Module 8B.

Dynamic Memory Management & Linked List

COMP2113 Programming Technologies / ENGG1340 Computer Programming II

Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong

We are going to learn…

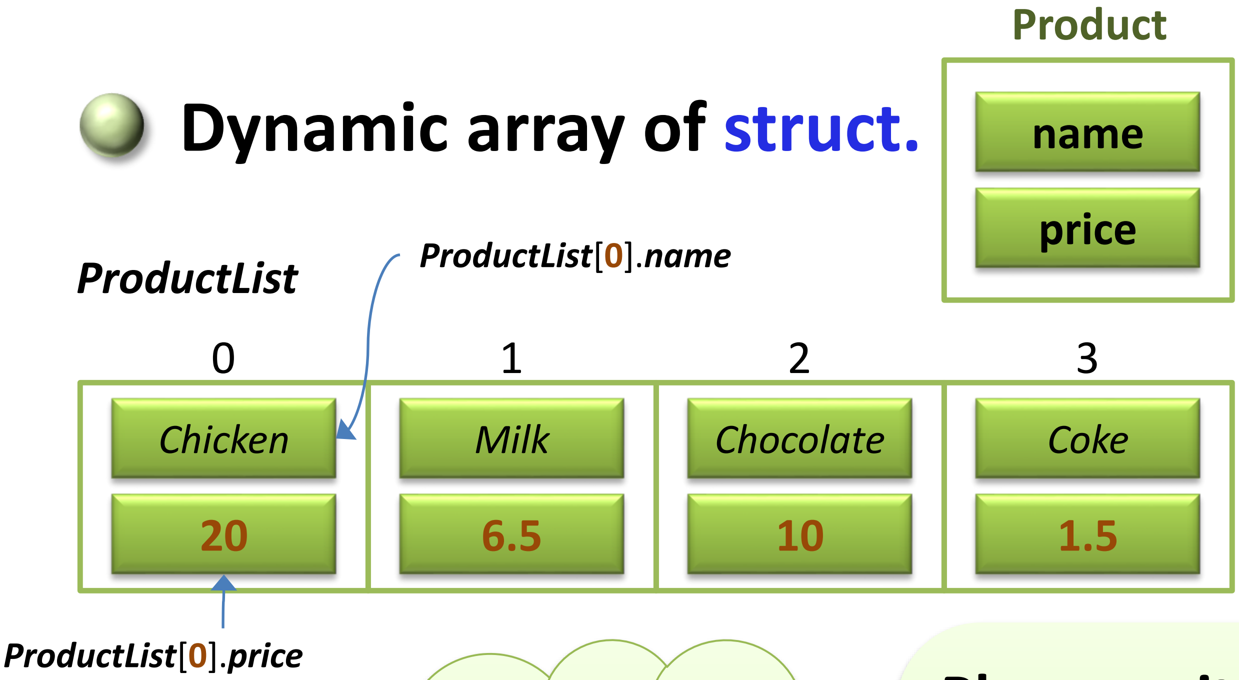
- **Dynamic arrays**
- **Linked lists**
  - Searching
  - Insertion
  - Deletion

Dynamic array and **linked lists** are very important **data structures** in Computer Programming!
They are used to **organize a collection of items**.
(E.g., product list, student list, …etc)

Dynamic array

Dynamic array of struct

- Dynamic array of struct.



Product

name

price

ProductList

ProductList[0].name

0	1	2	3
Chicken	Milk	Chocolate	Coke
20	6.5	10	1.5

ProductList[0].price

```
struct Product{
	string name;
	double price;
};
```

```
Product *ProductList;
ProductList = new Product[4];
```

Option 1:
One possible solution is to use an array of **Products!**

Please write a program that stores a list of products, and allows :

1. **Product search.**
2. **Product insertion.**
3. **Product deletion.**

“nothrow” for dynamic array

When we create a dynamic array with big size such as:

```
Student * S = new Student[10000];
```

there can be a case that the main memory cannot allocate enough space to fulfill our need. Under normal situation, the program will throw an exception (error).

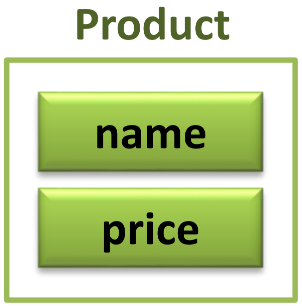
However, if you create the same array as follows:

```
Student * S = new (nothrow) Student[10000];
```

the program will not throw exception but just assign NULL to S.

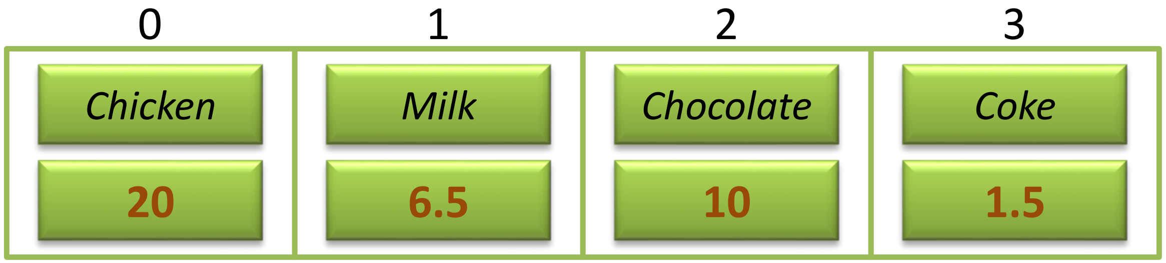
Dynamic array of struct

- Dynamic array of struct.



```
struct Product{
        string name;
        double price;
};
```

ProductList



0	1	2	3
Chicken	*Milk*	*Chocolate*	*Coke*
20	6.5	10	1.5

```
Product *ProductList;
ProductList = new Product[4];
```

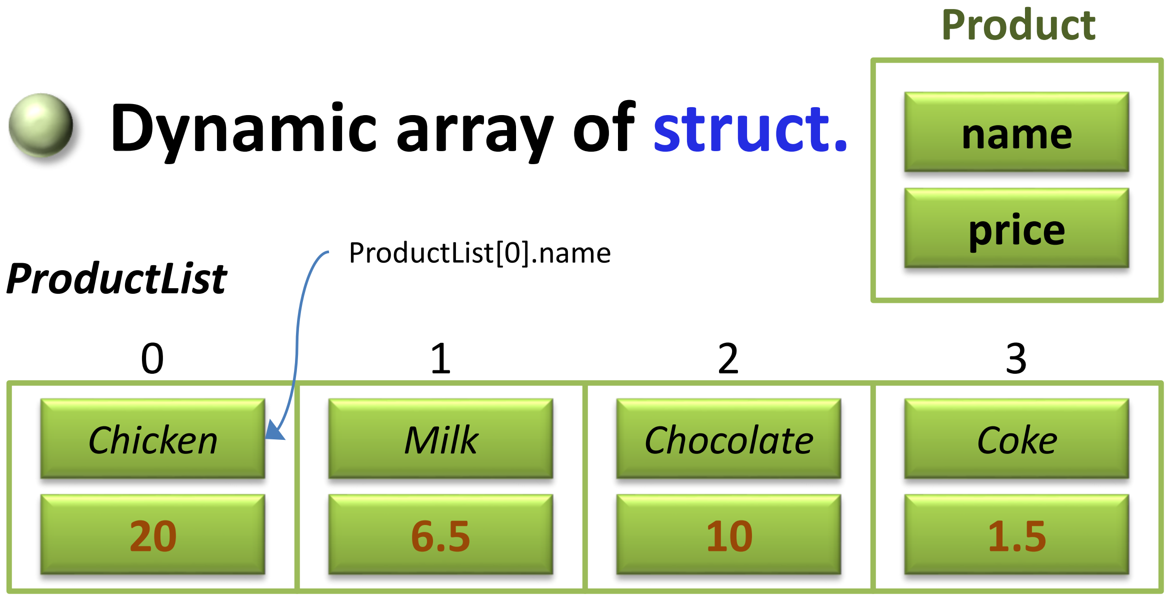
- Searching

```
void search (Product *ProductList, string searchName){


}
```

Actually, we are passing **the pointer to the first slot of the dynamic array** ***ProductList*** into the function.

Dynamic array of struct

- **Dynamic array of struct.**



```
struct Product{
        string name;
        double price;
};
```

```
Product *ProductList;
ProductList = new Product[4];
```

- **Searching**

Inside the function, **we can simply treat the dynamic array as ordinary array**.

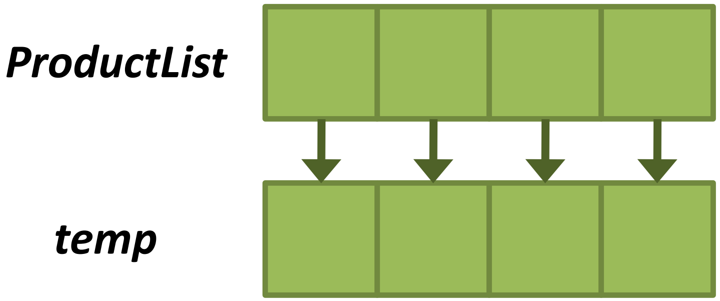
```
void search (Product *ProductList, string searchName){
   for ( int i = 0 ; i < 4 ; i++ ){
      if ( ProductList[i].name  == searchName )
         cout << ProductList[i].price  << endl;
   }
}
```

Dynamic array of struct

- **Insertion**

1. Create a temporary array

- Create a temporary array called ***temp*** and copy all current content of ***ProductList*** to this temporary array.



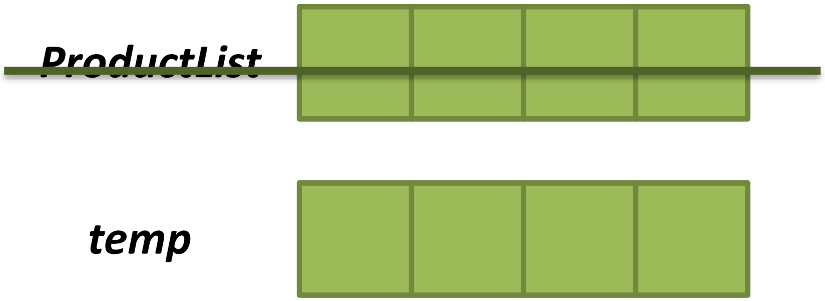
```
int main () {
...
   int size = 4;
   Product *temp;
   temp = new Product[size];
   for (int i =0 ; i < size ; i++){
      temp[i] = ProductList[i];
   }
...
   return 0;
}
```

Dynamic array of struct

Insertion

2. Delete *ProductList*

- Delete the ***ProductList*** dynamic array and free all the memory occupied by ***ProductList***.



```
int main () {
…
   int size = 4;
   Product *temp;
   temp = new Product[size];
   for (int i =0 ; i < size ; i++){
      temp[i] = ProductList[i];
   }
   delete [] ProductList;

…
   return 0;
}
```

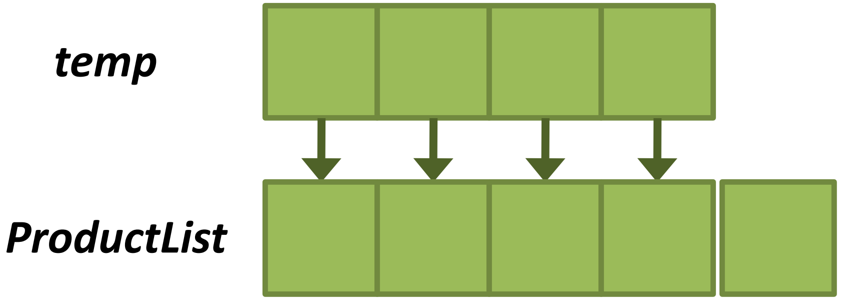
Extend a dynamic array

Dynamic array of struct

- Insertion

3. Re-create larger *ProductList*

- Re-create ***ProductList*** with more slots (in this case, one more slot).
- Copy back the content from ***temp*** to ***ProductList.***



```
int main () {
…
   int size = 4;
   Product *temp;
   temp = new Product[size];
   for (int i =0 ; i < size ; i++){
      temp[i] = ProductList[i];
   }
   delete [] ProductList;

   ProductList = new Product [size+1];

   for (int i =0 ; i < size ; i++){
      ProductList[i] = temp[i];
   }
…
   return 0;
}
```

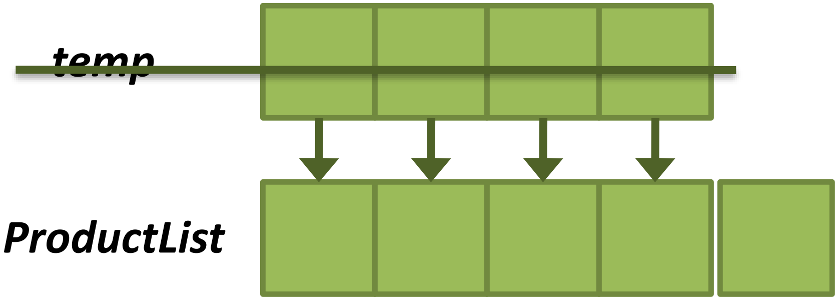
Extend a dynamic array

Dynamic array of struct

- Insertion

4. Remember to free *temp*

- As ***temp*** is not used anymore, delete ***temp*** and free the memory.



```
int main () {
...
    int size = 4;
    Product *temp;
    temp = new Product[size];
    for (int i =0 ; i < size ; i++){
        temp[i] = ProductList[i];
    }
    delete [] ProductList;

    ProductList = new Product [size+1];

    for (int i =0 ; i < size ; i++){
        ProductList[i] = temp[i];
    }
    delete [] temp;

...
    return 0;
}
```

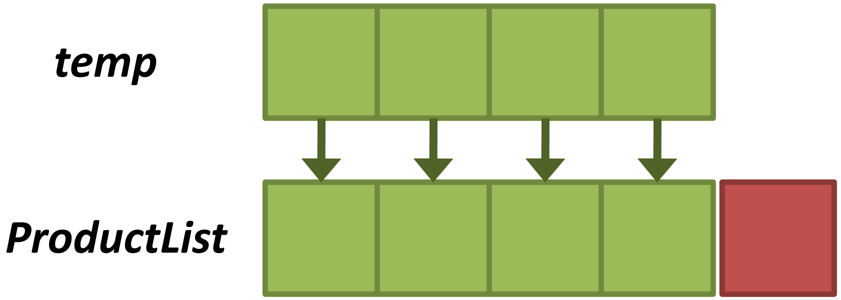
Extend a dynamic array

Dynamic array of struct

- Insertion

5. Add the new product

- Now the dynamic array ***ProductList*** has enough slots to accommodate the new product.



```
int main () {
...
    int size = 4;
    Product *temp;
    temp = new Product[size];
    for (int i =0 ; i < size ; i++){
        temp[i] = ProductList[i];
    }
    delete [] ProductList;

    ProductList = new Product [size+1];

    for (int i =0 ; i < size ; i++){
        ProductList[i] = temp[i];
    }
     delete [] temp;

    ProductList[size].name = "Candy";
    ProductList[size].price = 2.5;
    size++;
...
    return 0;
}
```

Extend a dynamic array

Dynamic array of struct

Insertion

Summary of operations:

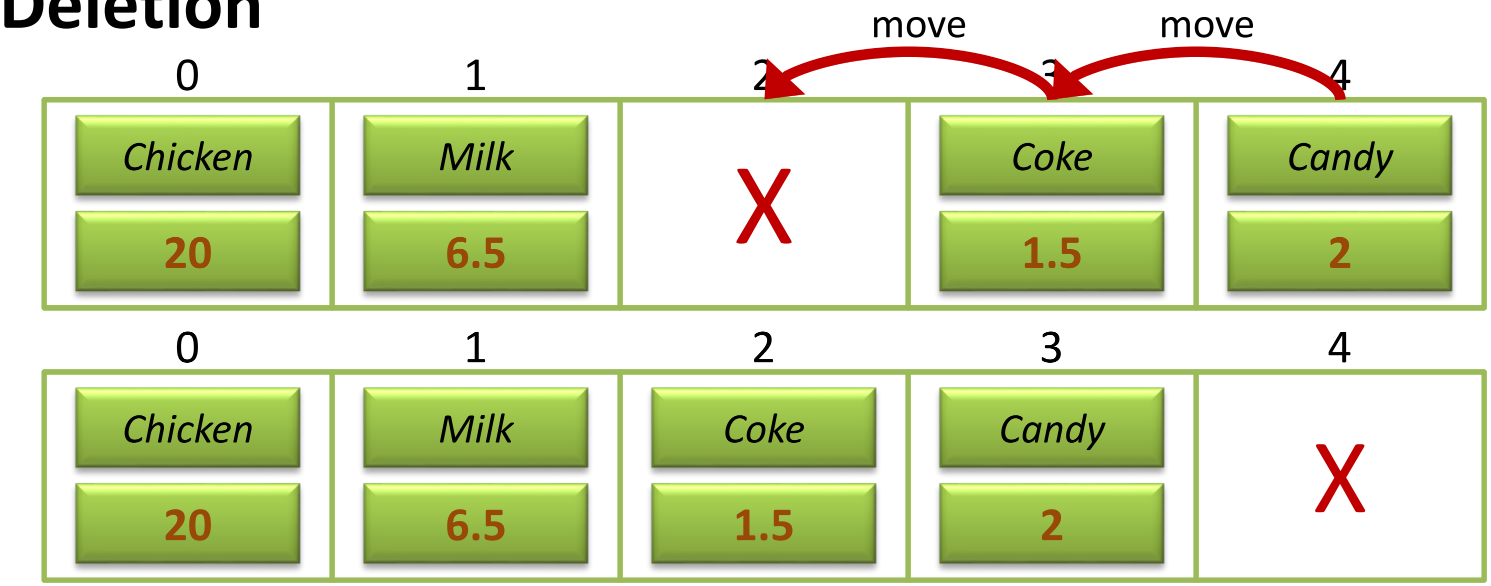
1. Create a temporary array with the same size as the old array
2. Copy elements from the old array to the temporary array
3. Delete the old array
4. Create a new array with old array's name and with size (size of old array + 1)
5. Copy elements from the temporary array to the new array
6. Delete the temporary array
7. Add the new element

```
int main () {
…
   int size = 4;
   Product *temp;   // Step 1
   temp = new Product[size];
   for (int i =0 ; i < size ; i++){   // Step 2
      temp[i] = ProductList[i];
   }
   delete [] ProductList;   // Step 3

   ProductList = new Product [size+1];   // Step 4

   for (int i =0 ; i < size ; i++){   // Step 5
      ProductList[i] = temp[i];
   }
    delete [] temp;   // Step 6

   ProductList[size].name = "Candy";   // Step 7
   ProductList[size].price = 2.5;
   size++;
…
   return 0;
}
```

Extend a dynamic array

Dynamic array of struct

Deletion



move move

0	1	2	3	4
Chicken	Milk	X	Coke	Candy
20	6.5		1.5	2

0	1	2	3	4
Chicken	Milk	Coke	Candy	X
20	6.5	1.5	2	

There are many ways to delete a product, **one way is to move the subsequent slots to the left by one slot.**

```
void delete_product (Product *ProductList, int slotID, int size){
    for (int i = slotID ; i < size-1 ; i++){
        ProductList[i] = ProductList[i+1];
    }
}
```

Delete a product

Dynamic array of struct

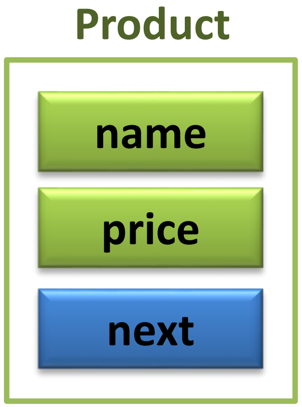
- **Weaknesses of using array to store the list of items.**
  - Need to initialize the initial number of array slots.
  - Deletion of an item need to move a lot of other items (those items after the slot that contains the entry to be deleted).
- **Alternative way to store a list of items:**
  - **Linked list**

Linked list

Linked list

- A fundamental data structure.
- Allow insertion and removal of items (struct) at any point in the list in constant time (efficient).
- Make use of the concepts of structure (struct), pointers (*ptr) and dynamic variable allocation (new).

In the struct, we add one more pointer variable ***next*** that is used to store the address of the next struct variable.



Product

name

price

next

```
struct Product{
        string name;
        double price;
        Product *next;
};
```

Analogy

- A secret organization has 6 members (M1, M2, M3, M4, M5, M6).
- For safety purpose, there does not exist a list containing addresses of all members.
- Each member only keeps the address of one other member. The head of the organization (H) also keeps the address of one member.

Member:	Keeping address of member:
H	M1
M1	M2
M2	M3
M3	M4
M4	M5
M5	M6
M6	Nil

Challenge: How can one visit all members in the organization?

Answer: Visit H to get address of M1, visit M1 to get address of M2, etc.

Analogy

Challenge: What if a new member M7 wants to join the organization?

Answer: Create a new member record and let M6 keeps its address

Member:	Keeping address of member:
H	M1
M1	M2
M2	M3
M3	M4
M4	M5
M5	M6
M6	~~Nil~~ **M7**
M7	**Nil**

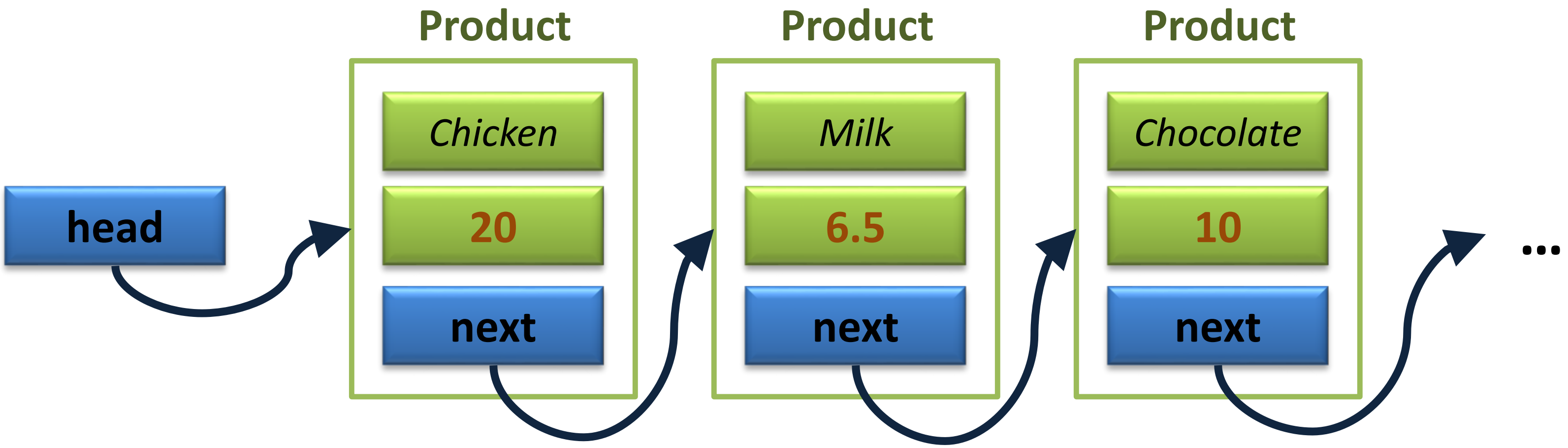
Analogy

Challenge: What if member M3 wants to leave the organization?

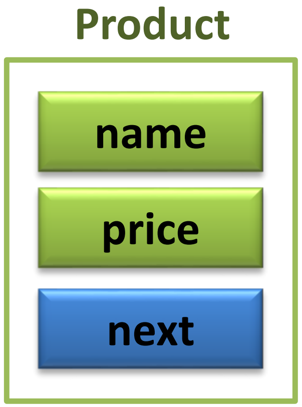
Answer: Pass the address kept by M3 originally to M2.

Member:	Keeping address of member:
H	M1
M1	M2
M2	~~M3~~ M4
~~M3~~	~~M4~~
M4	M5
M5	M6
M6	M7
M7	Nil

Linked list



In the **struct**, we add one more pointer variable ***next*** that is used to store the address of the next **struct** variable.



```
struct Product{
        string name;
        double price;
        Product *next;
};
```

1. Declaration

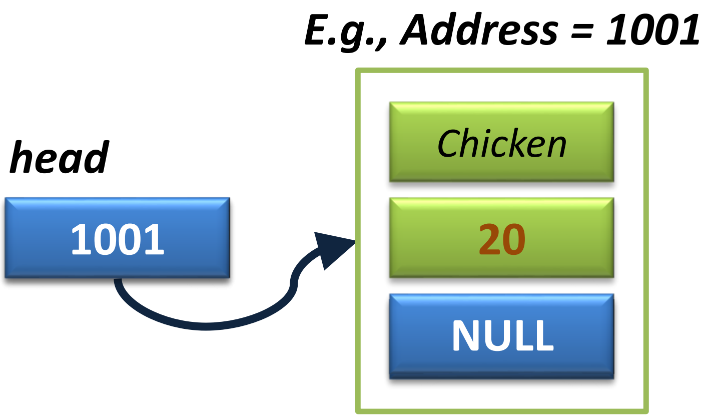
head

NULL

```
Product *head = NULL;
```

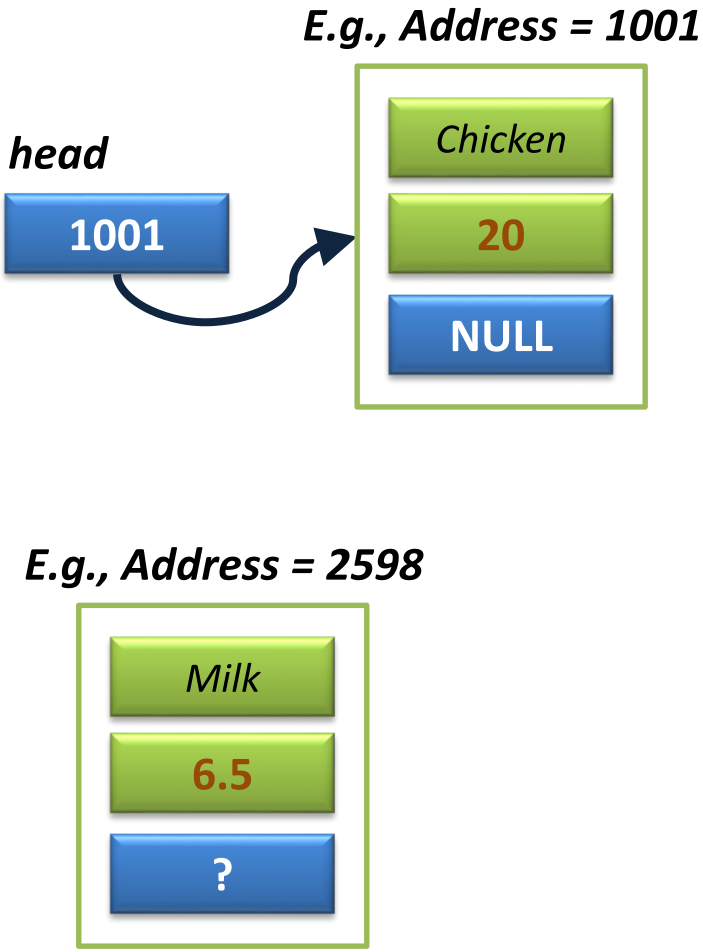
- In order to access a linked list, a pointer variable often called **head** is used to store a pointer to the first node of the list.
- Initially when the linked list is empty (i.e., a linked list with no nodes), head will simply contain a **NULL** pointer.

2. Insertion



- Starting from an empty list, new nodes may be created and inserted into the linked list.
- A **NULL** pointer is assigned the pointer variable of the last node to indicate the end of the linked list.
- Node insertion **may take place at the head of the linked list** (Why?).

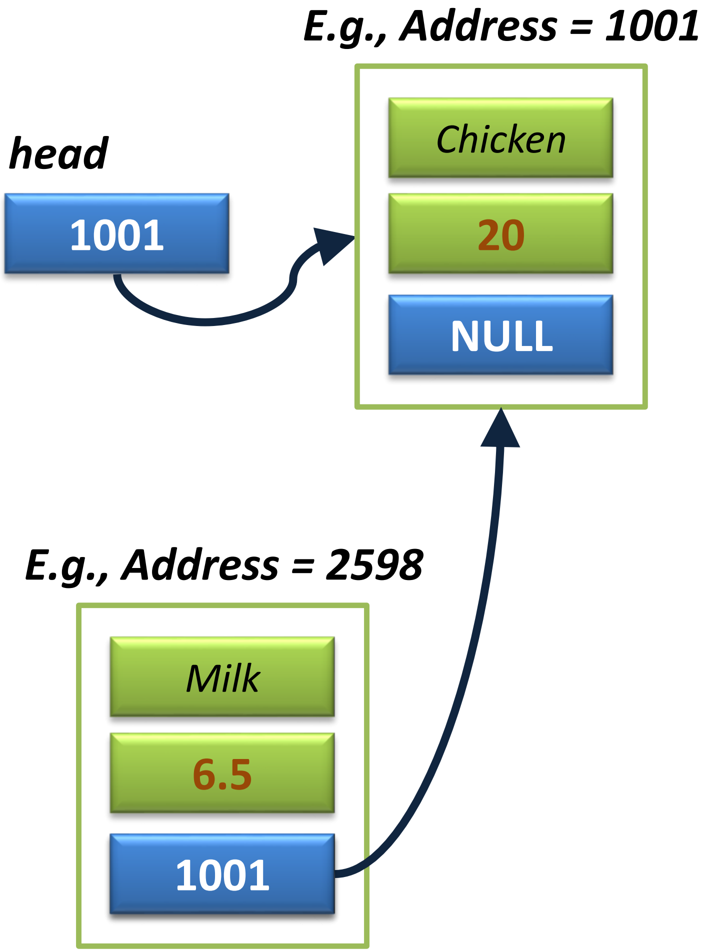
2. Insertion



Insertion steps

- **Step 1.** Create a new node.

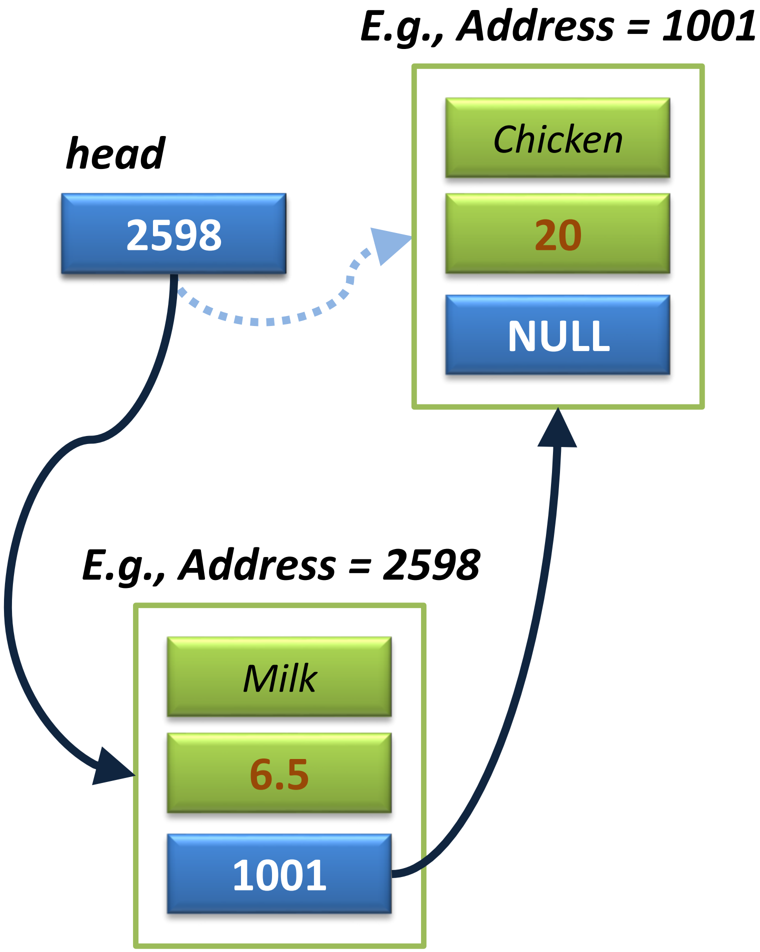
2. Insertion



Insertion steps

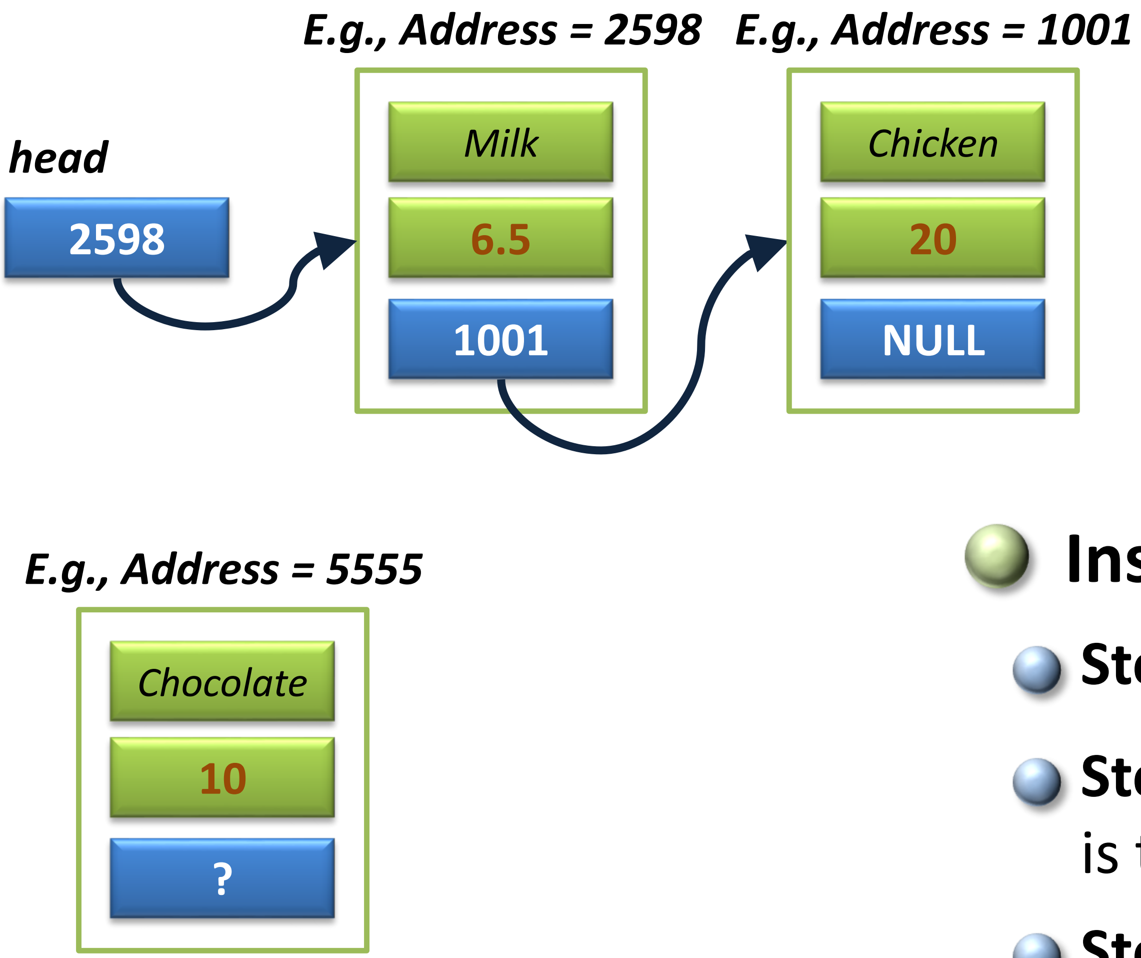
- **Step 1.** Create a new node.
- **Step 2.** New node’s next is the first node.

2. Insertion



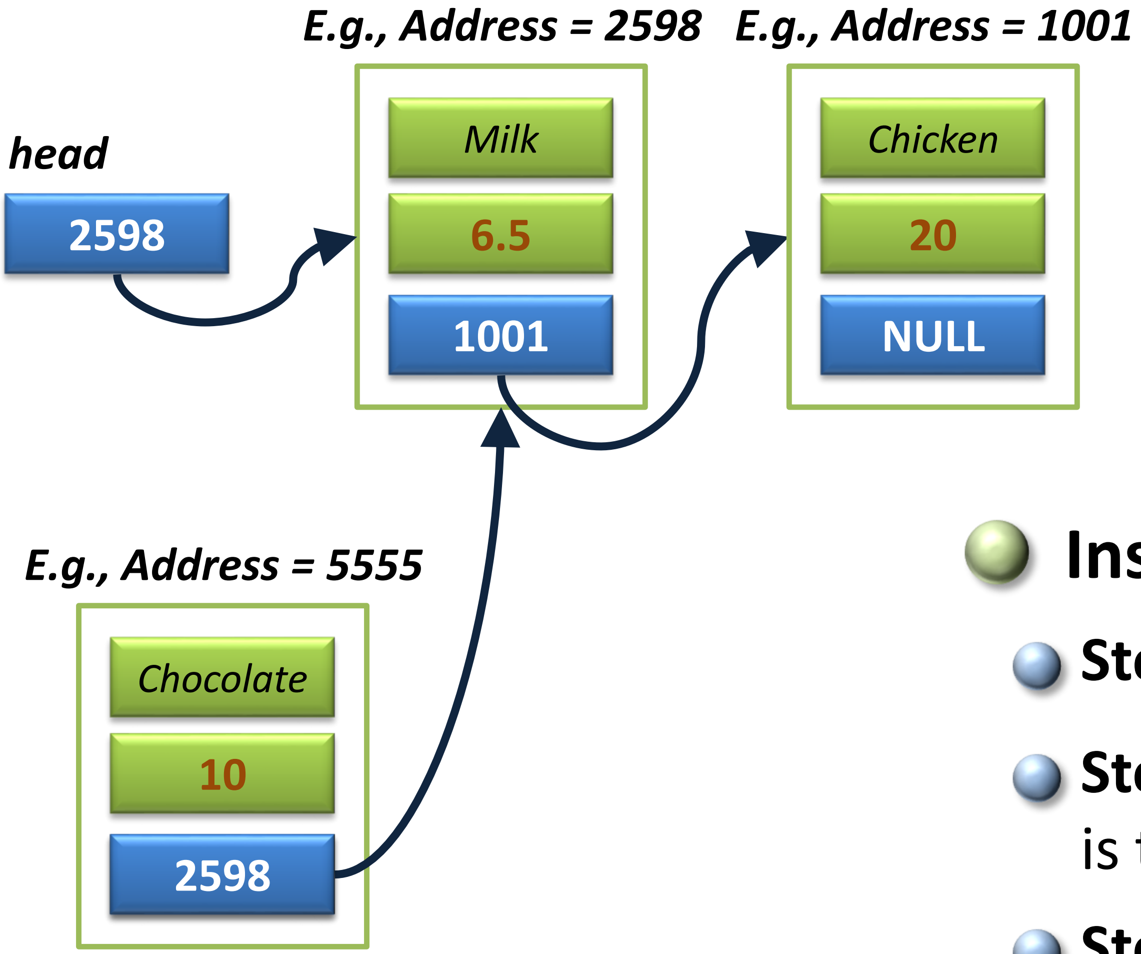
- **Insertion steps**
  - **Step 1.** Create a new node.
  - **Step 2.** New node's next is the first node.
  - **Step 3.** Head points to the new node.

2. Insertion



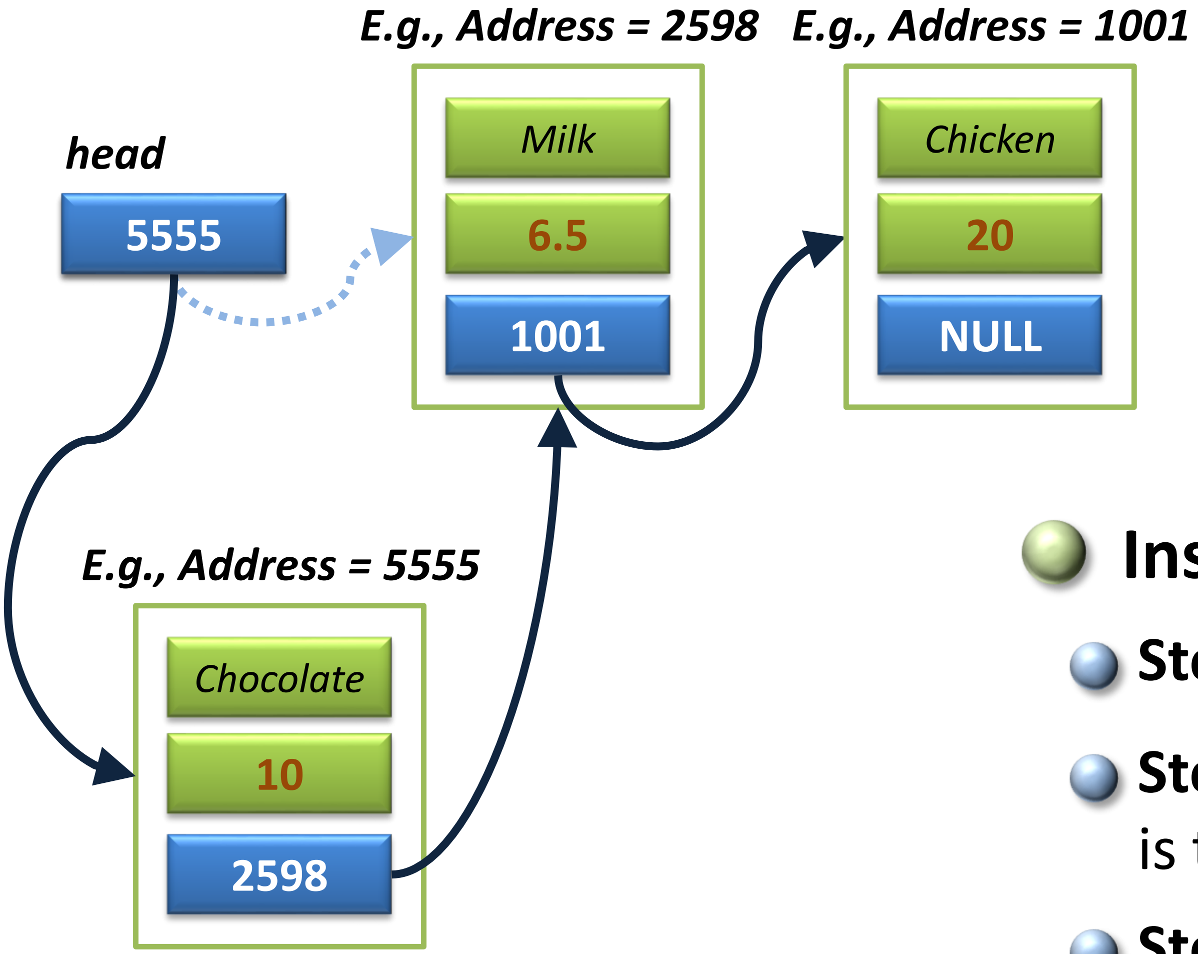
- **Insertion steps**
  - **Step 1.** Create a new node.
  - **Step 2.** New node's next is the first node.
  - **Step 3.** Head points to the new node.

2. Insertion



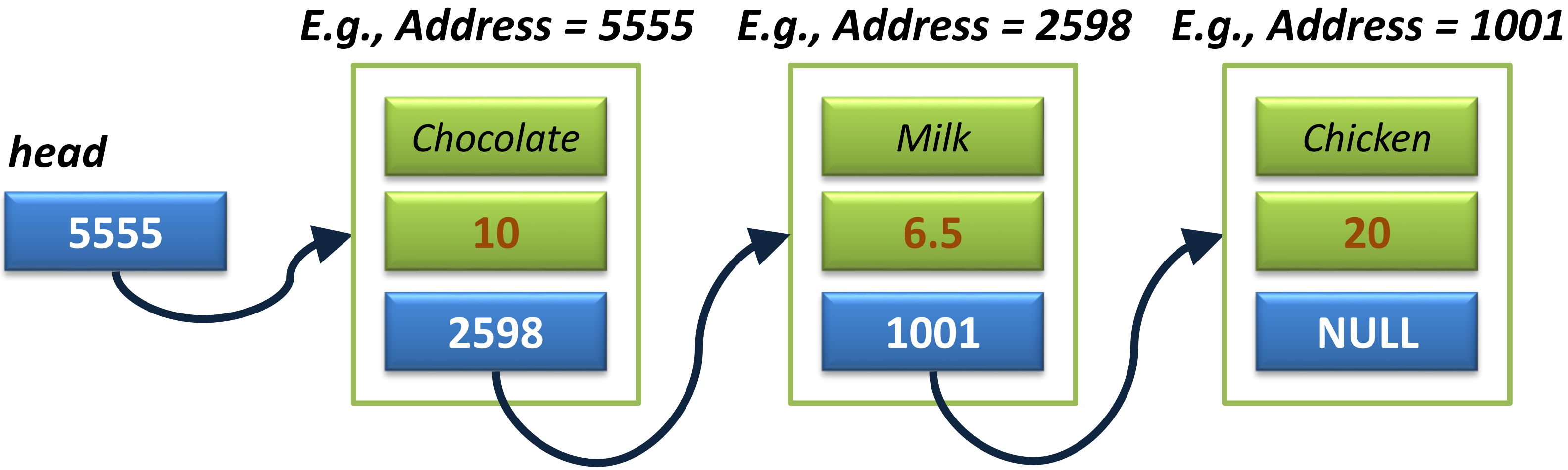
- **Insertion steps**
  - **Step 1.** Create a new node.
  - **Step 2.** New node's next is the first node.
  - **Step 3.** Head points to the new node.

2. Insertion



- **Insertion steps**
  - **Step 1.** Create a new node.
  - **Step 2.** New node's next is the first node.
  - **Step 3.** Head points to the new node.

2. Insertion

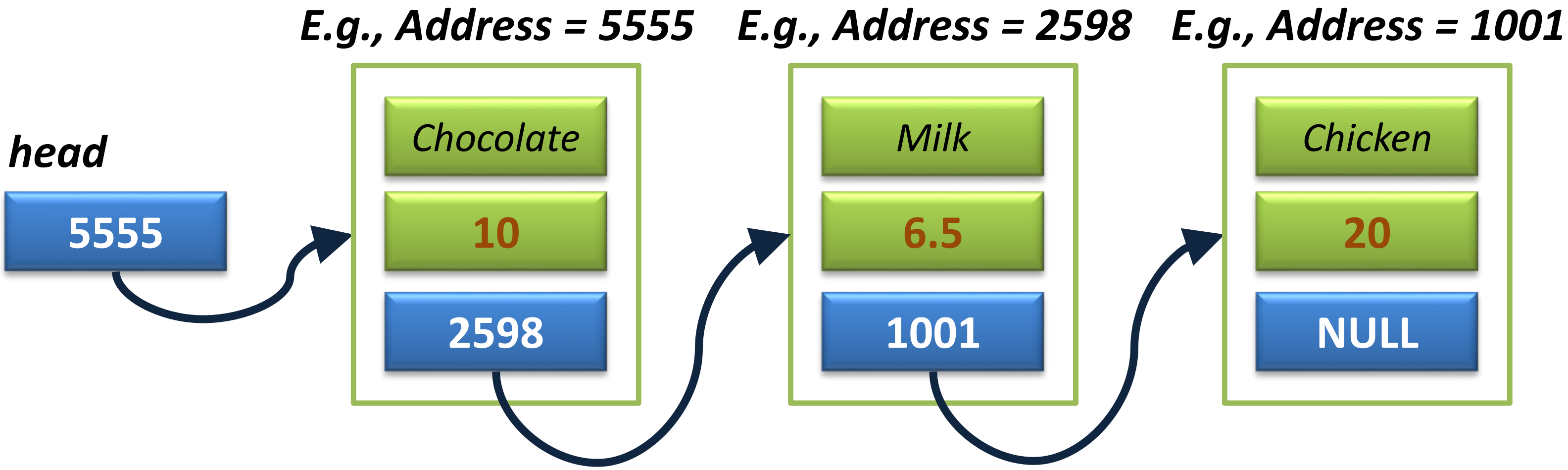


```
Product * headInsert (Product *h, string n, double p){




    return h;
}
```

Insertion steps

- **Step 1.** Create a new node.
- **Step 2.** New node's next is the first node.
- **Step 3.** Head points to the new node.

2. Insertion

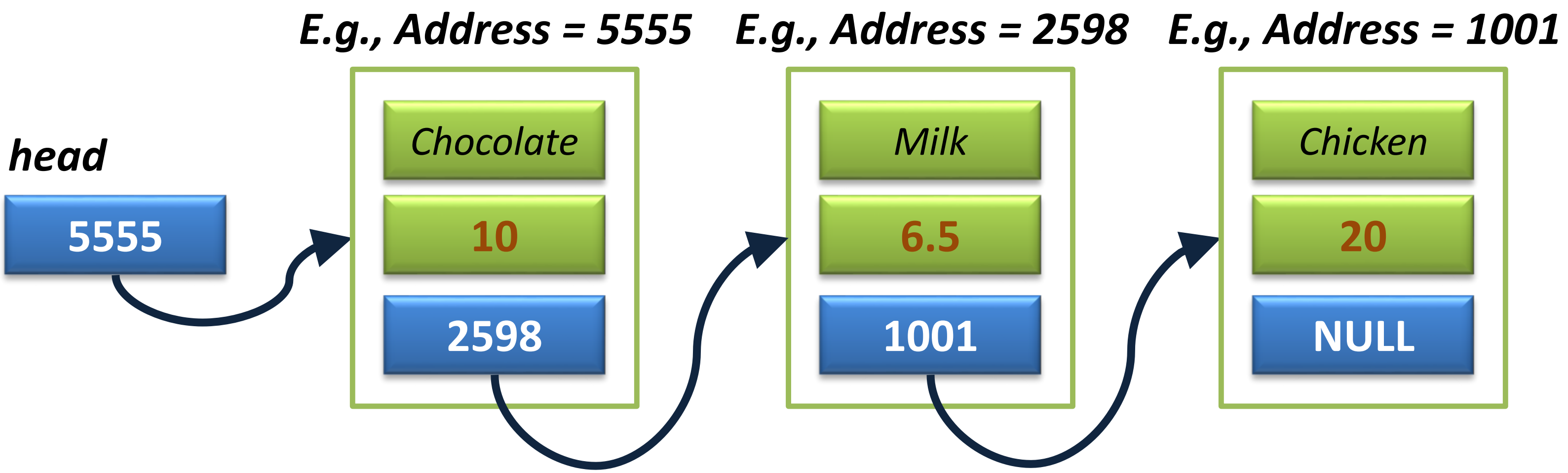


```
Product * headInsert (Product *h, string n, double p){
    Product *pNode = new Product;
    pNode -> name = n;
    pNode -> price = p;


    return h;
}
```

Insertion steps

- **Step 1.** Create a new node.
- **Step 2.** New node's next is the first node.
- **Step 3.** Head points to the new node.

2. Insertion

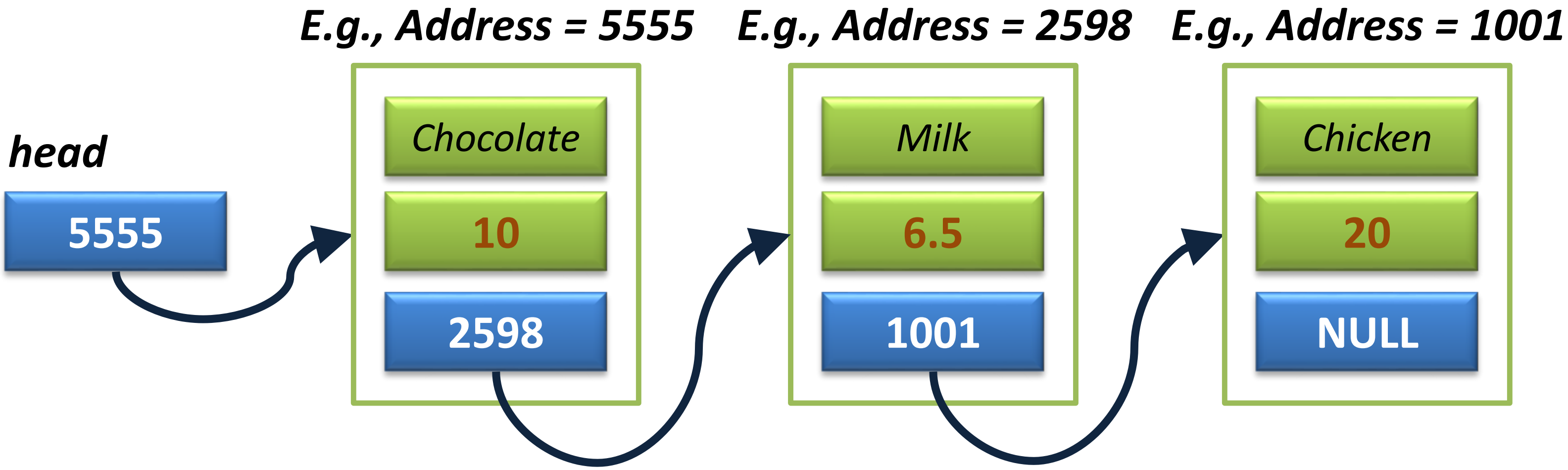


```
Product * headInsert (Product *h, string n, double p){
    Product *pNode = new Product;
    pNode -> name = n;
    pNode -> price = p;
    pNode -> next = h;

    return h;
}
```

Insertion steps

- **Step 1.** Create a new node.
- **Step 2.** New node's next is the first node.
- **Step 3.** Head points to the new node.

2. Insertion

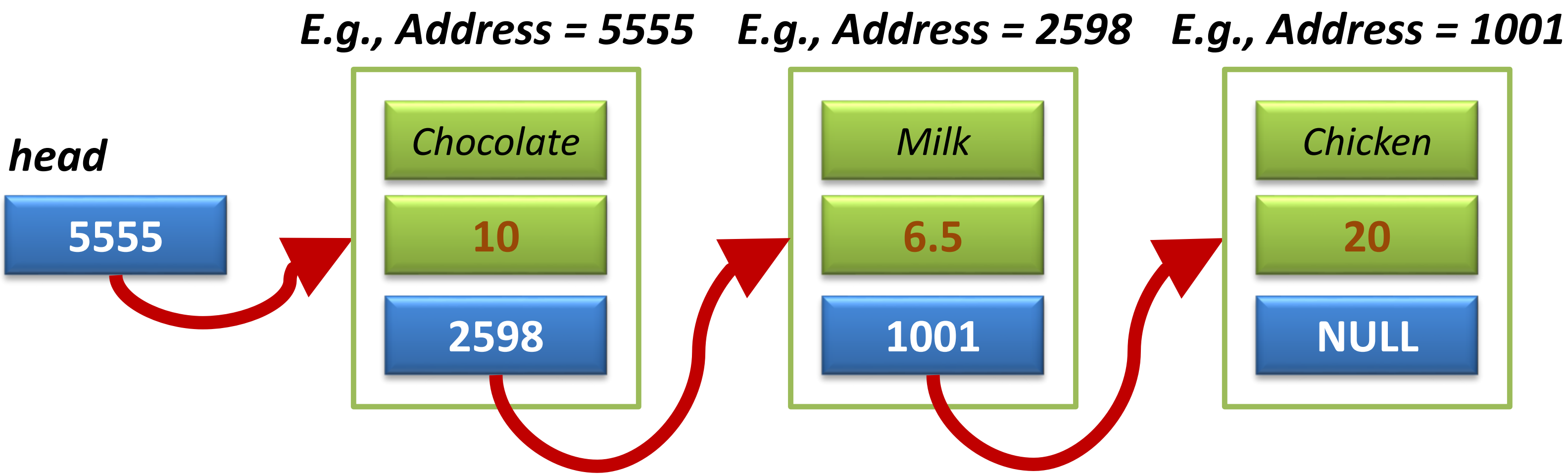


```
Product * headInsert (Product *h, string n, double p){
    Product *pNode = new Product;
    pNode -> name = n;
    pNode -> price = p;
    pNode -> next = h;
    h = pNode;
    return h;
}
```

Insertion steps

- **Step 1.** Create a new node.
- **Step 2.** New node’s next is the first node.
- **Step 3.** Head points to the new node.

3. Searching



E.g., Address = 5555 E.g., Address = 2598 E.g., Address = 1001

head

5555

Chocolate

10

2598

Milk

6.5

1001

Chicken

20

NULL

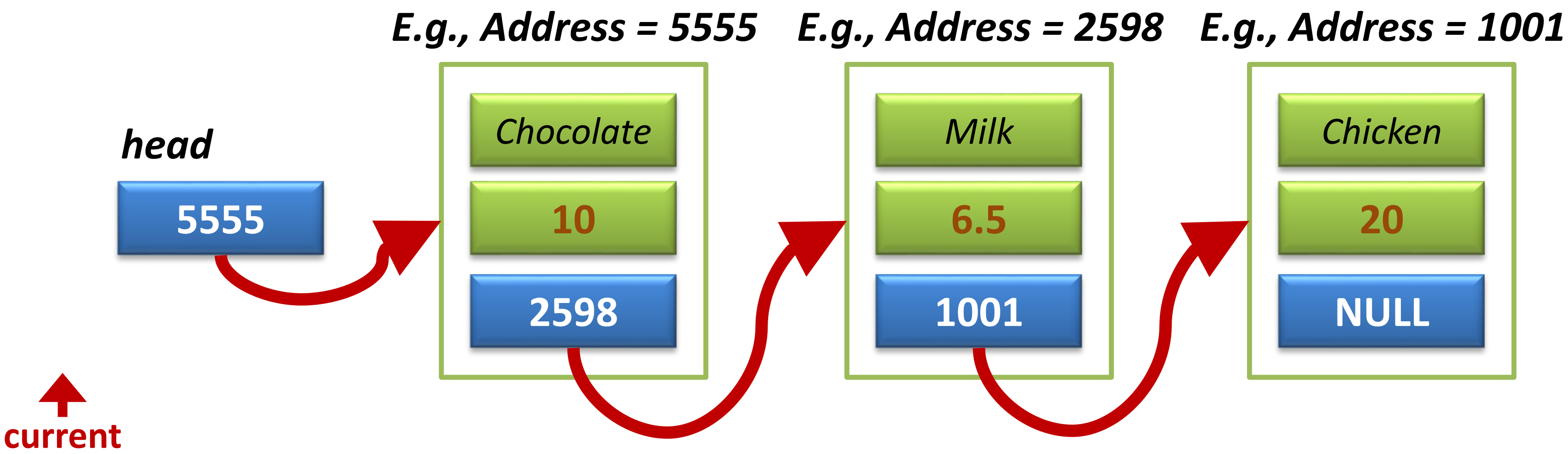
```
void searchList (Product *head, string n){






}
```

List traversal

- **Searching steps**

We can use the pointers to traverse the list!

3. Searching



E.g., Address = 5555 *E.g., Address = 2598* *E.g., Address = 1001*

head

5555

Chocolate

10

2598

Milk

6.5

1001

Chicken

20

NULL

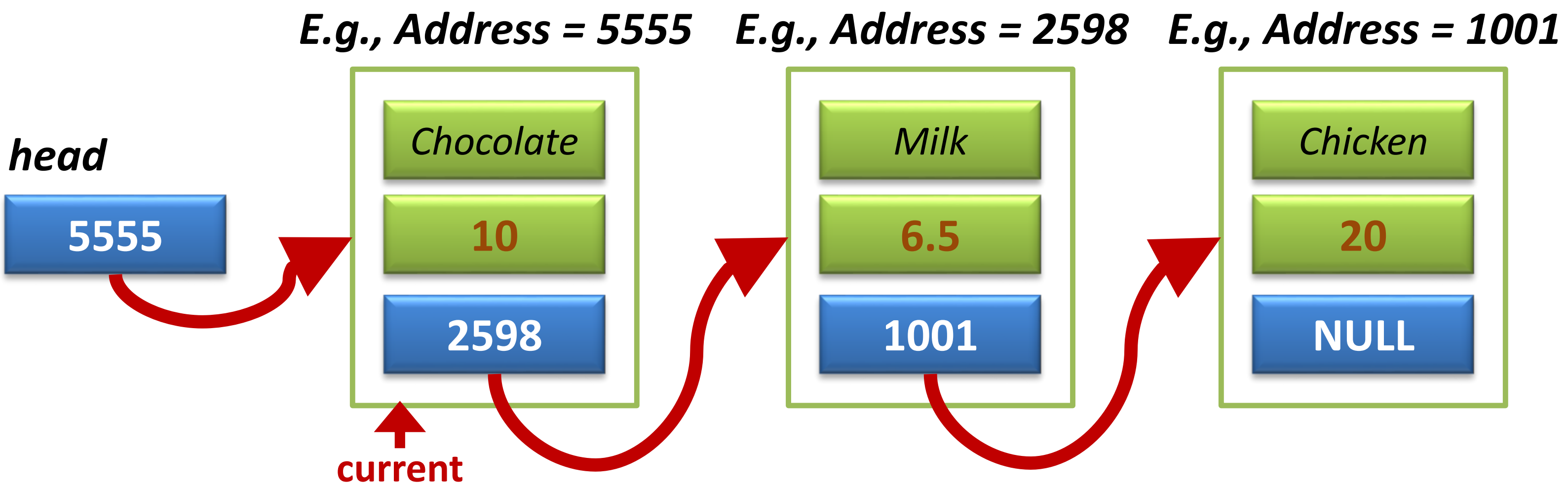
current

```
void searchList (Product *head, string n){
    Product *current;

}
```

List traversal

- **Searching steps**
  - **Step 1.** Create a pointer ***current***.

3. Searching

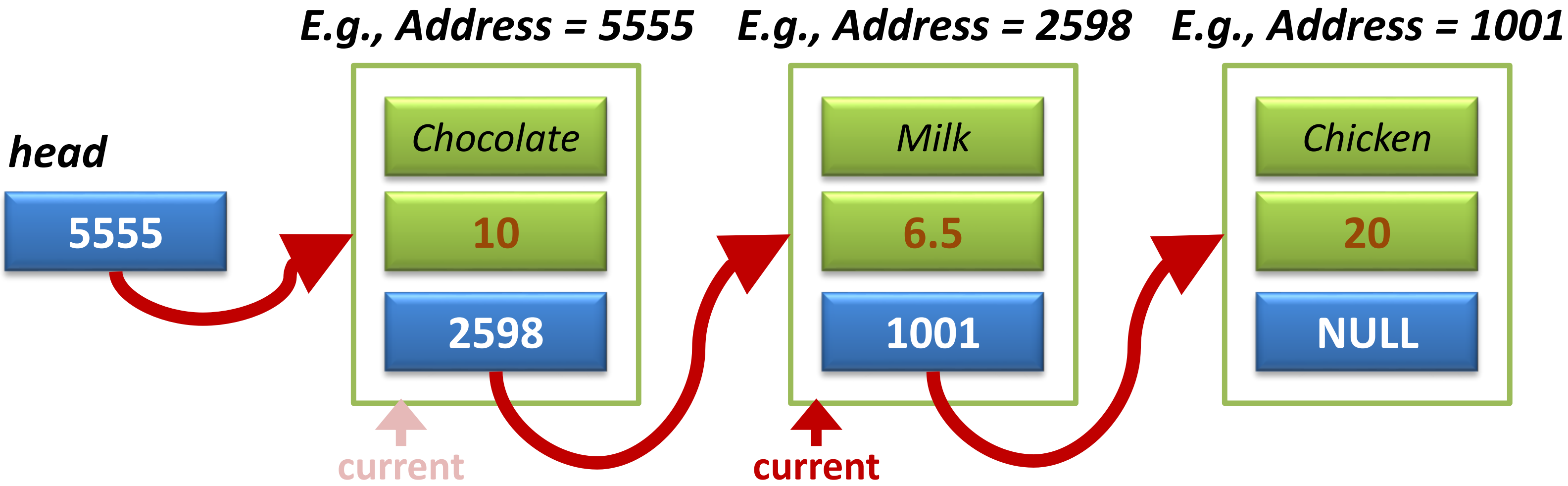


```
void searchList (Product *head, string n){
   Product *current;
   current  = head;

}
```

- **Searching steps**
  - **Step 1.** Create a pointer ***current***.
  - **Step 2.** Initialize ***current***:

    ***current* = *head*;**

3. Searching



```
void searchList (Product *head, string n){
  Product *current;
  current  = head;
  while (current != NULL){
     if (current -> name == n)
          cout << current-> price << endl;
     current = current -> next;
  }
}
```

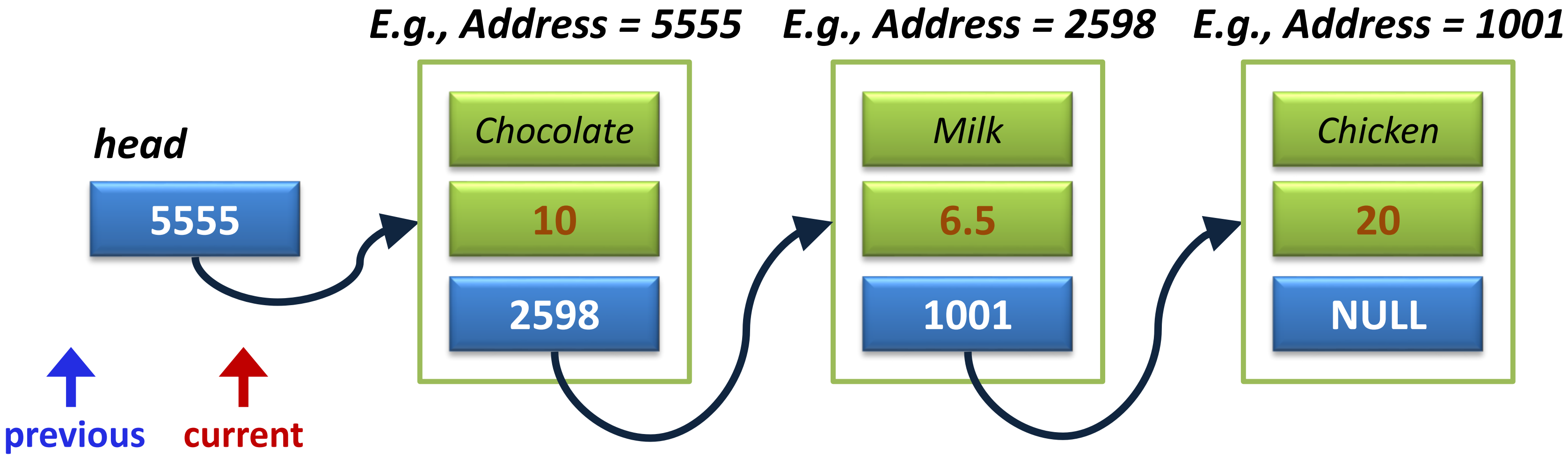
List traversal

- **Searching steps**
  - **Step 1.** Create a pointer ***current***.
  - **Step 2.** Initialize ***current***:

    current = ***head***;
  - **Step 3.** Traverse the linked list by:

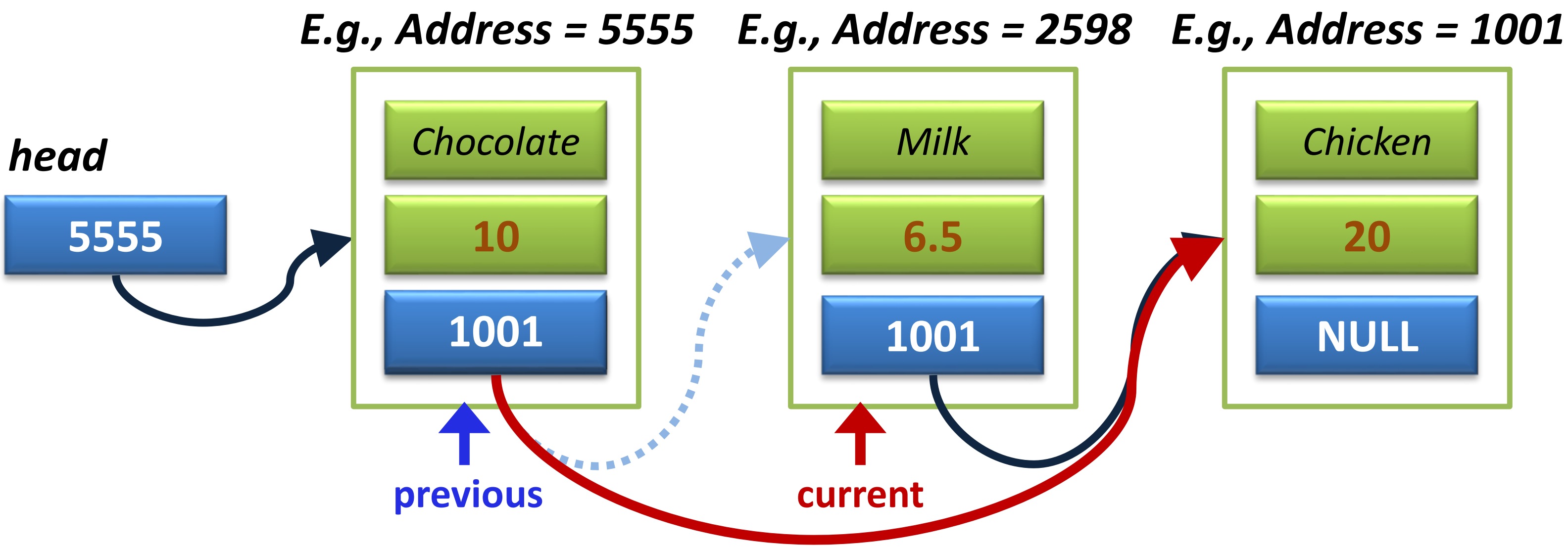
    current = ***current*** -> ***next***;

4. Deletion



- **Deletion steps (Except the first node)**
  - **Step 1.** Create 2 pointers ***current, previous***.

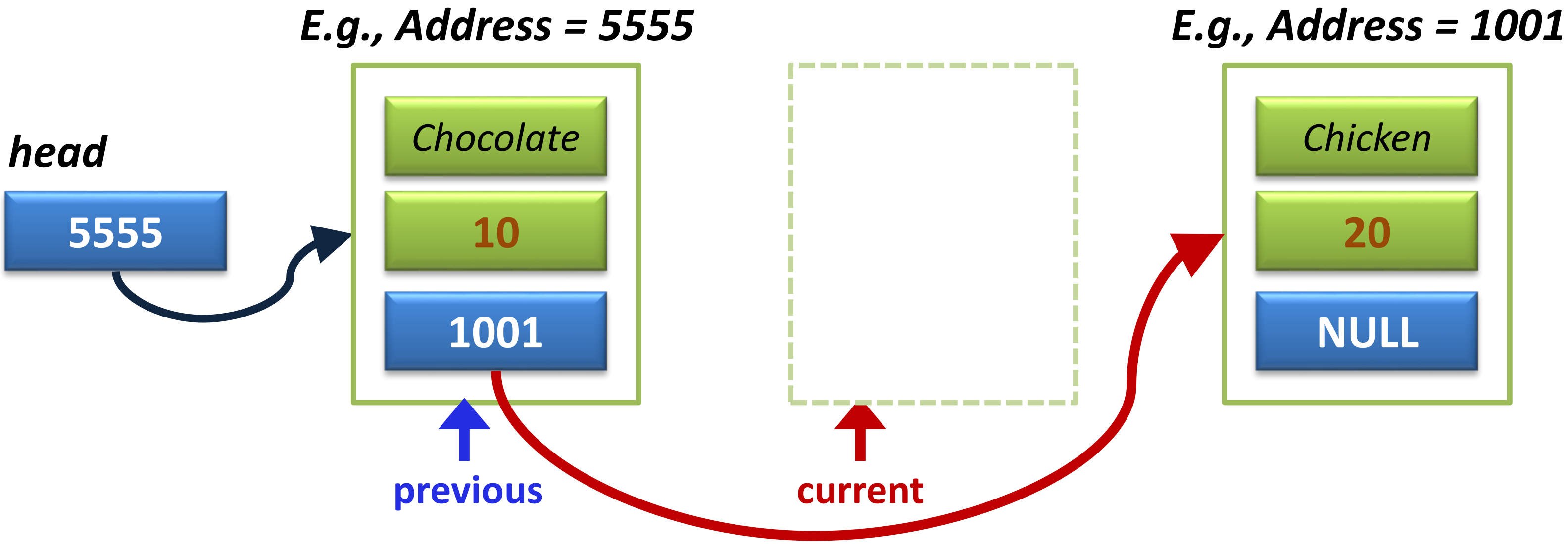
4. Deletion



- Deletion steps (Except the first node)
  - Step 1. Create 2 pointers ***current, previous***.
  - **Step 2.** Search for the node to delete, and update the pointers:

    previous -> ***next*** = ***current*** -> ***next***;

4. Deletion

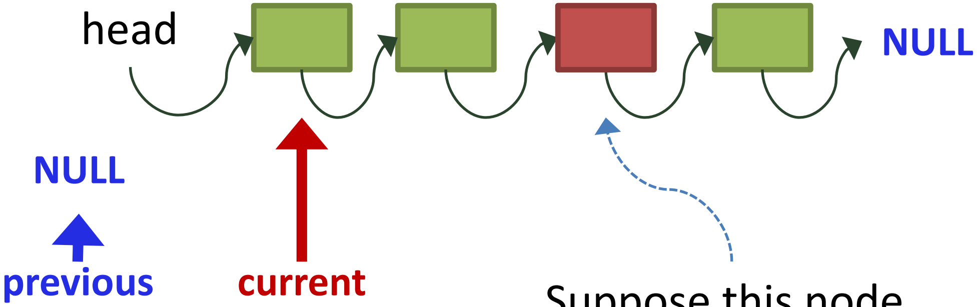


- **Deletion steps (Except the first node)**
  - **Step 1.** Create 2 pointers ***current, previous***.
  - **Step 2.** Search for the node to delete, and update the pointers:

    previous -> ***next*** = ***current*** -> ***next***;
  - **Step 3.** Remove the node to delete:

    delete *current*;

4. Deletion



Suppose this node is to be deleted.

```
int main(){
    …
    Product *current, *previous;
    previous = NULL;
    current = head;

    // More to be done
    …
}
```

Remove a node

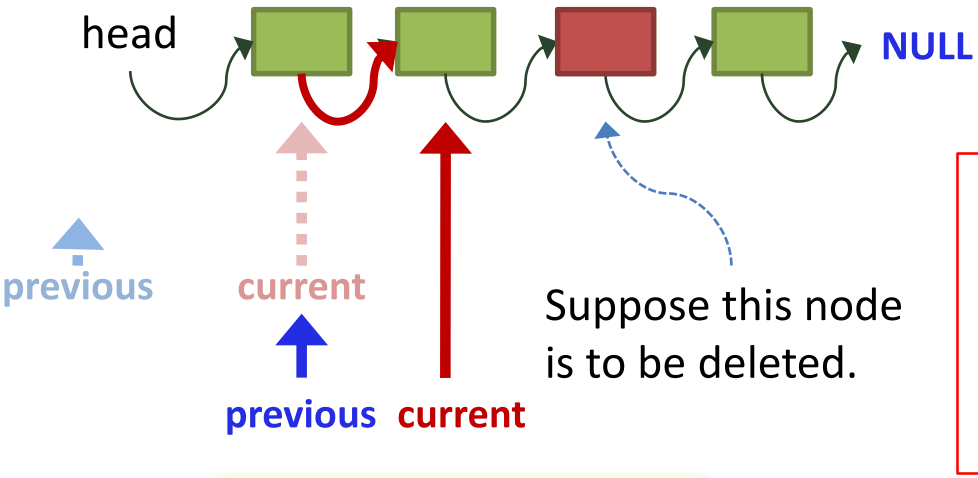
Step1. Create 2 pointers ***current*** and ***previous***, how should we initialize the two pointers?

Answer:

Initially , ***current*** points to the **first node**.

previous points to **NULL**.

4. Deletion



Suppose this node is to be deleted.

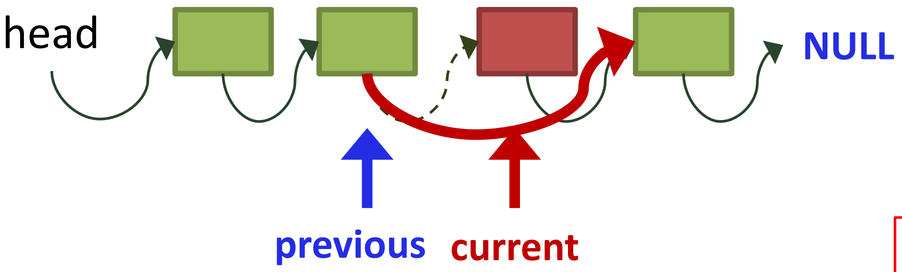
```
int main(){
    …
    Product *current, *previous;
    previous = NULL;
    current = head;

    while (current != NULL){
        // More to be done

        previous = current;
        current = current -> next;
    }
}
```

Remove a node

Step2. Search for the node to delete, and update the pointers. **How do we traverse the linked list**?

Answer:
We traverse the linked list by:
previous = ***current***;
current = ***current*** -> ***next***;

4. Deletion

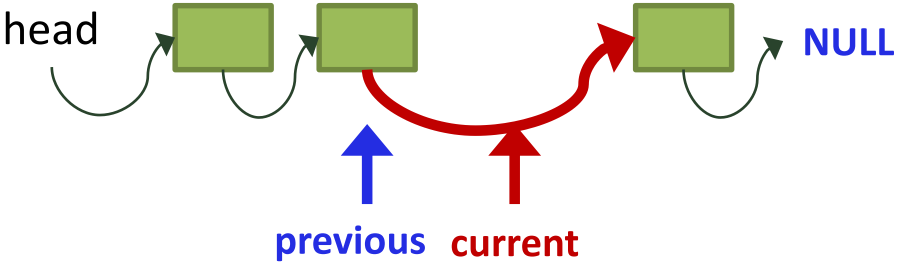


```
int main(){
    …
    Product *current, *previous;
    previous = NULL;
    current = head;

    while (current != NULL){
        if (current -> name == "Chicken"){
            previous -> next = current -> next;
            // More to be done
        }

        previous = current;
        current = current -> next;
    }
}
```

Remove a node

Step2. If we found the node to delete, update the pointers. **How should the pointers be updated**?

4. Deletion

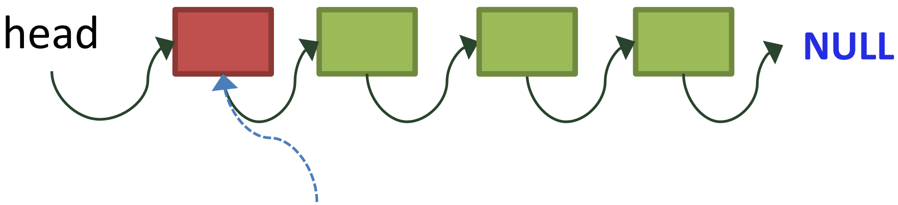


```
int main(){
   …
   Product *current, *previous;
   previous = NULL;
   current = head;

   while (current != NULL){
       if (current -> name == "Chicken"){
              previous -> next = current -> next;
              delete current;
              break;
       }

       previous = current;
       current = current -> next;
   }
}
```

Remove a node

Step3. Remove the node to delete.

4. Deletion



Suppose this node is to be deleted.

```
int main(){
   …
   Product *current, *previous;
   previous = NULL;
   current = head;

   while (current != NULL){
        if (current -> name == "Chicken"){
                previous -> next = current -> next;
                delete current;
                break;
        }

        previous = current;
        current = current -> next;
   }
   …
}
```

Remove a node

Take home exercise…
Think about if the node to be deleted is in the first node.
Will there be any problems?

We are happy to help you!

“Are the concepts too difficult? If you face any problems in understanding the materials, **please feel free to contact me, our TAs or student TAs**. **We are very happy to help you!**
We wish you enjoy learning programming in this class ☺.”

Module 8B.

END

COMP2113 Programming Technologies / ENGG1340 Computer Programming II
Dr. T.W. Chim (E-mail: twchim@cs.hku.hk)
Department of Computer Science, The University of Hong Kong