Linked Data Representations
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

Sequential Representations
- Lists
- Stacks
- Sets
- Trees
- Queues

Linked Representations
- Arrays
- Strings
- Arrays of Records
- Pointer Representations
- Parallel Arrays

ADTs
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 (A is a pointer to X):

```
A: &X  X:  
```

```
A:  X:  
```
Pointers in C

```c
int X;
int *A = &X;
```

The above code results in the following situation:
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.

• *typedefs* are useful because:
  – They help to organize our data type definitions nicely.
  – They provide better documentation for our program.
  – They make our program portable.
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 and your program will still work correctly.
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

- **Stack**: Managed “automatically” (by compiler)
- **Dynamic Data (Heap)**: Managed by programmer
- **Static Data**: Initialized when process starts
- **Literals**: Initialized when process starts
- **Instructions**: Initialized when process starts
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.
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 (cont’d)

• Consider again the following statements:

```c
int X = 10;
int *A = &X;
```

• **Question:** What happens if we now execute
  ```c
  A=20;  
  ```
Pointers in C (cont’d)

• **Answer:** We have a **type mismatch error** since 20 is an integer but \( A \) holds a pointer to integers.

• gcc will give a **warning** (at **compile time**): "assignment makes pointer from an integer without a cast."

• Any use of \( *A \) will (most likely) cause a **segmentation fault** at **runtime**
Pointers in C (cont’d)

Suppose we start with the diagram below:

A:  

B:  

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

A:  

B:  

A:  

B:  

Data Structures and Programming Techniques
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. Some languages such as Lisp 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`:

```c
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.
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:

![Diagram of A: NULL]

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

- We want to swap two variables using a function `Swap`.
- **Question:** Does this work?
  ```c
  int A = 1, B = 2;
  Swap(A, B);
  ```

  ```c
  void Swap(int X, int Y)
  {
      int Temp;
      Temp=X;
      X=Y;
      Y=Temp;
  }
  ```
Pointers and Function Arguments (cont’d)

• **Answer:** No. C passes arguments to functions by value (κατ’ αξία) therefore \( \text{Swap} \) can’t affect the arguments \( A \) and \( B \) in the routine that called it. \( \text{Swap} \) only swaps copies of \( A \) and \( B \).

In the calling program:  

\[
\begin{align*}
A & : 1 \\
B & : 2
\end{align*}
\]

In \( \text{Swap} \):  

\[
\begin{align*}
X & : 2 \\
Y & : 1
\end{align*}
\]
The Correct Function Swap

• Solution: pass **pointers** to the values to be changed:

```c
int A = 1, B = 2;
Swap(&A, &B);

void Swap(int *P, int *Q)
{
    int Temp;
    Temp=*P;
    *P=*Q;
    *Q=Temp;
}
```
int A = 1, B = 2;
Swap(&A, &B);

In the calling program:

In `Swap`:

A: \( \rightarrow 2 \)

B: \( \rightarrow 1 \)

P:  

Q:  
Linked Lists

- A **linear linked list** (or **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

L: 

Data Structures and Programming Techniques
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.
Example

• Given the previous `typedefs`, what would be the output of the following piece of code:

```c
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);
```
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:
  
  \[
  \text{structure-name.member}
  \]

- 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** \( P->\text{member} \).
Question

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

• The assignment \( C = "\text{BRU}"; \) assigns to variable \( C \) a pointer to the character array "\text{BRU}". This would result in an error (type mismatch) because \( C \) is of type AirportCode.

• Similarly for the second assignment.
Example

Given the previous typedefs, what does the following piece of code do?:

```c
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

• The piece of code on the previous slide constructs the following linked list of two elements:
Inserting a New Second Node on a List

• **Example**: adding one more airport to our list

we want to create:

```
L: DUS Ord
```

```
L: DUS Ord San
```
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
Inserting a New Second Node on a List (cont’d)

Let us execute the previous function step by step:

\[ N = (\text{NodeType} *) \text{malloc}(\text{sizeof(NodeType)}); \]

\[ \text{strcpy}(N->\text{Airport}, \text{"BRU"}); \]

Data Structures and Programming Techniques
Inserting a New Second Node on a List (cont’d)

\[ N \rightarrow \text{Link} = L \rightarrow \text{Link}; \]

```
N:  B     R     U
   X
L:  D   U   S  \rightarrow  O   R   D  \rightarrow  S   A   N
```

Data Structures and Programming Techniques
Inserting a New Second Node on a List (cont’d)

L->Link=N;

N: [BRU]

L: [DUS] X [ORD] [SAN]
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 $\textit{A}$ and a pointer to a list $\textit{L}$ and returns a pointer to the first node of $\textit{L}$ which has that code. If the code cannot be found, then the function returns $\texttt{NULL}$. 
Searching for an Item on a List

```c
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;
}
```
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:

```
L: [Airport, Link, SAN, Link, Airport, ORD]
N: [ ]
```

Data Structures and Programming Techniques
Comments (cont’d)

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

```
L: DUS -> ORD -> SAN
N:  
```

Data Structures and Programming Techniques
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:

\[ L: \text{Airport Link} \quad \text{DUS} \rightarrow \text{ORD} \rightarrow \text{SAN} \quad N: \text{ORD} \]
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`:
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");
}

**Question:** can we use L instaed of N in the while loop?
Inserting a New Last Node on a List

• Write a function to **insert a new last node** in a list \( X \).

• Algorithm
  — find the **current last node** \( P \)
  — create a **new node** \( N \)
  — set \( P \rightarrow \text{link} = N \)

• **But**: what if \( X \) is empty? (\( X == \text{NULL} \))
Inserting a New Last Node on a List

• We need to change X inside our function, so we need the **address** of X as an actual parameter \( L = &X \).
• Therefore the corresponding formal parameter \( L \) of the function will be a **pointer to a pointer to** \( \text{NodeType} \)

```c
NodeType *X = NULL;
InsertNewLastNode("ATH", &X);
void InsertNewLastNode(char *A, NodeType **L) {
    ...
}
```
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;
    }
}
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 (again we need \( \text{NodeType}** \)).
Deleting the Last Node of a List

- If \( L \) has **two or more nodes** then we need to get both the last node and its previous one.
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);
        }
    }
}
Why **?**

- This is for the case that $L$ has one node only.
- Then, the value of pointer $L$ must be set to NULL in the function `DeleteLastNode`.
- This can only be done by passing $&L$ in the call of the function.
The Main Program

```c
#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 *);
```
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. */
Question

• Assume now that we have a pointer Tail 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 Tail?
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 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


• (προαιρετικά) R. Sedgewick. *Αλγόριθμοι σε C.* Κεφάλαιο 3.