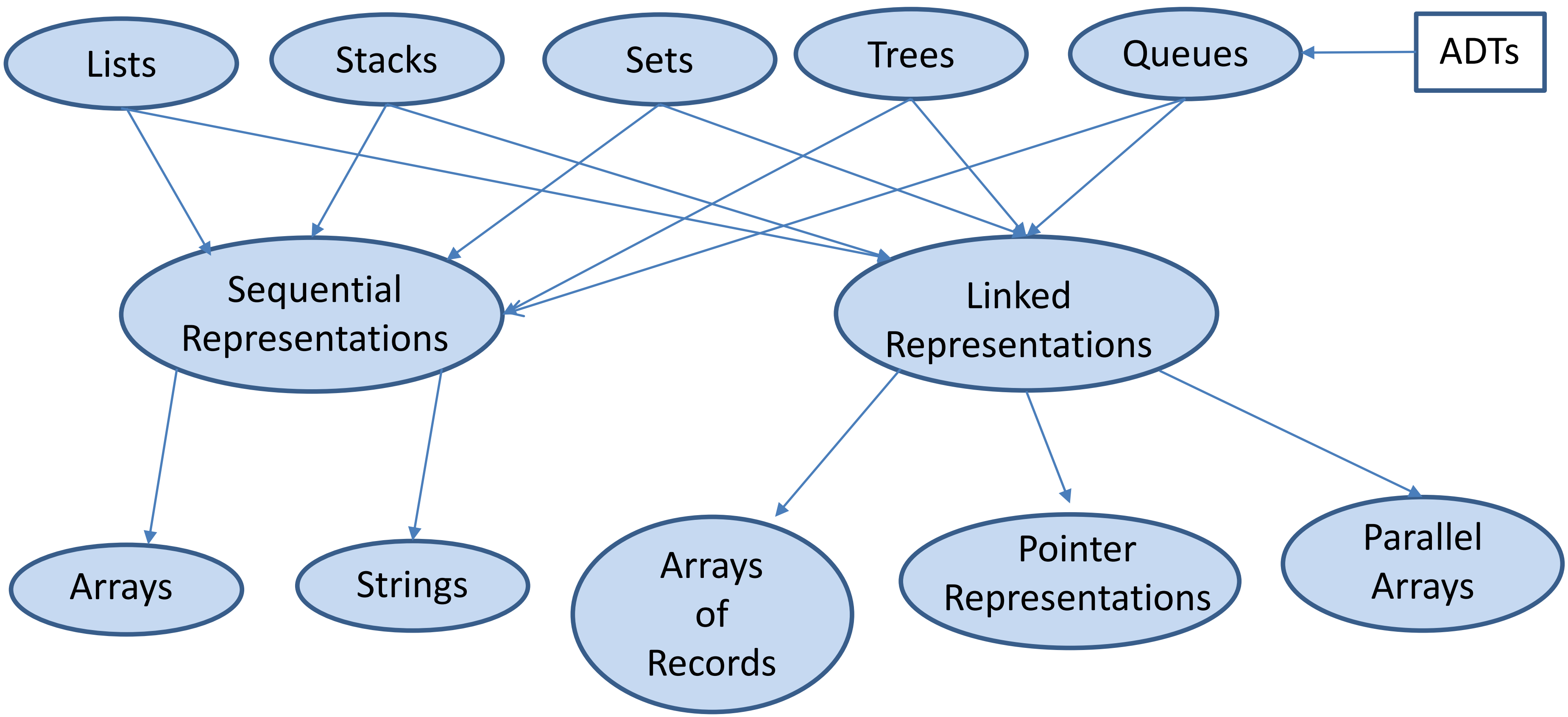
Linked Data Representations

Manolis Koubarakis

Linked Data Representations

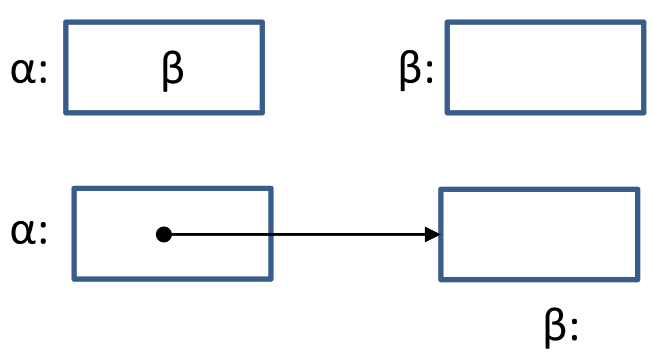
- Linked data representations such as **lists, stacks, queues, sets** and **trees** are very useful in Computer Science and applications. E.g., in Databases, Artificial Intelligence, Graphics, Web, Hardware etc.
- We will cover all of these data structures in this course.
- Linked data representations are useful when it is difficult to predict the size and shape of the data structures needed.

Levels of Data Abstraction



Pointers

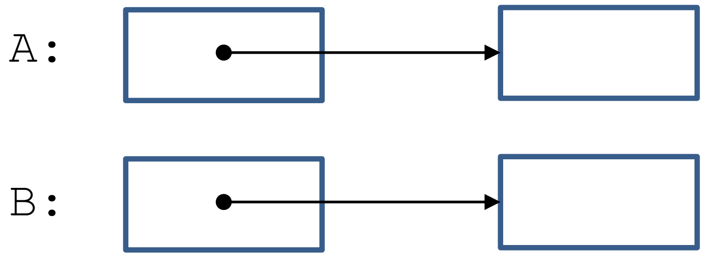
- The best way to realize linked data representations is using pointers.
- A **pointer (δείκτης)** is a variable that references a unit of storage.
- Graphical notation (α is a pointer to β):



Pointers in C

```
typedef int *IntegerPointer;
IntegerPointer A, B;
/* the declaration int *A, *B has the same effect */

A=(IntegerPointer)malloc(sizeof(int));
B=(int *)malloc(sizeof(int));
```

The above code results in the following situation:



`typedef`

- C provides a facility called `typedef` for creating new data type names.

- `typedef`s are useful because:
  - They help to organize our data type definitions nicely.
  - They provide better documentation for our program.

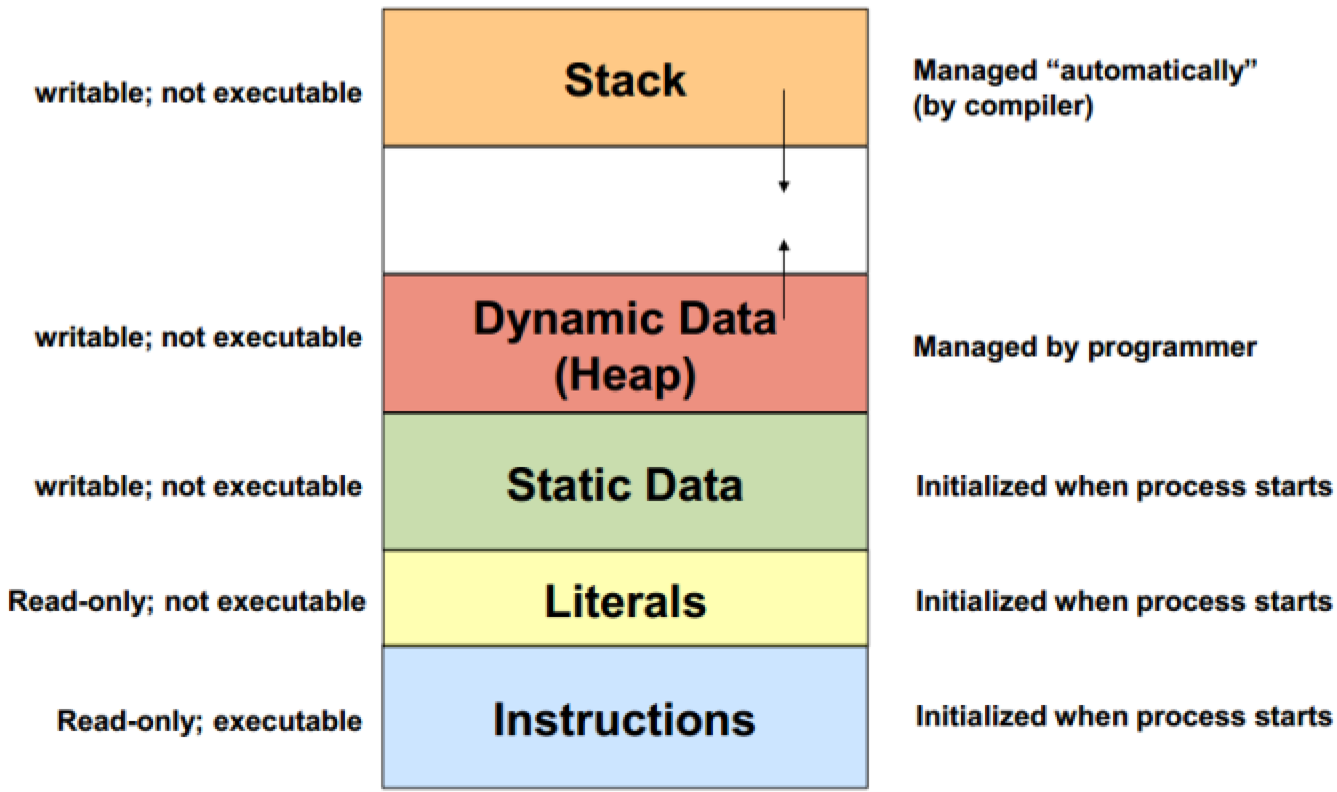
Pointers in C (cont'd)

- The previous statements first define a new data type name `IntegerPointer` which consists of a pointer to an integer.
- Then they define two variables `A` and `B` of type `IntegerPointer`.
- Then they allocate two blocks of storage for two integers and place two pointers to them in `A` and `B`.
- The `void` pointer returned by `malloc` is casted into a pointer to a block of storage holding an integer. **You can omit this casting (προσαρμογή) and your program will still work correctly** because the conversion to the required pointer type is done implicitly.

`malloc`

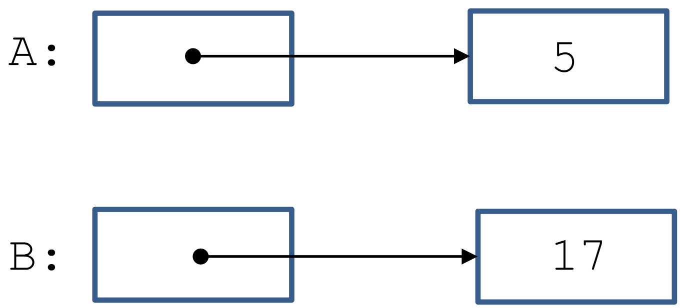
- `void *malloc(size_t size)` is a function of the standard library `stdlib`.
- `malloc` returns a pointer to space for an object of size `size`, or `NULL` if the request cannot be satisfied. The space is obtained from the **heap** and is uninitialized.
- This is called **dynamic storage allocation (δυναμική δέσμευση μνήμης).**
- `size_t` is the unsigned integer type returned by the `sizeof` operator.

Program Memory



The Operator *

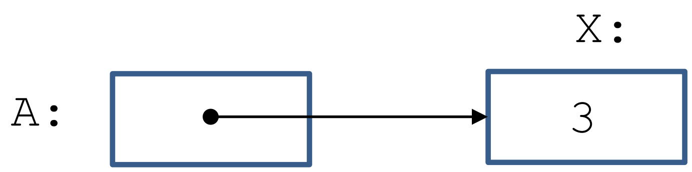
```
*A=5;
*B=17;
```



The unary operator `*` (**τελεστής αναφοράς**) on the left side of the assignment designates the storage location to which the pointer `A` refers. We call this **pointer dereferencing**.

The Operator &

```
int X=3;
A=&X;
```



The unary operator `&` (**τελεστής διεύθυνσης**) gives the address of some object (in the above diagram the address of variable `X`).

Pointers in C - Quiz (cont'd)

- Consider again the following statements:

```
int *A, *B;
*A=5;
*B=17;
```

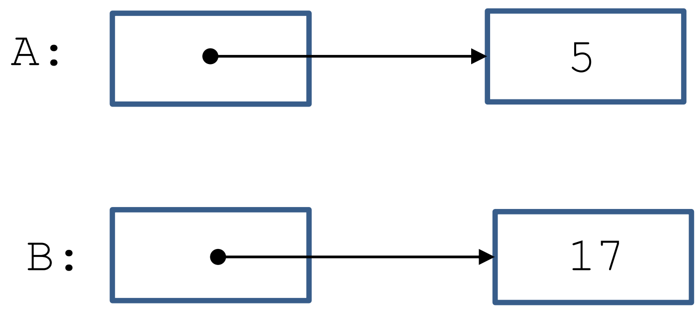
- **Question:** What happens if we now execute `B=20;`?

Pointers in C - Quiz (cont'd)

- **Answer:** We have a **type mismatch error** since `20` is an integer but `B` holds a pointer to integers.
- The compiler gcc will give a **warning**: "assignment makes pointer from an integer without a cast."

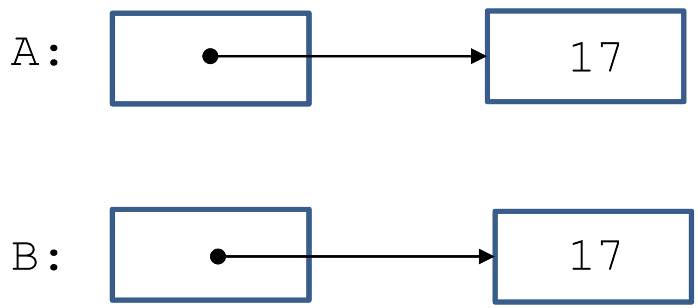
Pointers in C - Quiz (cont'd)

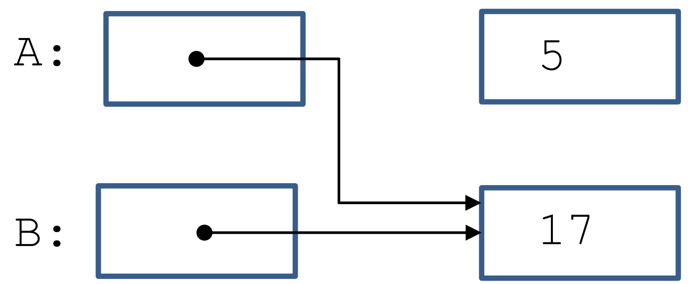
Suppose we start with the diagram below:



Pointers in C - Quiz (cont'd)

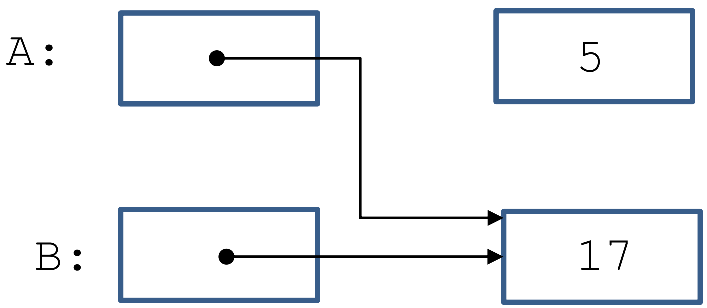
Question: If we execute `A=B;` which one of the following two diagrams results?





Pointers in C - Quiz (cont'd)

```
A=B;
```

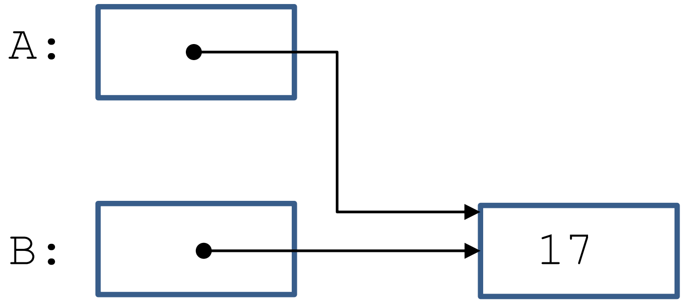


Answer: The right diagram. Now `A` and `B` are called **aliases** because they name the same storage location. Note that the storage block containing 5 is now **inaccessible**. Modern programming languages have a **garbage collection** facility for such storage.

Recycling Used Storage

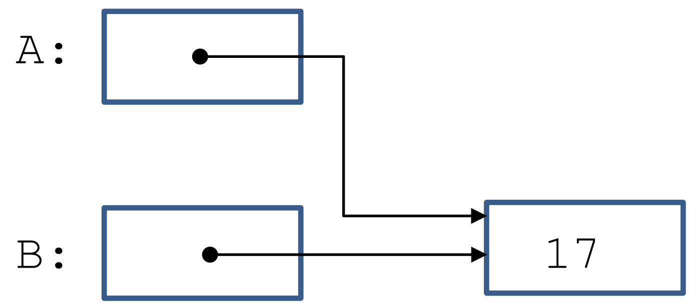
We can reclaim the storage space to which `A` points by using the **reclamation function** `free`:

```
free(A);
A=B;
```



Dangling Pointers

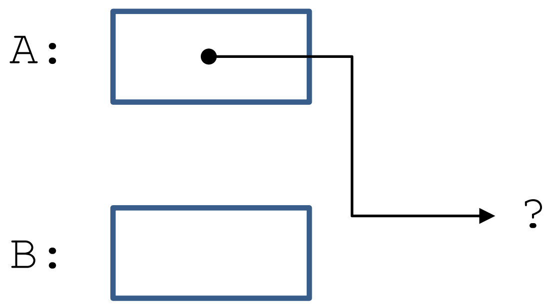
Let us now consider the following situation:



Question: Suppose now we call `free(B)`. What is the value of `*A+3` then?

Dangling Pointers (cont'd)

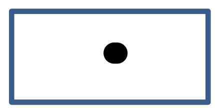
Answer: We do not know. Storage location `A` now contains a **dangling pointer** and should not be used.



It is reasonable to consider this to be a **programming error** even though the compiler or the runtime system will not catch it.

NULL

There is a special address denoted by the constant `NULL` which is not the address of any node. The situation that results after we execute `A=NULL;` is shown graphically below:

A:


Now we cannot access the storage location to which `A` pointed to earlier. So something like `*A=5;` will give us "**segmentation fault**".

`NULL` is automatically considered to be a value of any pointer type that can be defined in C. `NULL` is defined in the standard input/output library <stdio.h> and has the value `0`.

Pointers and Function Arguments

- Let us suppose that we have a **sorting algorithm** that works by **exchanging two out-of-order elements** `A` and `B` using a function `Swap`.

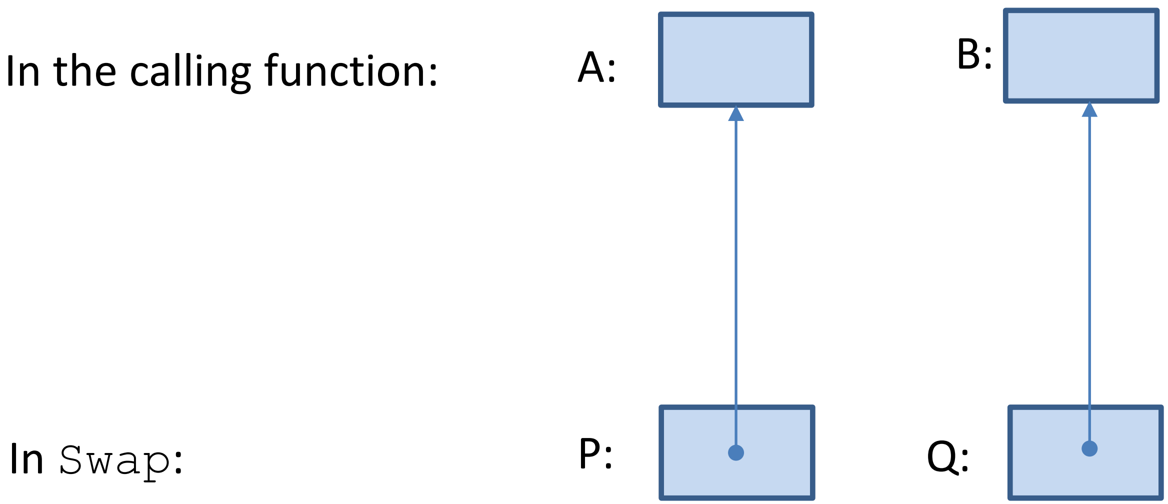
- **Question:** Can we call `Swap(A,B)` where the `Swap` function is defined as follows?

```
void Swap(int X, int Y)
{
   int Temp;

   Temp=X;
   X=Y;
   Y=Temp;
}
```

Pointers and Function Arguments (cont’d)

- **Answer:** No!
- Why?
  - Because C passes arguments to functions **by value (κατ’ αξία)** therefore `Swap` can’t affect the arguments `A` and `B` in the function that called it. `Swap` only swaps **copies** of `A` and `B`.

What we Need in Pictures



The Correct Function Swap

```
void Swap(int *P, int *Q)
{
    int Temp;

    Temp=*P;
    *P=*Q;
    *Q=Temp;
}
```

`Swap` uses the operator `*` to do the exchange of values.

Pointers and Function Arguments (cont'd)

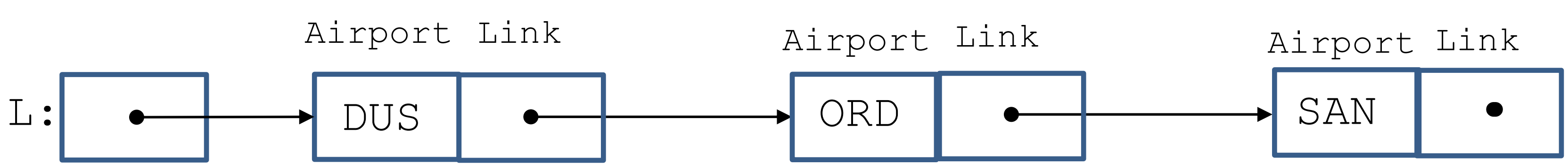
- The way to have the desired effect is for the calling function to pass **pointers** to the values to be changed:

```
Swap(&A,&B);
```

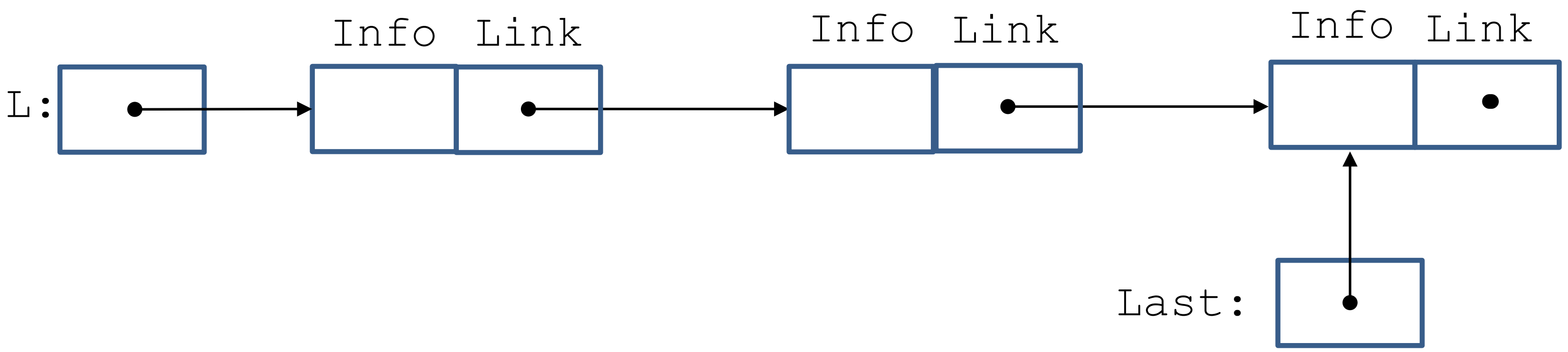
Linked Lists

- A **linked list** is a sequence of nodes in which each node, except the last, links to a successor node.
- We usually have a pointer variable `L` containing a pointer to the first node on the list.
- The link field of the last node contains `NULL`.

- **Example:** a list representing a flight



Diagrammatic Notation for Linked Lists



Declaring Data Types for Linked Lists

The following statements declare appropriate data types for our linked list:

```
typedef char AirportCode[4];
typedef struct NodeTag {
                        AirportCode Airport;
                        struct NodeTag *Link;
                } NodeType;
typedef NodeType *NodePointer;
```

We can now define variables of these datatypes:

```
NodePointer L;
```

or equivalently

```
NodeType *L;
```

Structures in C

- A **structure (δομή)** is a collection of one or more variables possibly of different types, grouped together under a single name.
- The variables named in a structure are called **members (μέλη).**
- In the previous structure definition, the name `NodeTag` is called a **structure tag** and can be used subsequently as a shorthand for the part of the declaration in braces.

Question

- Given the previous `typedef`s, what would be the output of the following piece of code:

```
AirportCode C;
NodePointer L;

strcpy(C, “BRU”);
printf(“%s\n”, C);

L=(NodePointer)malloc(sizeof(NodeType));
strcpy(L->Airport, C);
printf(“%s\n”, L->Airport);
```

Answer

```
BRU
BRU
```

The Function `strcpy`

- The function `strcpy(s,ct)` copies string `ct` to string `s`, including `'\0'`. It returns `s`.
- The function is defined in header file `<string.h>`.

Accessing Members of a Structure

- To access a member of a structure, we use the **dot notation** as follows:

  `structure-name.member`

- To access a member of a structure pointed to by a pointer `P`, we can use the notation `(*P).member` or the equivalent **arrow notation** `P->member`.

Question

- Why didn't I write `C="BRU";` and `L->Airport="BRU"` in the previous piece of code?

Answer

- The assignment `C="BRU";` assigns to variable `C` a **pointer to the character array** `"BRU"`. This would result in an error (type mismatch) because `C` is of type `AirportCode`.
- Similarly for the second assignment.

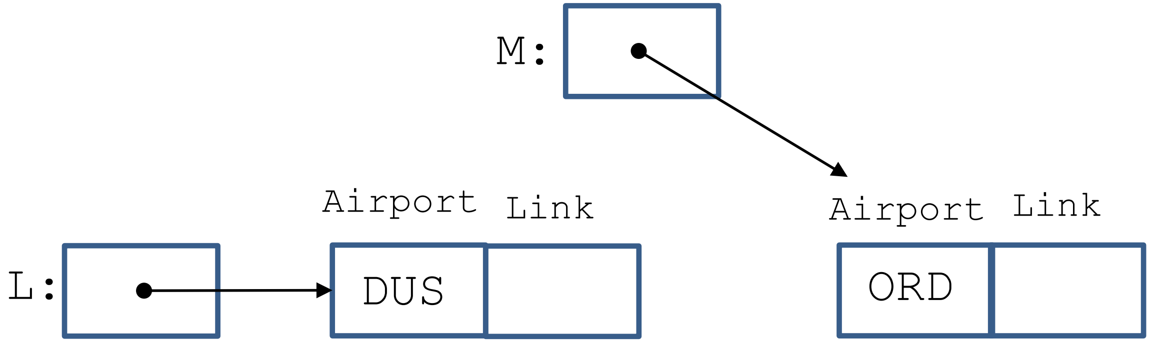
Question

- Given the previous `typedef`s, what does the following piece of code do?:

```
NodePointer L, M;

L=(NodePointer)malloc(sizeof(NodeType));
strcpy(L->Airport, "DUS");

M=(NodePointer)malloc(sizeof(NodeType));
strcpy(M->Airport, "ORD");

L->Link=M;
M->Link=NULL;
```

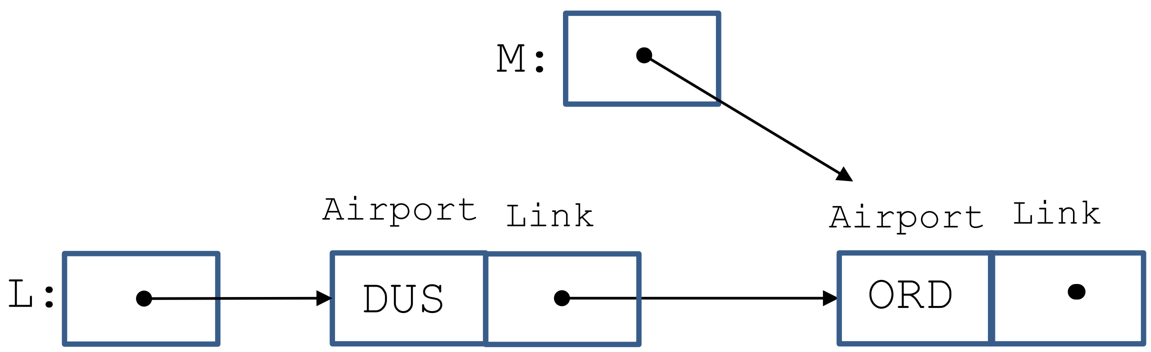
Answer

```
NodePointer L, M;
L=(NodePointer)malloc(sizeof(NodeType));
strcpy(L->Airport, "DUS");

M=(NodePointer)malloc(sizeof(NodeType));
strcpy(M->Airport, "ORD");
```



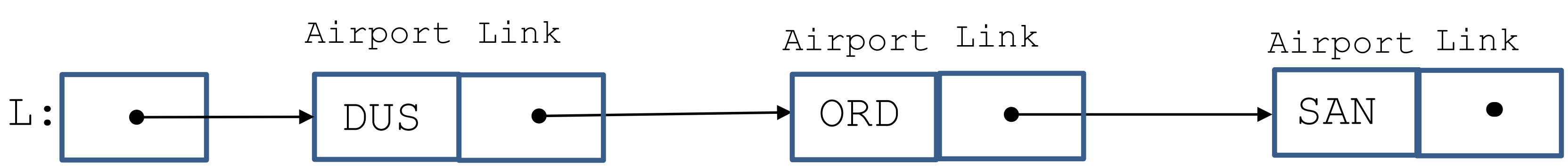
Answer (cont'd)

```
L->Link=M;
M->Link=NULL;
```



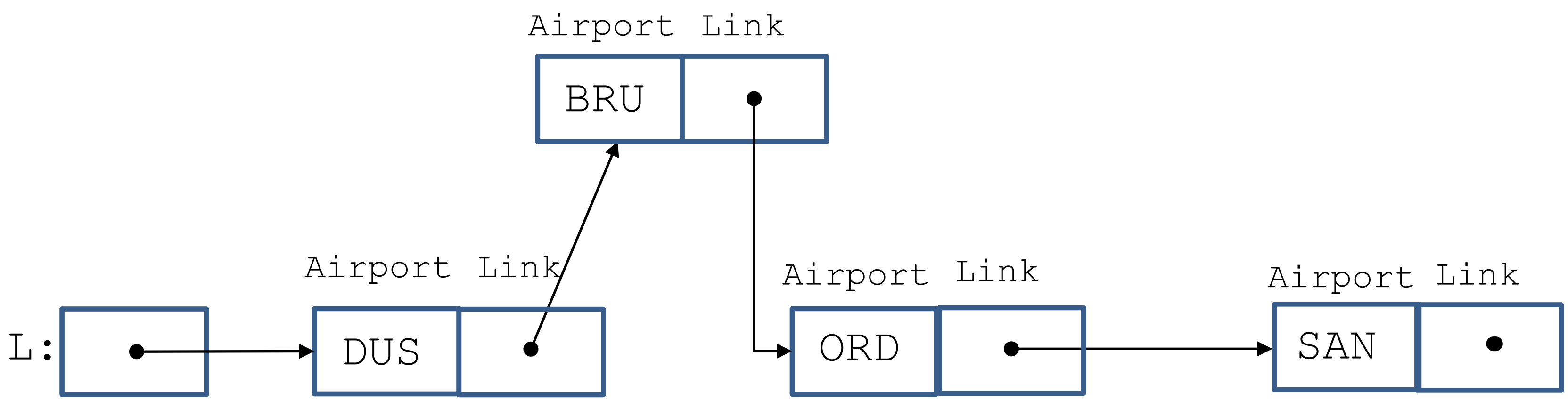
Inserting a New Second Node on a List

- **Example:** a list of three airports representing a flight



Inserting a New Second Node on a List (cont'd)

- **Example:** adding the airport `BRU` as second node



Inserting a New Second Node on a List (cont’d)

```
void InsertNewSecondNode(void)
{
      NodeType *N;
      N=(NodeType *)malloc(sizeof(NodeType));
      strcpy(N->Airport,"BRU");
      N->Link=L->Link;
      L->Link=N;
}
```

Inserting a New Second Node on a List (cont'd)

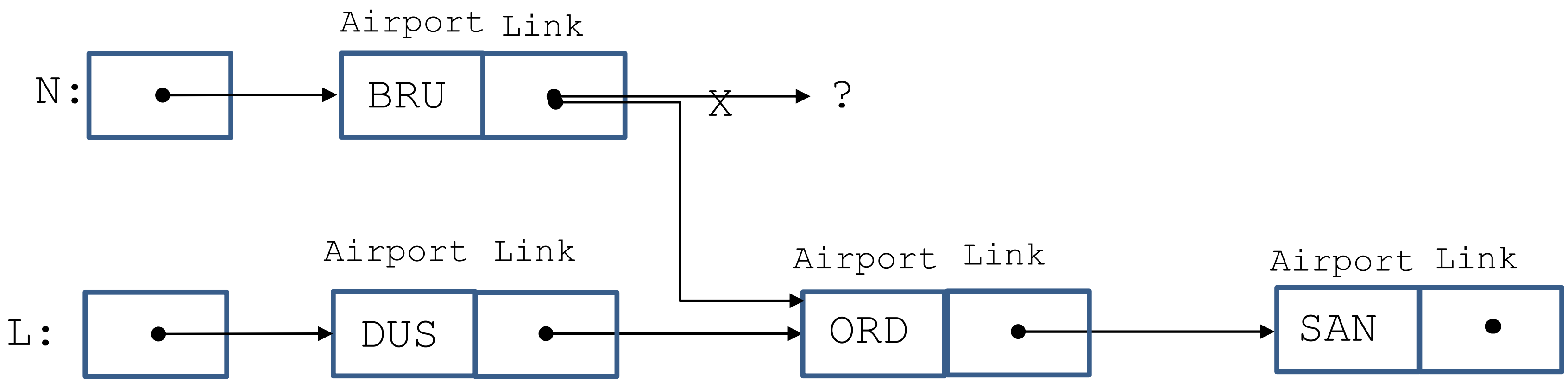
Let us execute the previous function step by step:

```
N=(NodeType *)malloc(sizeof(NodeType));
```


```
strcpy(N->Airport,"BRU");
```

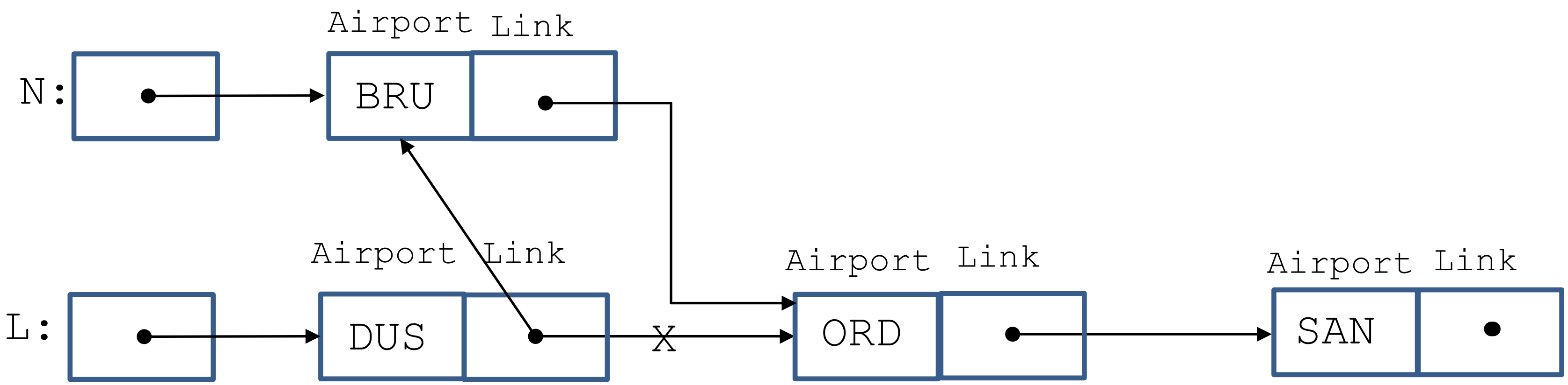
Inserting a New Second Node on a List (cont'd)

```
N->Link=L->Link;
```



Inserting a New Second Node on a List (cont'd)

```
L->Link=N;
```



Comments

- In the function `InsertNewSecondNode`, variable `N` is **local**. Therefore, it vanishes after the end of the function execution. However, **the dynamically allocated node remains in existence** after the function has terminated.

Searching for an Item on a List

- Let us now define a function which takes as input an airport code `A` and a pointer to a list `L` and returns a pointer to the first node of `L` which has that code. If the code cannot be found, then the function returns `NULL`.

Searching for an Item on a List

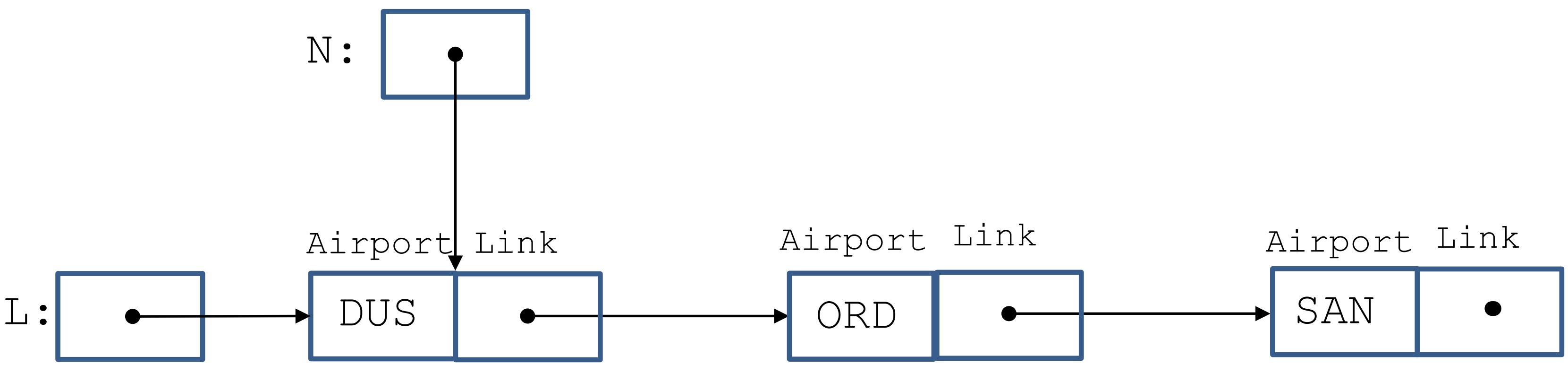
```
NodeType *ListSearch(char *A, NodeType *L)
{
      NodeType *N;

      N=L;
      while (N != NULL){
        if (strcmp(N->Airport,A)==0){
           return N;
        } else {
           N=N->Link;
        }
      }
      return N;
}
```

Comments

- The function `strcmp(cs,ct)` compares string `cs` to string `ct` and returns a negative integer if `cs` precedes `ct` alphabetically, `0` if `cs==ct` and a positive integer if `cs` follows `ct` alphabetically (using the ASCII codes of the characters of the strings).

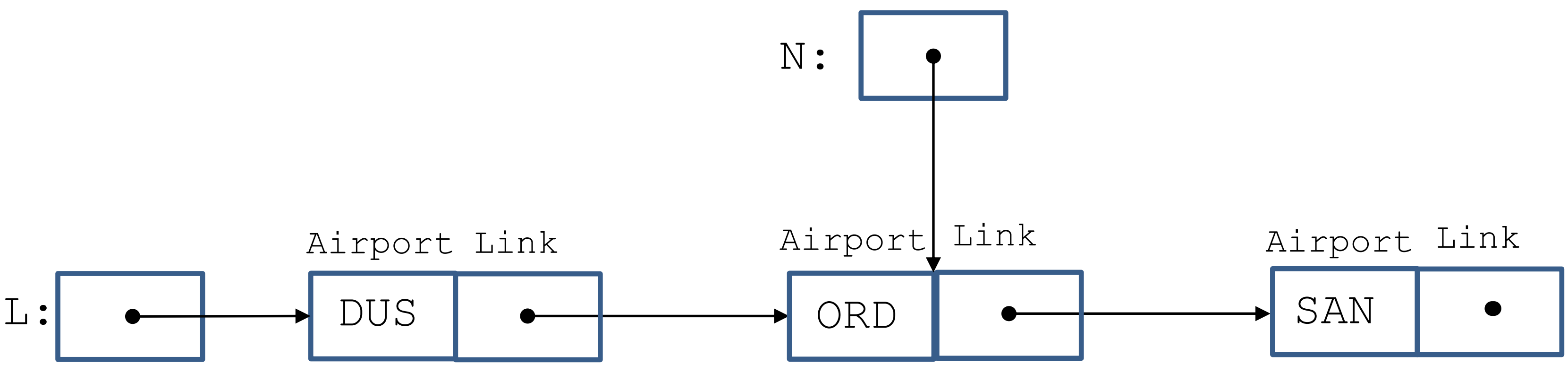
Comments (cont'd)

- Let us assume that we have the list below and we are searching for item "`ORD`". When the initialization statement `N=L` is executed, we have the following situation:



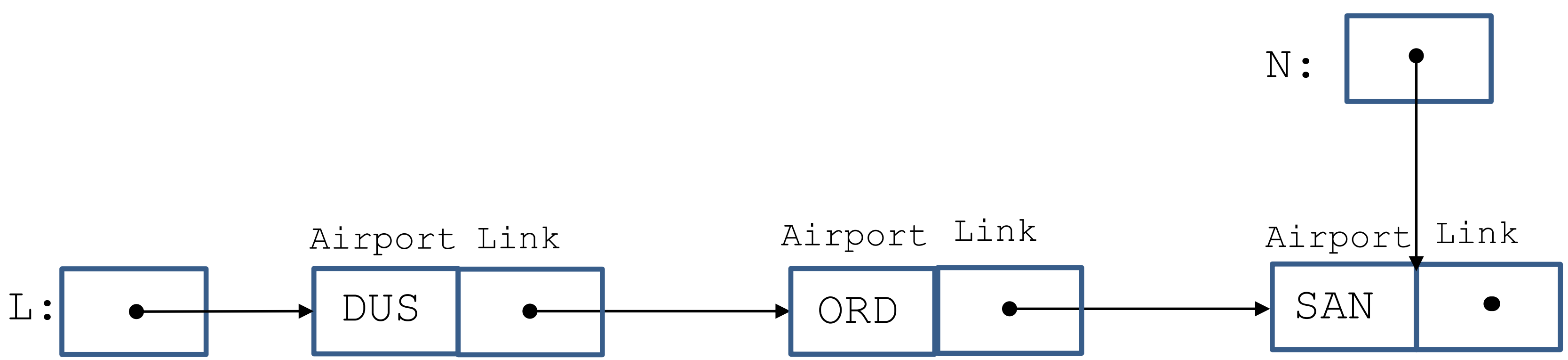
Comments (cont'd)

- Later on, inside the `while` loop, the statement `N=N->Link` is executed, and we have the following situation:



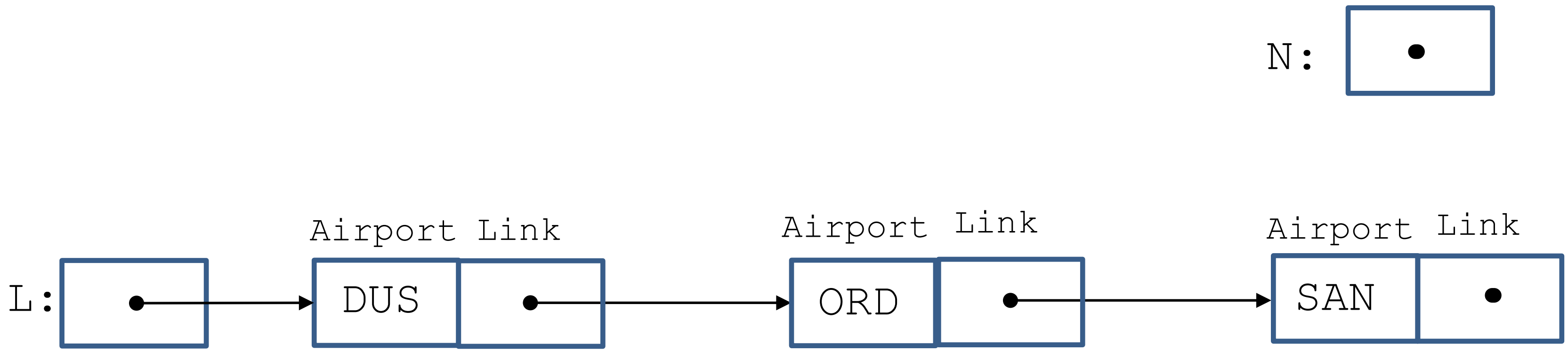
Comments (cont'd)

- Then, the `if` inside the `while` loop is executed and the value of `N` is returned. Assuming that we did not find `"ORD"` here, the statement `N=N->Link` is again executed, and we have the following situation:



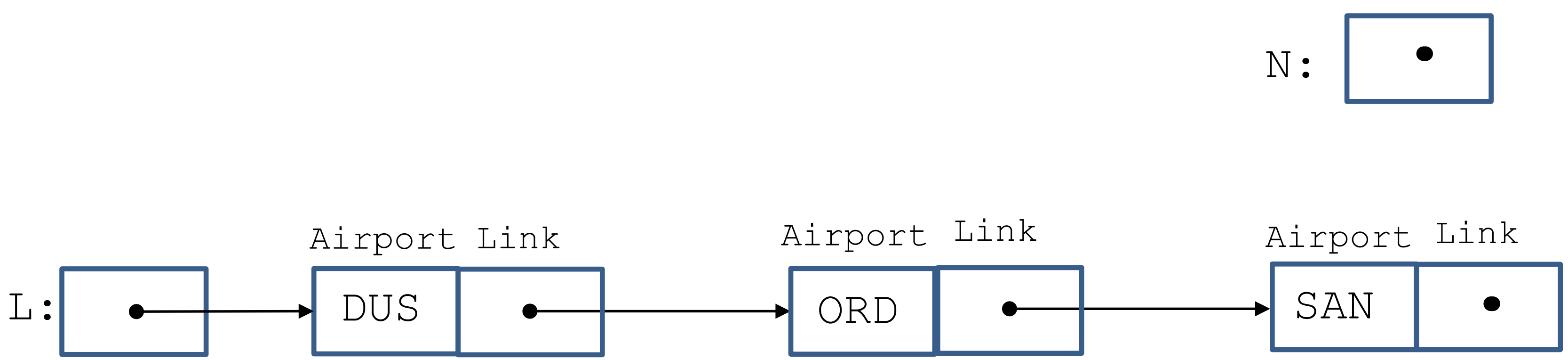
Comments (cont'd)

- Then, the `while` loop is executed one more time and the statement `N=N->Link` results in the following situation:



Comments (cont'd)

- Then, we exit from the `while` loop and the statement `return N` returns `NULL`:



Deleting the Last Node of a List

- Let us now write a function to delete the last node of a list `L`.
- If `L` is **empty**, there is nothing to do.
- If `L` has **one node**, then we need to dispose of the node's storage and then set `L` to be the empty list.
- If `L` has **two or more nodes,** then we can use a pair of pointers to implement the required functionality as shown on the next slides.

Question

- Is the following function definition appropriate?

```
void DeleteLastNode(NodeType *L)
```

Deleting the Last Node of a List (cont’d)

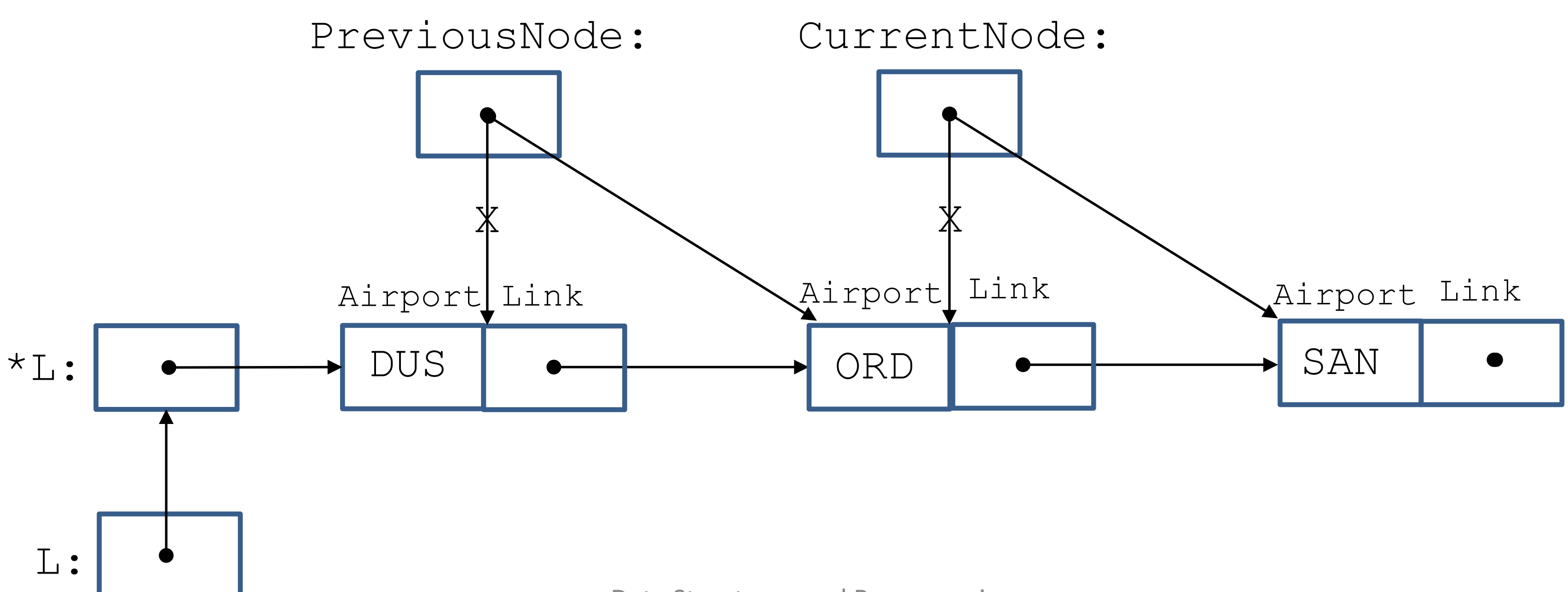
- No!
- Assume that in the main program, we have a list and `L1` is the pointer to its first element.
- We need to pass the **address** of (the pointer to the first element of the list) `L1`, for which we want to delete the last node, as an actual parameter in the form of `&L1` enabling us to **change the contents of `L1`** inside the function `DeleteLastNode`.
- Therefore, the corresponding formal parameter of the function `DeleteLastNode` should be a **pointer to a pointer** to `NodeType`.

Deleting the Last Node of a List

```
void DeleteLastNode(NodeType **L)
{
      NodeType *PreviousNode, *CurrentNode;

      if (*L != NULL) {
         if ((*L)->Link == NULL){
            free(*L);
            *L=NULL;
         } else {
            PreviousNode=*L;
            CurrentNode=(*L)->Link;
            while (CurrentNode->Link != NULL){
               PreviousNode=CurrentNode;
               CurrentNode=CurrentNode->Link;
            }
            PreviousNode->Link=NULL;
            free(CurrentNode);
         }
      }
}
```

Comments

- When we advance the pointer pair to the next pair of nodes, the situation is as follows:



Why **?

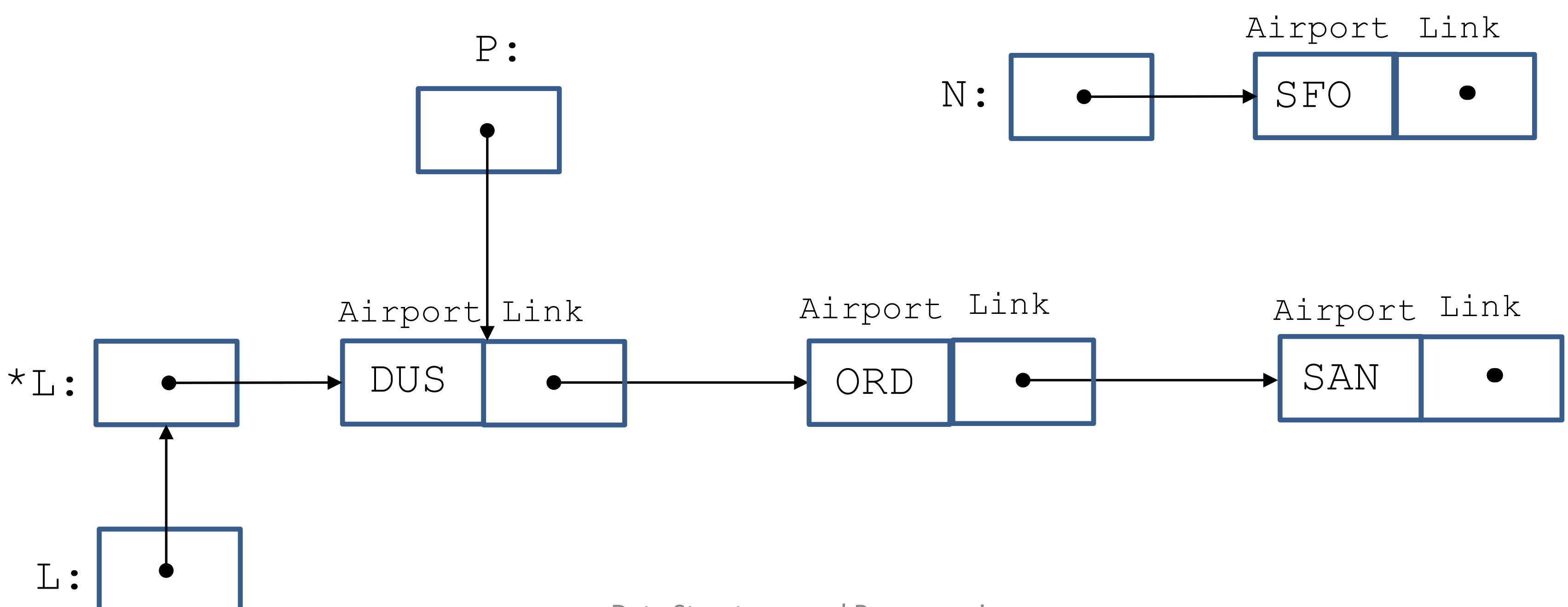
- This is for the case that the list in the calling function has one node only.
- Then, the value of pointer (e.g., `L1`) to the only element of that list must be set to `NULL` in the function `DeleteLastNode`.
- This can only be done by passing `&L1` in the call of the function `DeleteLastNode`.

Inserting a New Last Node on a List

```
void InsertNewLastNode(char *A, NodeType **L)
{
      NodeType *N, *P;

      N=(NodeType *)malloc(sizeof(NodeType));
      strcpy(N->Airport, A);
      N->Link=NULL;

      if (*L == NULL) {
         *L=N;
      } else {
         P=*L;
         while (P->Link != NULL) P=P->Link;
         P->Link=N;
      }

}
```

Comments

- `P` is used to move across the list until we find the last node.



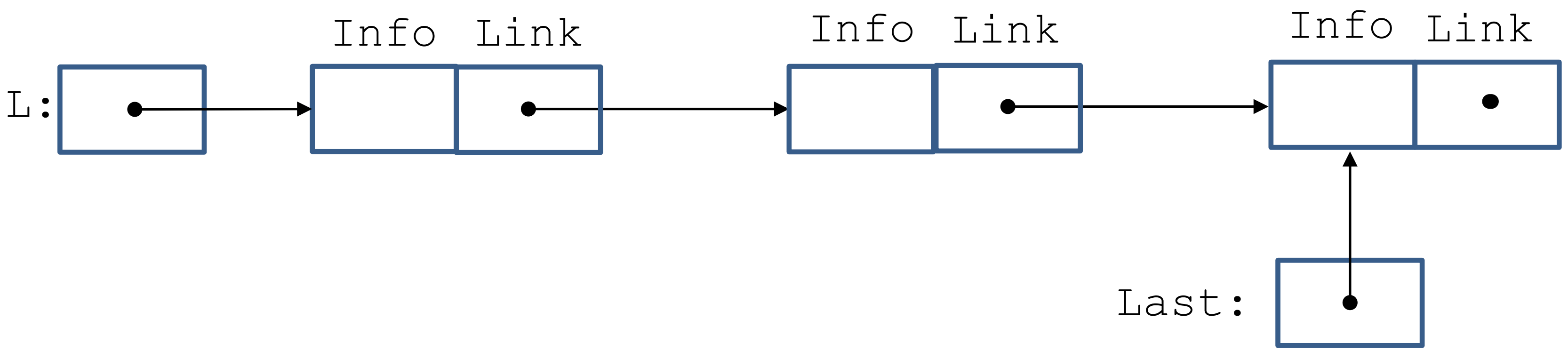
Why **?

- This is for the case that the list in the calling function is empty.
- Then, the value of pointer (e.g., `L1`) to the first element of that list must be set to point to the new node created in the function `InsertNewLastNode`.
- This can only be done by passing `&L1` in the call of the function `InsertNewLastNode`.

Question

- Assume now that we have a pointer `Last` pointing to the last element of a linked list.
- How would the operations of deleting the last node of a list or inserting a new last node on a list change to exploit the pointer `Last`?

Question (cont'd)



Printing a List

```
void PrintList(NodeType *L)
{
      NodeType *N;

      printf("(");
      N=L;
      while(N != NULL) {
         printf("%s", N->Airport);
         N=N->Link;
         if (N!=NULL) printf(",");
      }
      printf(")\n");
}
```

Examples

```
()
(ATH)
(ATH, FRA, JFK, SFO)
```

The Main Program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef char AirportCode[4];
typedef struct NodeTag {
                        AirportCode Airport;
                        struct NodeTag *Link;
                } NodeType;
typedef NodeType *NodePointer;

/* function prototypes */
void InsertNewLastNode(char *, NodeType **);
void DeleteLastNode(NodeType **);
NodeType *ListSearch(char *, NodeType *);
void PrintList(NodeType *);
```

The Main Program (cont’d)

```
int main(void)
{
      NodeType *L;

      L=NULL;

      PrintList(L);

      InsertNewLastNode(“DUS”, &L);
      InsertNewLastNode(“ORD”, &L);
      InsertNewLastNode(“SAN”, &L);
      PrintList(L);


      DeleteLastNode(&L);
      PrintList(L);

      if (ListSearch(“DUS",L) != NULL) {
         printf(“DUS is an element of the list\n");
      }
}

/* Code for functions InsertNewLastNode, PrintList,  */
/* ListSearch and DeleteLastNode goes here. */
```

Linked Lists vs. Arrays

- Compare the data structure linked list that we defined in these slides with arrays.
- What are the pros and cons of each data structure?

Linked Lists vs. Arrays

- The **simplicity of inserting and deleting a node anywhere** is what characterizes linked lists. This operation is more involved in an array because all the elements of the array that follow the affected element need to be moved.
- Linked lists are **not appropriate for finding the *i*-th element of a list** because we have to follow *i* pointers. In an array, the same functionality is implemented with one operation.
- **Such discussion is important** when we want to choose a data structure for solving a practical problem.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*

  Chapter 2.
- (προαιρετικά) R. Sedgewick. *Αλγόριθμοι σε C*. Κεφάλαιο 3.