University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Pointer and Linked List Review

Lecturer: Le Ngoc Thanh
Email: lnthanh@fit.hcmus.edu.vn

HCM City

Outline

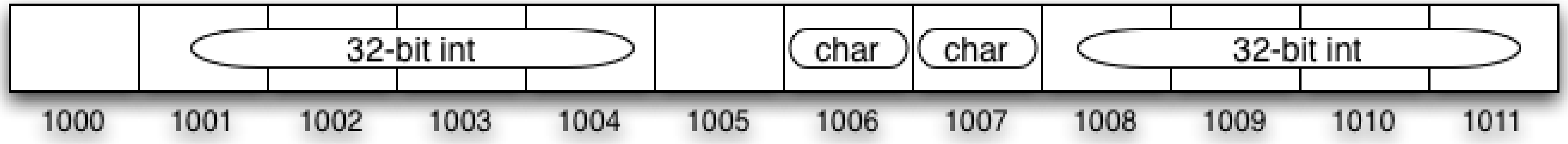
- Pointer
- Linked List

Main memory

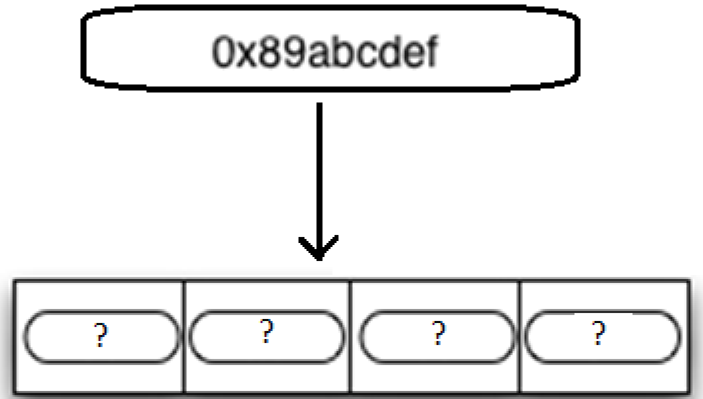
- Computer memory
  - RAM contains many cells, each with the size of 1 byte.
  - RAM is used to store part of the operating system, program instructions, data…
  - Each cell has a unique address and is indexed from 0 onwards (linear address space).
  - For example
    - RAM 512MB are addressed from 0 to $2^{29} - 1$
    - RAM 2GB are addressed from 0 to $2^{31} - 1$

Stored Value

- Depending on the data type, values can be stored in multiple cells.
  - The program only needs to know the starting address and size.

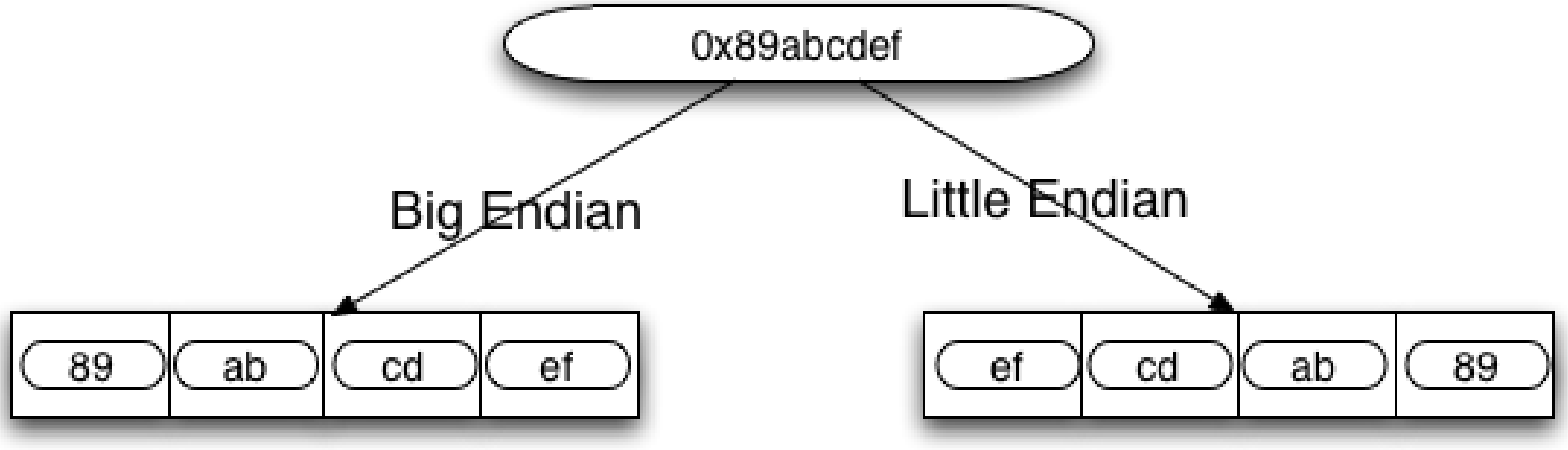


- How to contain a specific value?
  - Eg: x = 0x89abcdef



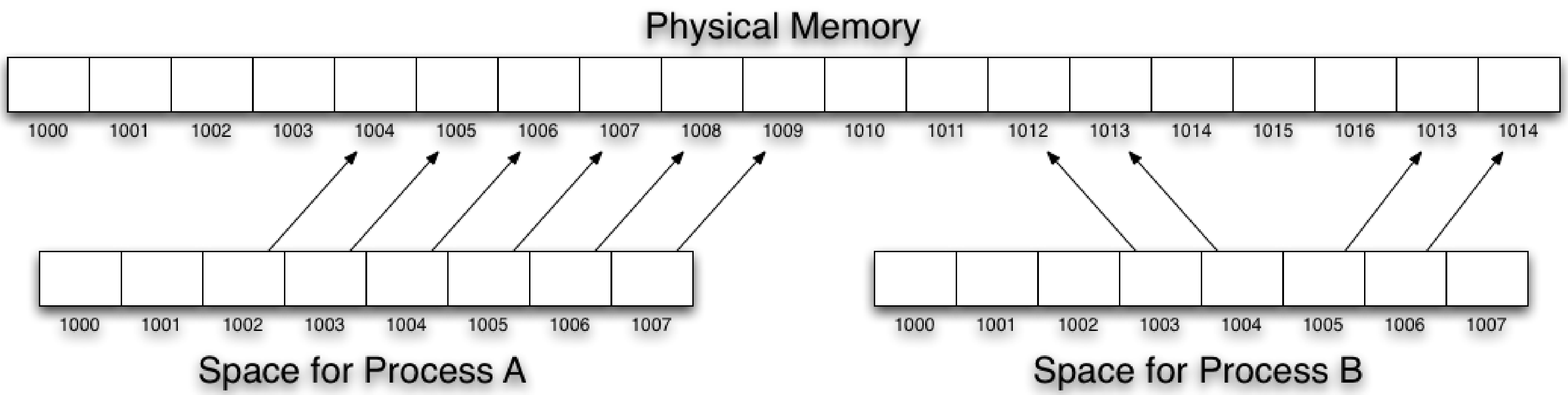
Stored Value

- The division of the data to store depends on where the most significant digit is stored.



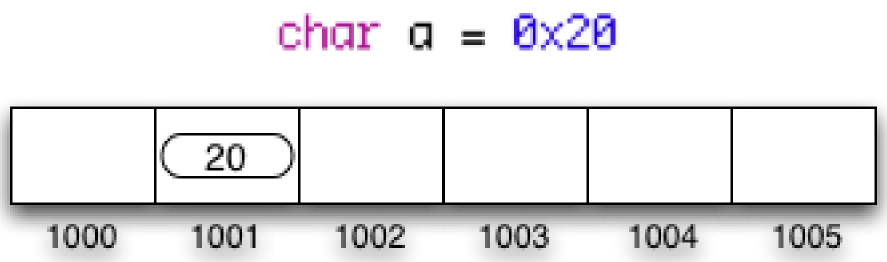
Virtual address

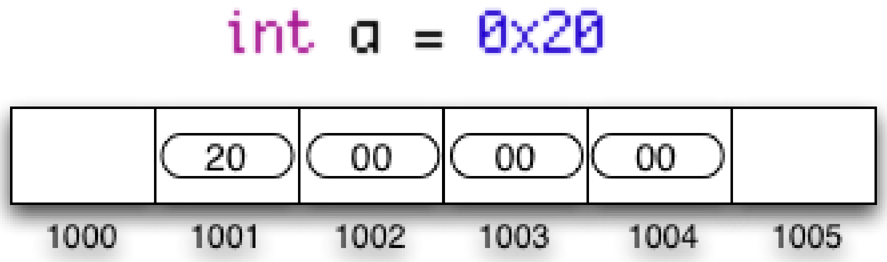
- In modern operating systems, the space allocated to programs is typically virtualized.
  - Contains a virtual memory -> physical memory mapping table.
  - Protect access to memory



Variables and cells

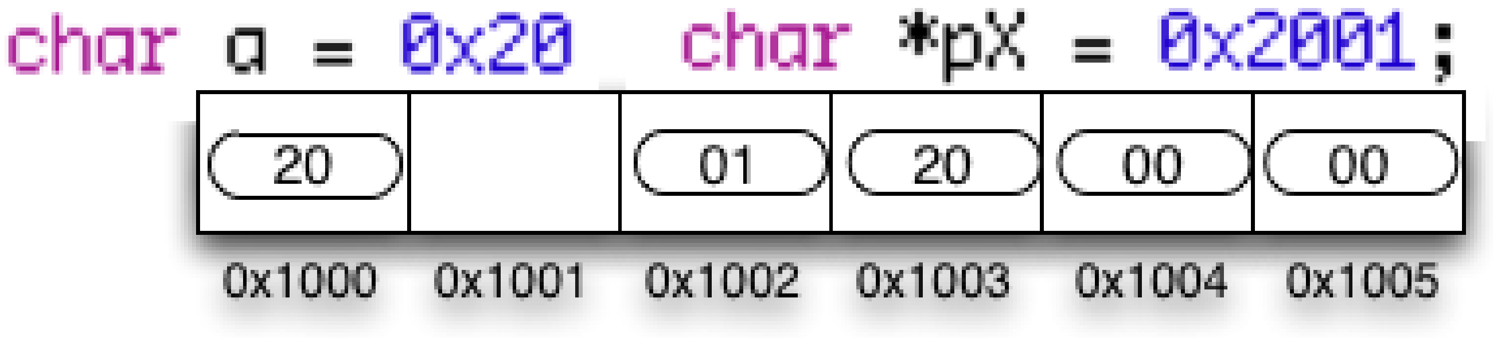
- Each variable is identified at a memory address
  - Can read and write values
  - It is possible to occupy many consecutive memory cells based on the type of data in which the variable is declared.
  - Have a certain lifetime and scope





Pointer

- Pointer is really just a variable
  - It also has an address and a value.
  - But the value contained is only the address.



Pointer Size

- What is the size of the pointer variable?

```
char *p1;
int *p2;
float *p3;
double *p4;
…
```

  - The pointer size depends on memory space, not on the data type declared.
    - MD-DOS (16 bit): 2 bytes (64KB)
    - Windows (32 bit): 4 bytes (4GB)
  - Data type refers to the value where it points to.

Declaring pointers

- Declare
  - Like any other variable, the pointer variable to be used needs to be declared

```
<data_type> *<pointer_name>;
```

- Example

```
char *ch1, *ch2;
int *p1, p2;
```

  - ch1 and ch2 is a pointer which points to a char (1 byte).
  - p1 is a pointer which points to an int (4 bytes) and p2 is a normal variable of type int.

Declaring pointers

- When declared, the pointer variable is placed at a certain address.
  - ➔ contains an undefined value
  - ➔ point to unknown memory.
- Therefore, it is not recommended to use pointers without being initialized.

```
int *p;
*p = 1904; // !!!
```

Pointer assignment

- Since pointer contains only addresses, pointer is assigned only one address value
  - Assign a specific address.
    - Eg: int *p;   p = 0x12AB; //danger!!!
  - Memory allocation.
    - Eg: int *p = new int;
  - Assign the address of the static variable.
    - Eg: int a;

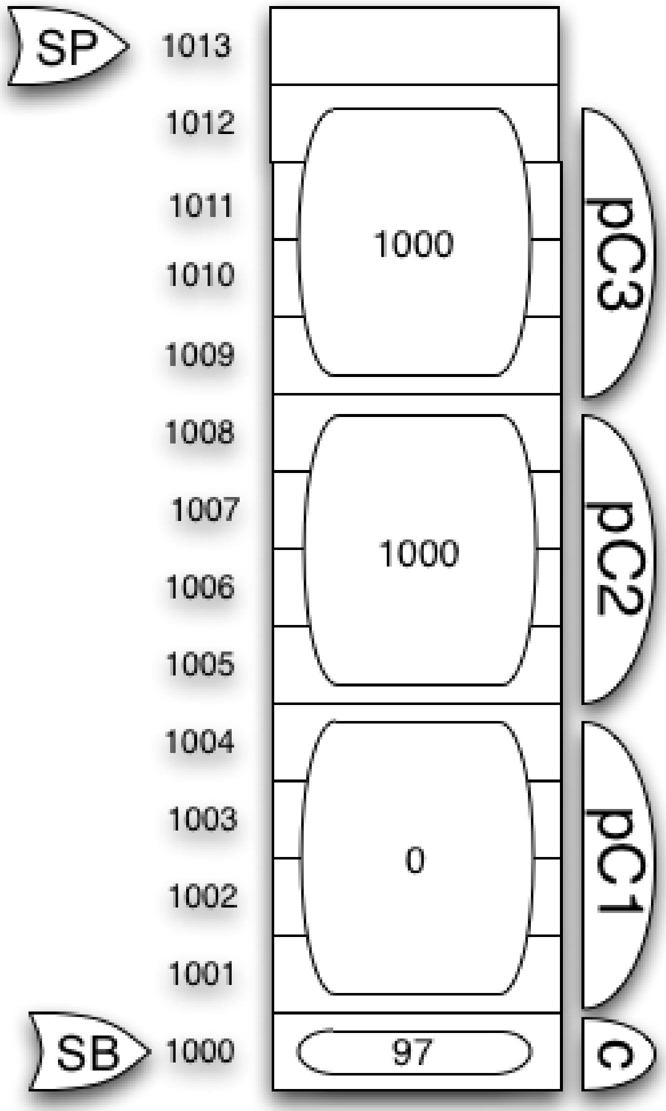
      int *p;   p = &a;//& is the operator to get the address
  - Assigns the address of another pointer.
    - Eg : int *p1, *p2;

      p1= p2;

      p1 = NULL;

Example

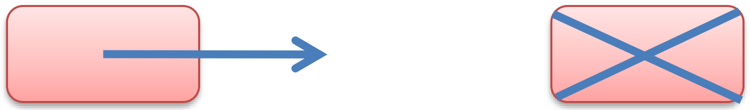
```
void foo()
{
    char c, *pC1, *pC2, *pC3;
    c = 'a';
    pC1 = NULL;
    pC2 = &c;
    pC3 = pC2;
}
```



NULL Pointer

- Concept
  - A NULL pointer is a pointer that does not point anywhere (or a value of 0 to say there is no pointed memory). It is different from the uninitialized pointer.
  - Reverse reference will cause execution errors.

```
int n;
int *p1 = &n;
int *p2;  // unreferenced local variable
int *p3 = NULL;
```

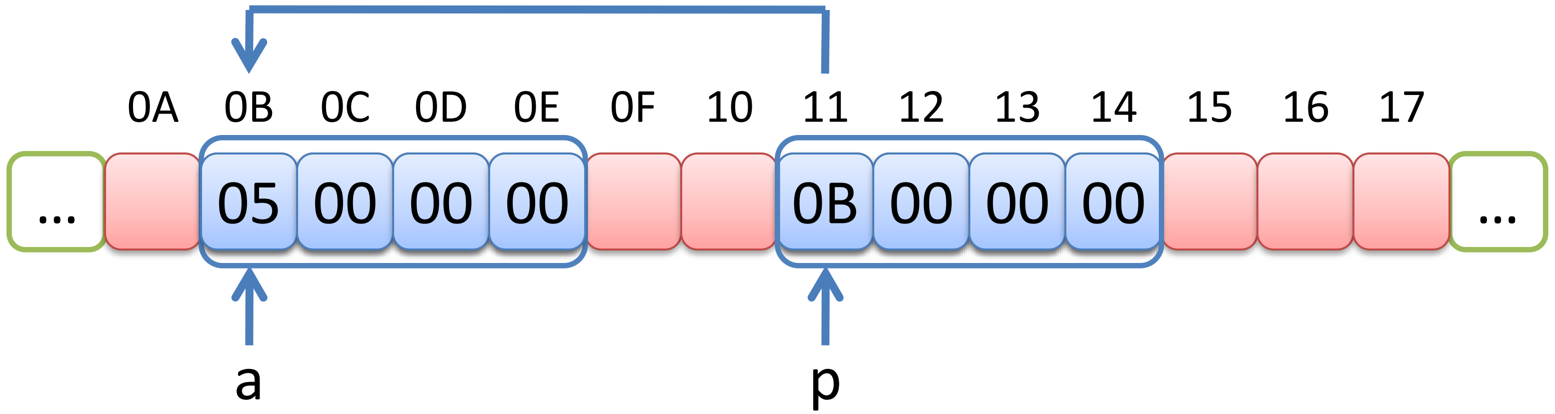


NULL

Dereference

- Accessing the pointed memory is called a dereference.
  - If p is a pointer, (* p) is the cell where it points to.

```
int a = 5;
int *p = &a;         //The * is used for a declaration,
                     //not for dereference
printf("%d\n", p);   // Variable value p
printf("%d\n", *p);  // Dereference
printf("%d\n", &p);  // Variable address p
```

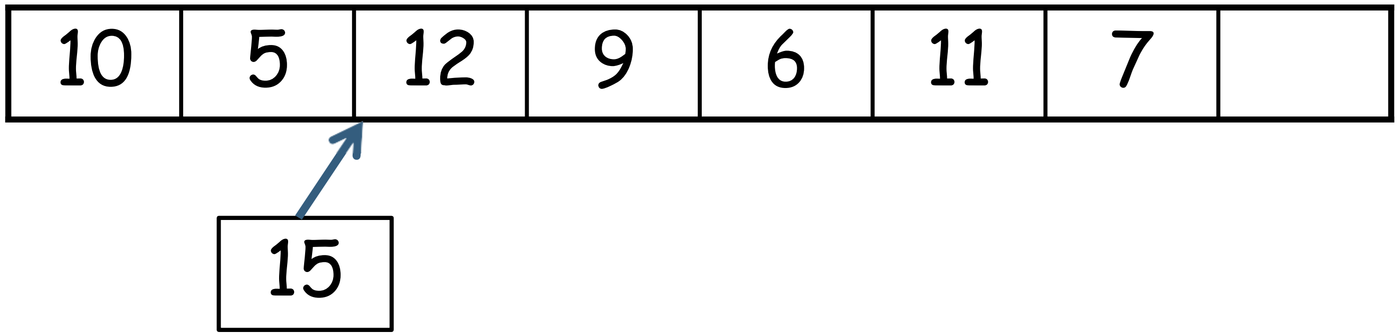


Outline

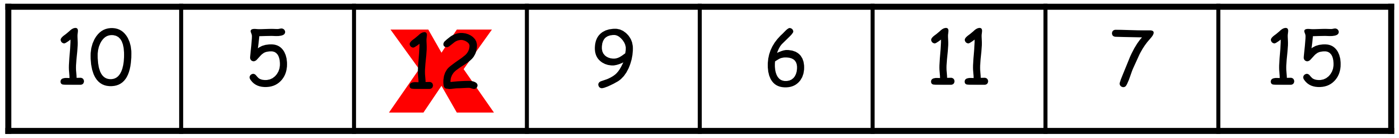
- Pointer
- Linked List

Array

- Use arrays to store list of elements:
  - Insert an element: O(n)



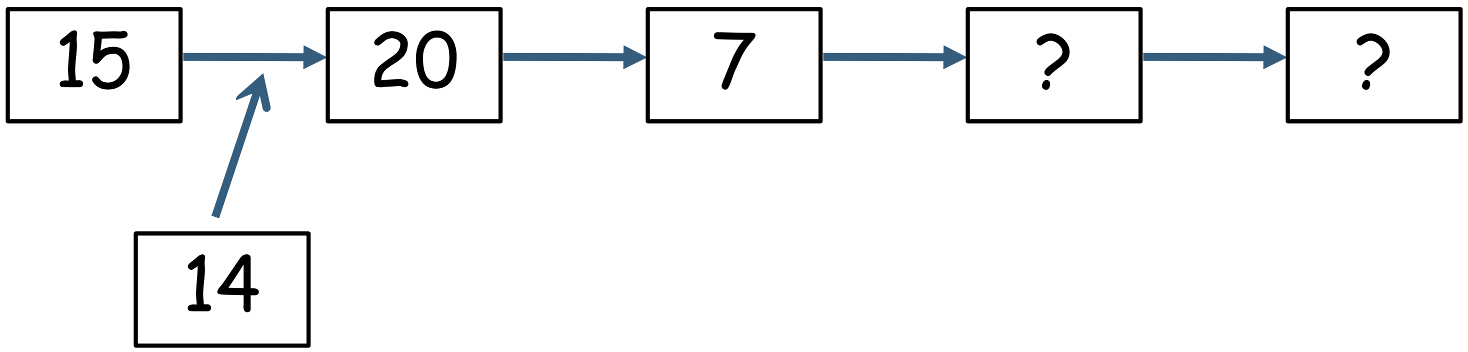
  - Delete an element: O(n)



  - The array size is fixed!

Linked List

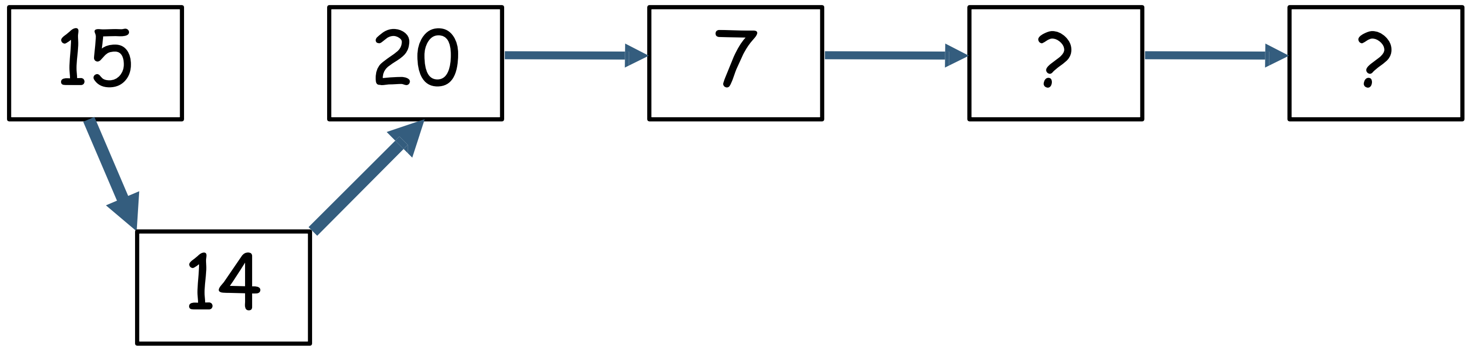
- Use linked list to store list of elements:
  - The elements are separated
  - and connected by chains



  - How to insert new element?

Insert new element to linked list

- The insertion only needs to change the links in place.



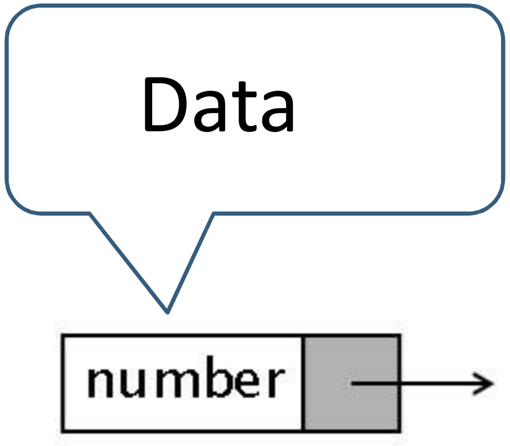
- Low cost of execution

Linked List

- A sequence of nodes
- Between two nodes there is a link pointer
- Nodes do not need to be continuously stored in memory
- Optionally expandable (limited only by memory capacity)
- The Insert/Delete operation doesn't need to move the element
- The first element is the pHead
- Other elements can be accessed through linked pointers

Construction of a node

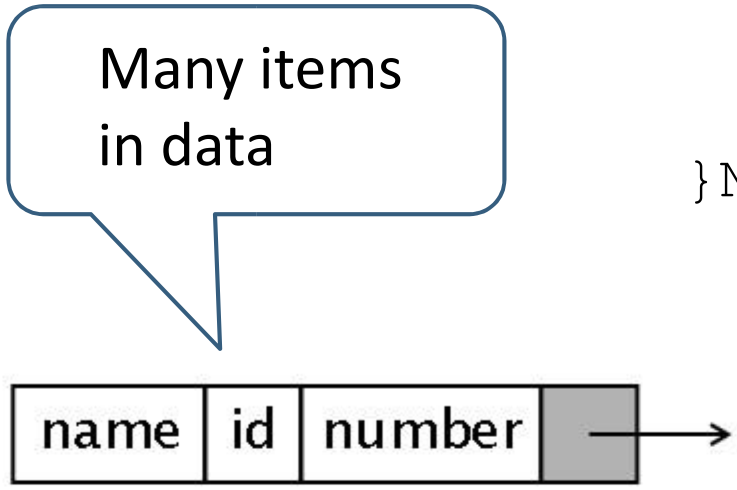
- Created by dynamic memory allocation
- Each node has 2 information:
  - Data
  - The pointer links to the next element in the list
- The last element in the list has the pointer pNext = NULL



```
Typedef struct tagNode{
  int        number;
  tagNode *pNext;
}NODE;
```

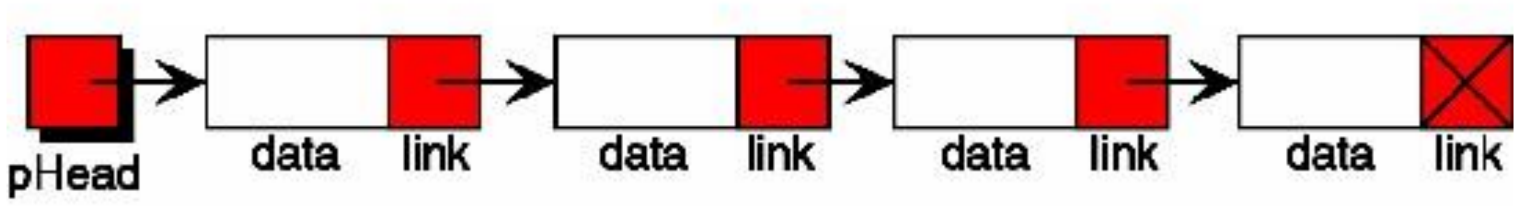
Construction of a node

```
typedef struct tagNode {
   char       name[30]
   int        id;
   float      grdPts;
   tagNode*pNext;
 }NODE;
```

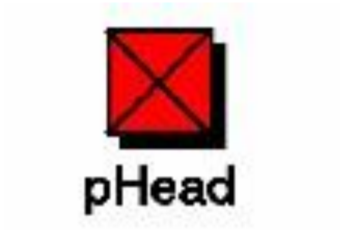


Structure of linked list

- Manage entire linked list via pHead pointer.
  - pHead is not a node but just a pointer to the node.
- We can also manage the list by adding end pointer (pTail)
  - pTail is not a node but just a pointer to the node.



Single linked list with the first element being pHead



List is empty, pHead = NULL

Create linked list

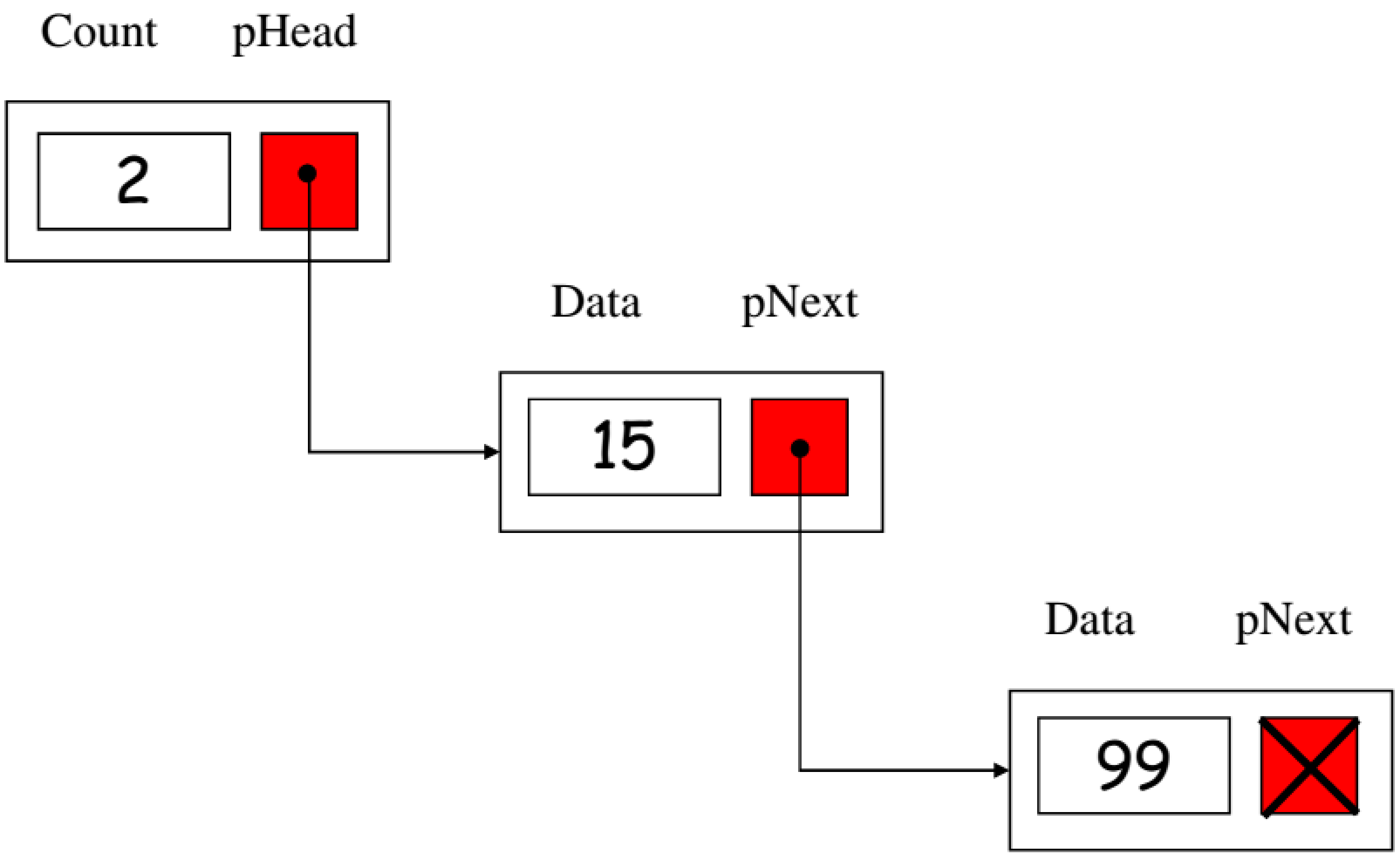
// Manage the list with the head pointer

```
typedef struct LINKED_LIST{
        NODE *pHead;
        unsigned int Count;           // the number of nodes in the list
}
```

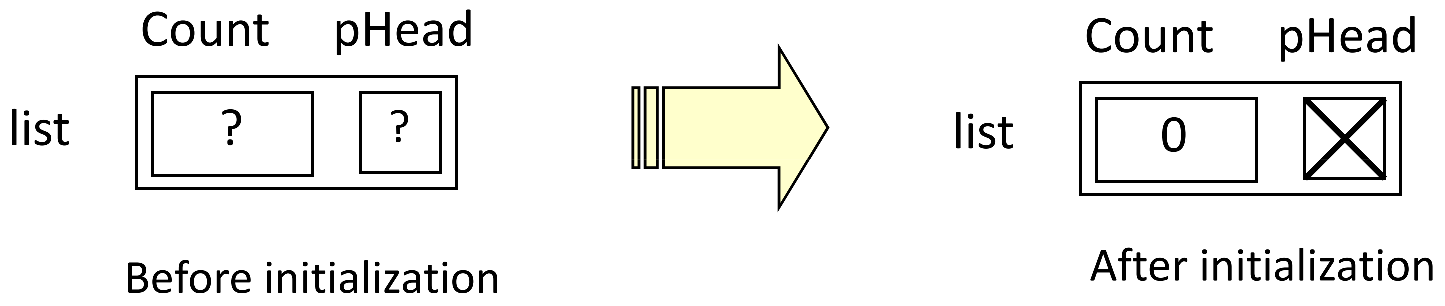
// Manage the list by head and tail pointers

```
typedef struct LINKED_LIST{
        NODE      *pHead, NODE *pTail;
        unsigned int Count;            // the number of nodes in the list
}
```

Example



Initialize empty list



```
void CreateEmptyList(LINKED_LIST &list)
{
  list.Count = 0;
  list.pHead = NULL;
}
```

Check out the linked list

- Check for empty list:

```
int  IsEmptyList(const LINKED_LIST &list)
{
     return (list.pHead ==NULL);
}
```

- Checks the number of items in the list:

```
int  CountNode(const LINKED_LIST &list)
{
   return list.Count;
}
```

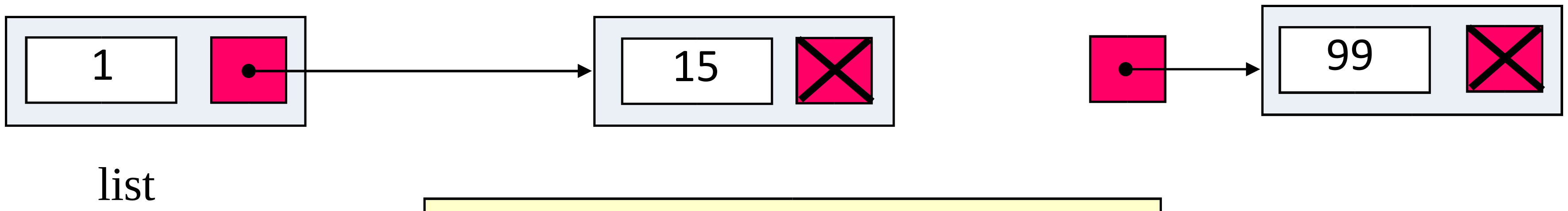
Create a node

```
NODE* CreateNode(<DataType> newdata)
{
    NODE *pNew = new NODE;
    if (pNew==NULL) return NULL;        //Error: cannot allocate
                                        //new element
    pNew->Data = newdata;
    pNew->pNext = NULL;
    return pNew;
}
```

Add a node to linked list

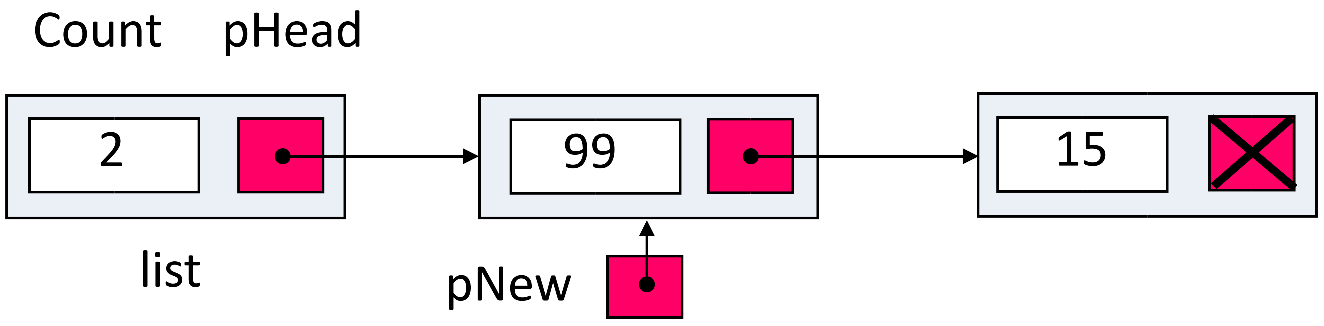
- Add to the top

Before insertion:



```
pNew->pNext = list.pHead;
list.pHead = pNew;
```

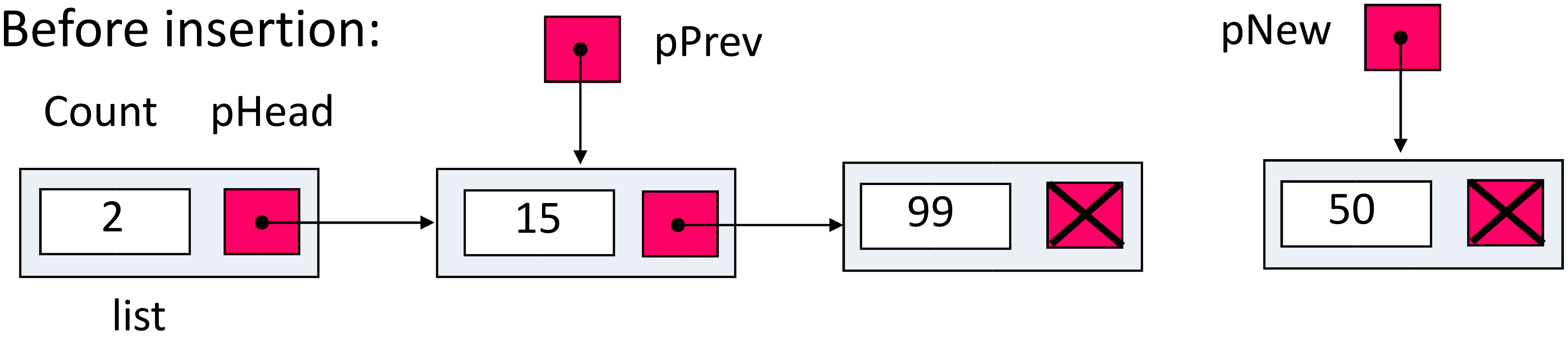
After insertion:



Add a node to linked list

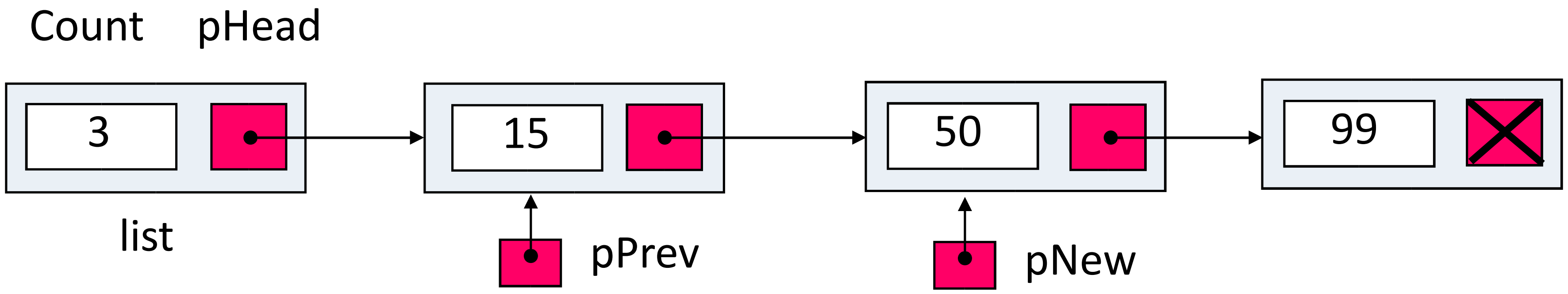
- Add to the inside

Before insertion:



```
pNew->pNext = pPrev->pNext;
pPrev->pNext = pNew;
```

After insertion:



Add a node to linked list

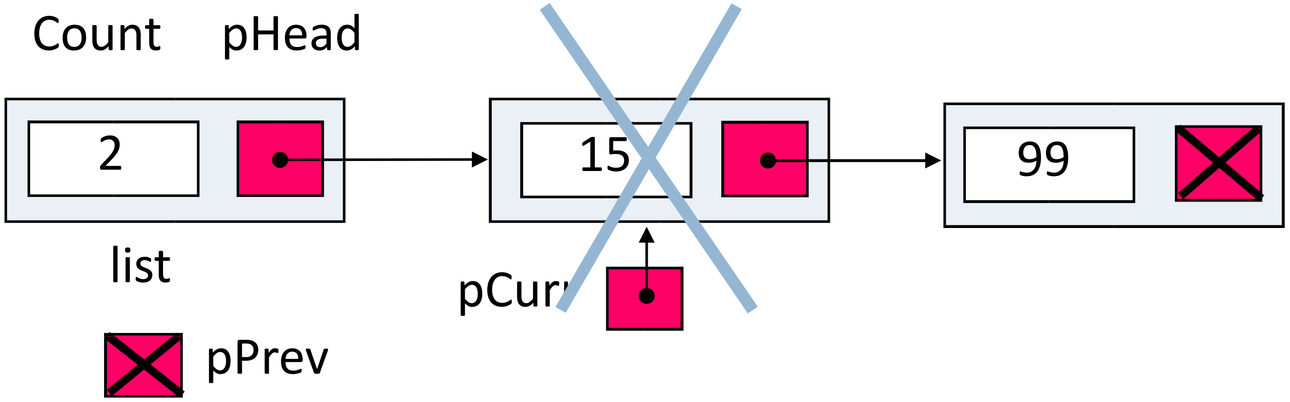
```
int  InsertNode(LINKED_LIST  &list, NODE *pPrev, <DataType>
   newdata)
{
   NODE *pNew;
   if (!(pNew = CreateNode(newdata)) return 0;
   // Add to the top of the list
   if (pPrev==NULL) {
         pNew->pNext = list.pHead;
         list.pHead = pNew;
   }

   else {// Add inside the list, after the pPrev element
         pNew->pNext = pPrev->pNext;
         pPrev->pNext = pNew;
   }
   list.Count++;
   return 1;
}  // end of InsertNode
```

Delete a node from linked list

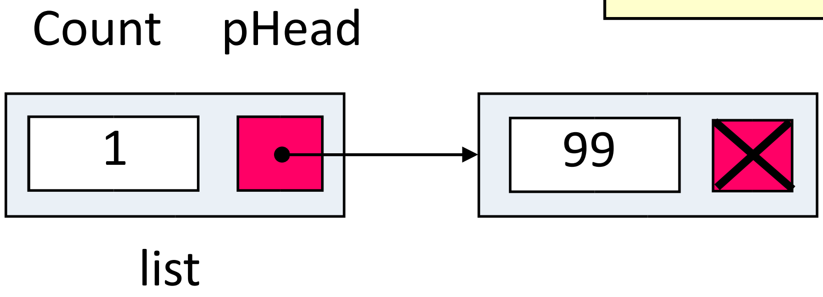
- Delete at the top of the list

Before deletion:



```
list.pHead = pCurr->pNext;
delete pCurr;
```

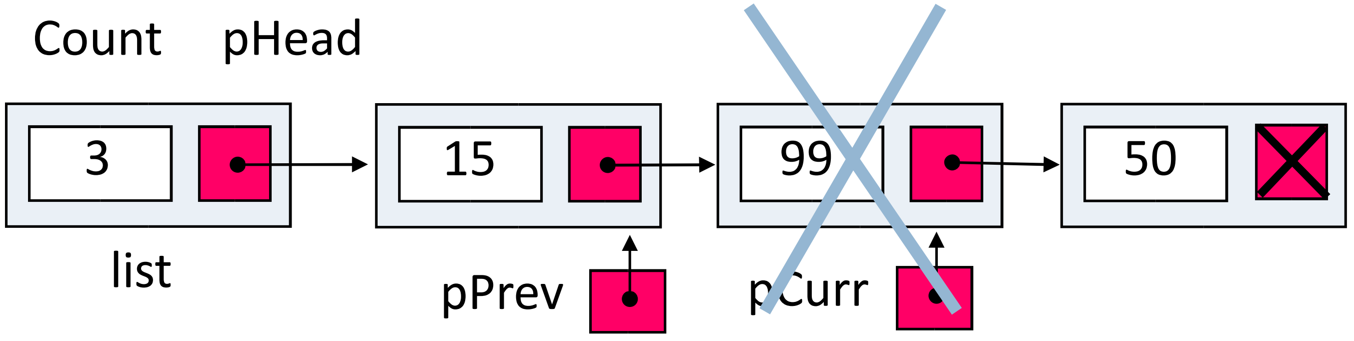
After deletion:



Delete a node from linked list

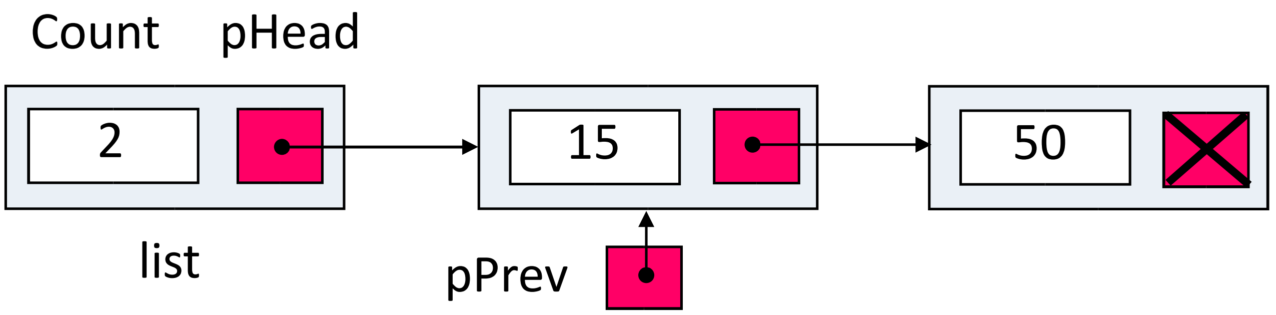
- Delete an element from inside the list

Before deletion:



```
pPrev->pNext    =   pCurr->pNext;
delete pCurr;
```

After deletion:

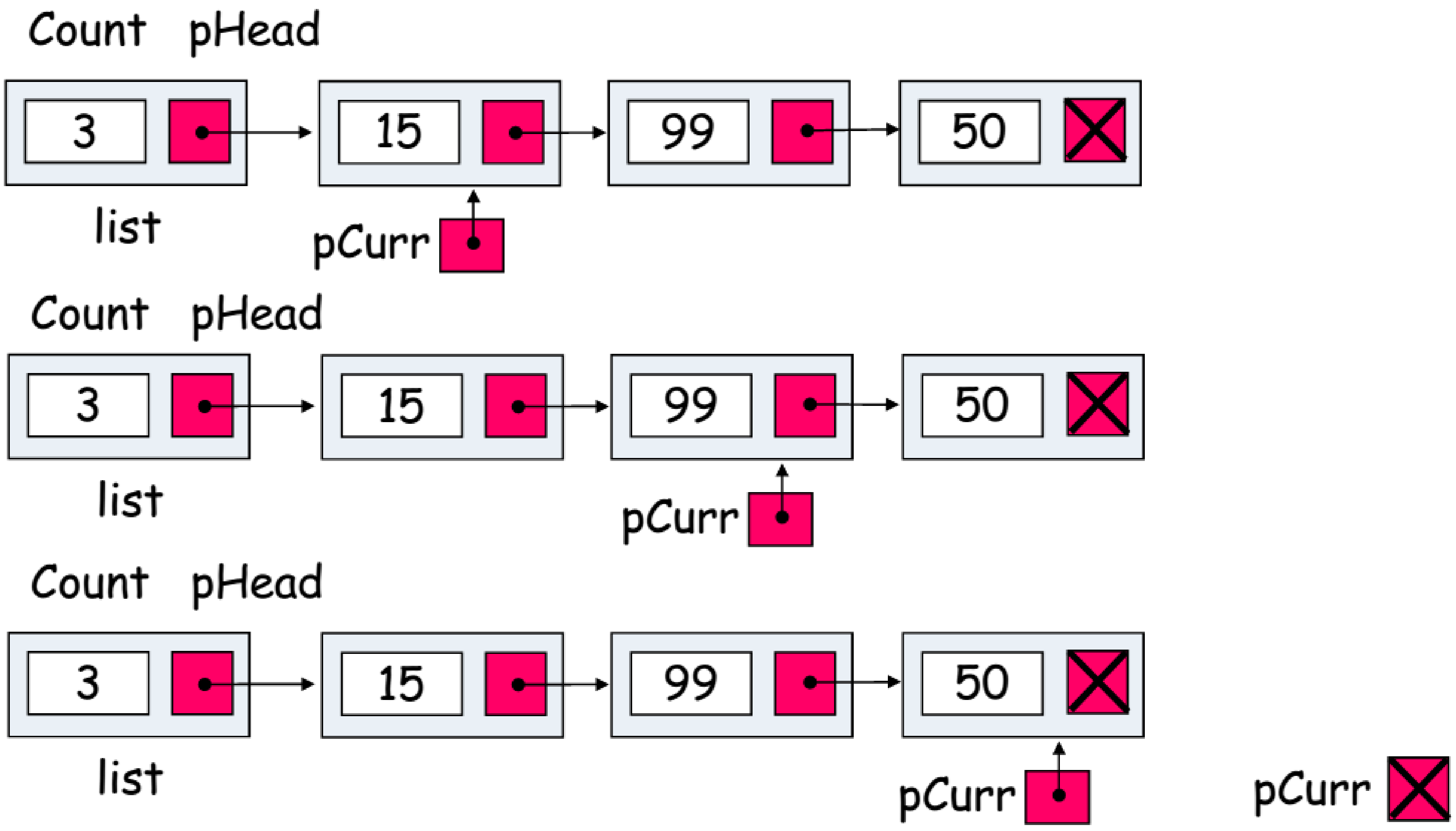


Delete a node from linked list

```
int  DeleteNode(LINKED_LIST  &list, NODE *pPrev, NODE *pCurr)
{
   if (pPrev==NULL)  // Delete the first node
           list.pHead = pCurr->pNext;
   else   // Delete the node inside
       pPrev->pNext = pCurr->pNext;

   delete pCurr;
   list.Count--;
   return 1;
}
```

Traverse linked list



Traverse linked list

```
void  TraverseList(const LINKED_LIST &list)
{

   NODE  *pCurr = list.pHead;
   while (pCurr!=NULL) {
            // Do something at pCurr
            pCurr = pCurr->pNext;   //go to next node
   }
}
```

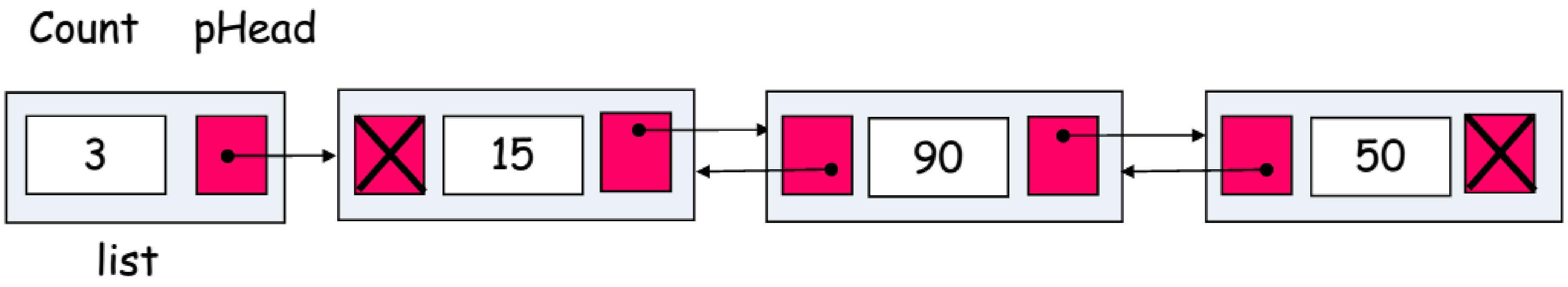
Traverse linked list to search

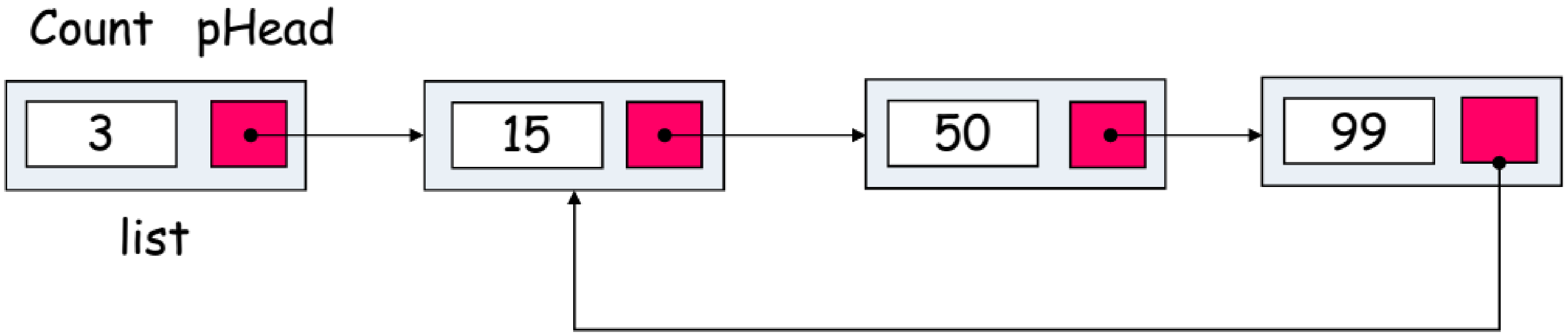
- Search an element

```
NODE * FindNode(const LINKED_LIST &list,<DataType> key)
{

   NODE  *pCurr = list.pHead;
   while (pCurr!=NULL) {
      if (pCurr->Data==key)
           return pCurr;    // Found
      pCurr = pCurr->pNext;   // go to next node
   }
   return NULL;  // Not found
}
```

Doubly linked list, Circular linked list





The End.