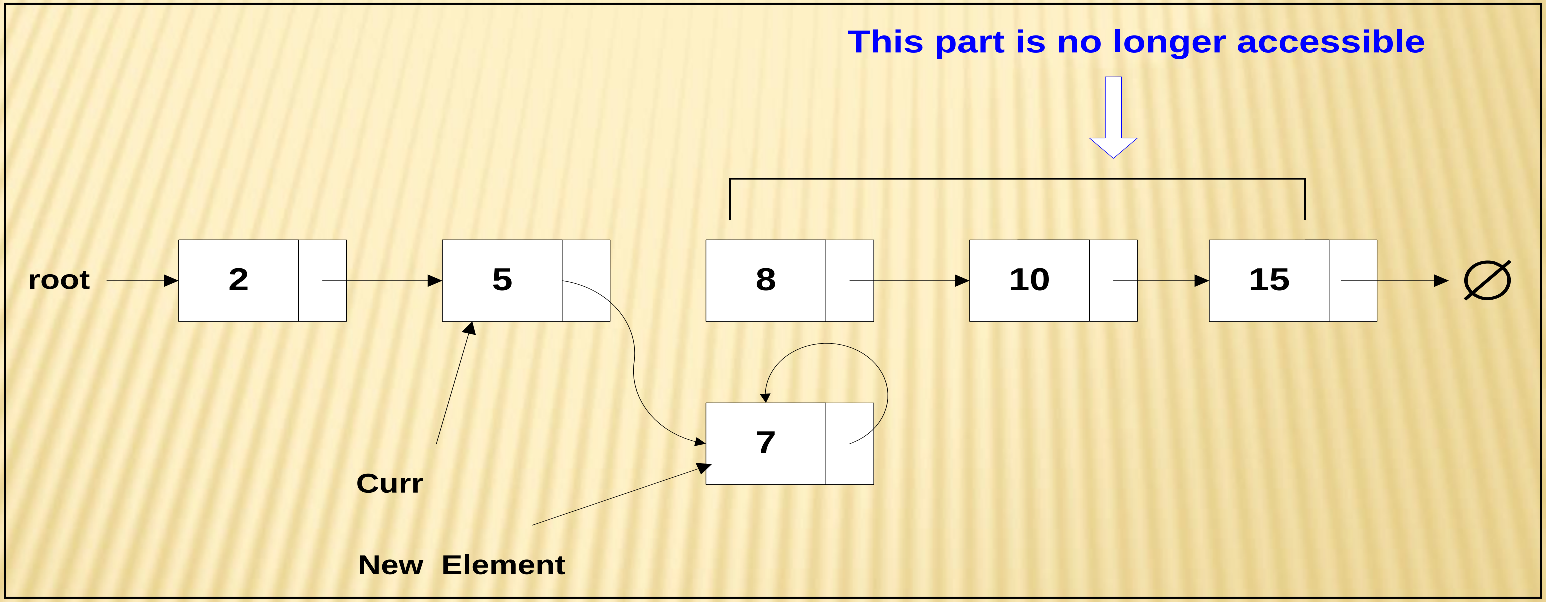


GARBAGE

- When all access paths to a data object are destroyed but the data object continues to exist, the data object is said to be garbage.
- Garbage is a less serious but still troublesome problem. A data object that has become garbage ties up storage that might otherwise be reallocated for another purpose.
- A buildup of garbage can force the program to terminate prematurely because of lack of available free storage.



GARBAGE


This part is no longer accessible
root
2
5
8
10
15
7
Curr
New Element

LINKED LIST - IMPLEMENTATION USING DYNAMICALLY ALLOCATED STORAGE

- Handle will be the pointer (memory address) of an element in the linked list.
- Space may be acquired at run-time on request from *free store* using *new.*
- At a point in time when a memory area pointed to by a pointer is no longer required, it can be returned back to *free store* using *delete*.
- We should always return storage after we no longer need it.
- Once an area is freed, it is improper and quite dangerous to use it.

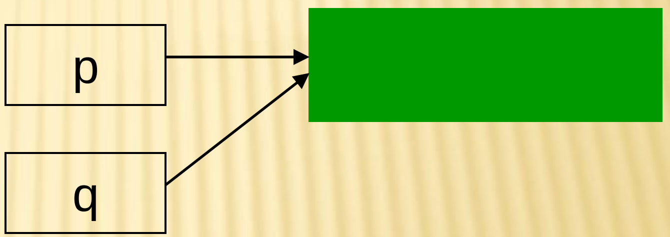
DANGLING REFERENCES

A dangling reference is an access path that continues to exist after the life time of the associated data object.

ObjectTypeA *p, *q;

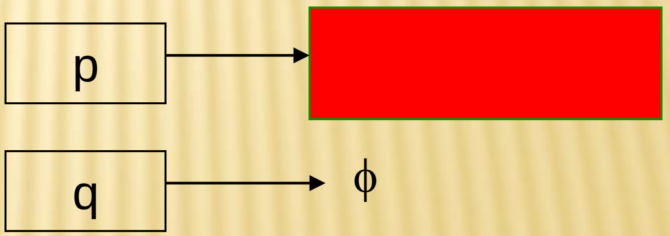
p = new ObjectTypeA;

q = p;



delete q; q = NULL;

What happens to p?

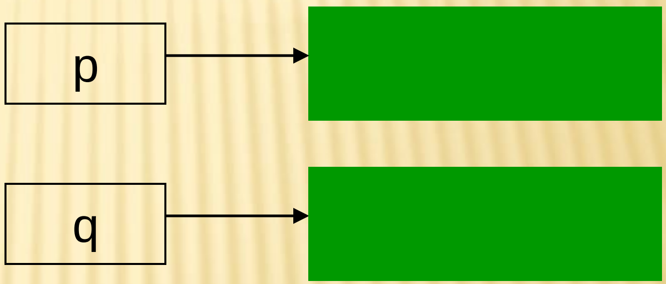


It’s a “dangling reference”!

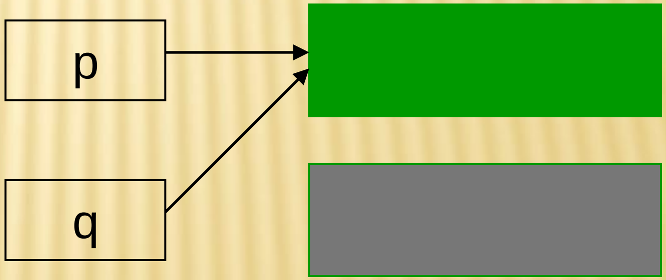
GARBAGE

When all access paths to a data object are destroyed but the data object continues to exist, the data object is said to be garbage.

```
ObjectTypeA *p, *q;
p = new ObjectTypeA;
q = new ObjectTypeA;
```




```
q = p;
```



What happens to the space that was pointed to by q?

It's "garbage"!

DANGLING REFERENCES AND GARBAGE

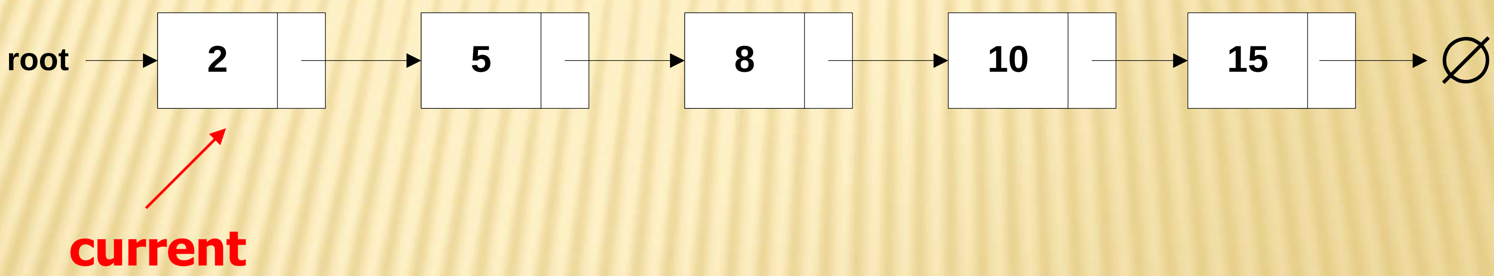
- Dangling references are particularly serious problem for storage management, as they might compromise the integrity of the entire run-time structure during program execution.
- Garbage is a less serious but still troublesome problem. A data object that has become garbage ties up storage that might otherwise be reallocated for another purpose.
- A buildup of garbage can force the program to terminate prematurely because of lack of available free storage.

LINKED LIST - IMPLEMENTATION USING DYNAMICALLY ALLOCATED STORAGE

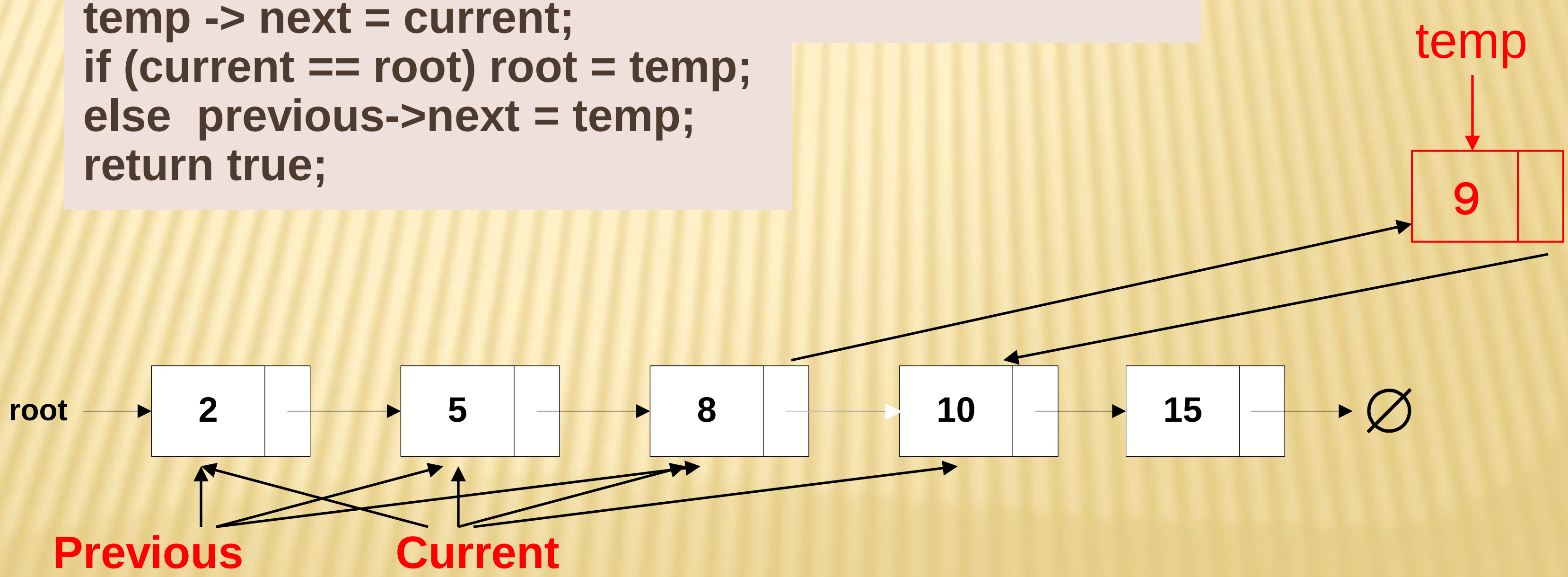
```
struct Node {
    int         data;
    Node        *next;
};

Class LinkedList {
    private:
        Node * root;
    public:
        LinkedList() { root = NULL; }
        void add (int data);
        void remove (int data);
        void print ();
        ~LinkedList();
};
```

```
void LinkedList:: print()
{
    Node *current = root;
    while (current != NULL) {
        cout << current->data;
        current = current->next;
}
```



```
bool LinkedList::add(int data)
{
    Node *temp = new Node;
    if (temp == NULL) return false;
    else {
        temp->data = data;
        Node *current, *previous;
        current = root;
        while (current != NULL && current->data < data;) {
                previous = current;
                current = current->next;
        }
        temp -> next = current;
        if (current == root) root = temp;
        else  previous->next = temp;
        return true;
    }
}
```



```
void LinkedList::remove(int data)
{
    Node *current, *previous;
    current = root;
    while (current != NULL && current->data < data) {
        previous = current;
        current = current->next
    }
    if (current != NULL && current->data == data) {
        if (current == root)
            root = root -> next;
        else
            previous->next = current->next;
        delete  current;
    }
}
```

THE DESTRUCTOR

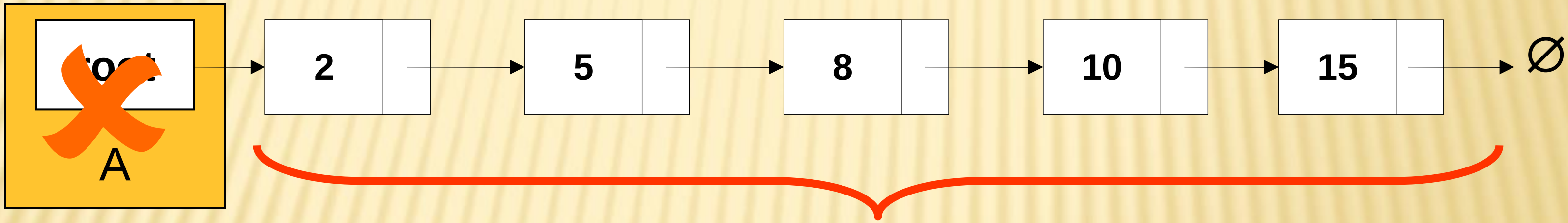
- The destructor is the counterpart of the constructor.
- It is a member function that is called automatically when a class object goes out of scope.
- Its purpose is to perform any cleanup work necessary before an object is destroyed.
- Destructors are required for more complicated classes, where they're used to release dynamically allocated memory.

~LINKEDLIST() { }

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

Class LinkedList {private: Node * root; public: … ~LinkedList(); };

LinkedList A; A.add(2); A.add(5); A.add(10); A.add(15); A.add(8);



- A goes out of scope. What Happens?
- When destructor is called, what is deleted?
- What Happens to this memory area?
- It is garbage!
- Destructor must also delete this area.

~LINKEDLIST()

```
~LinkedList()
{
    Node *temp1, *temp2;
    temp1 = root;
    while (temp1 != NULL)
    {
        temp2 = temp1->next;
        delete temp1;
        temp1 = temp2;
    }
}
```