Departamento de Informática
Universidad de Valladolid
Campus de Segovia

TOPIC 3: DYNAMIC DATA STRUCTURES.

LINKED LISTS, STACKS AND QUEUES

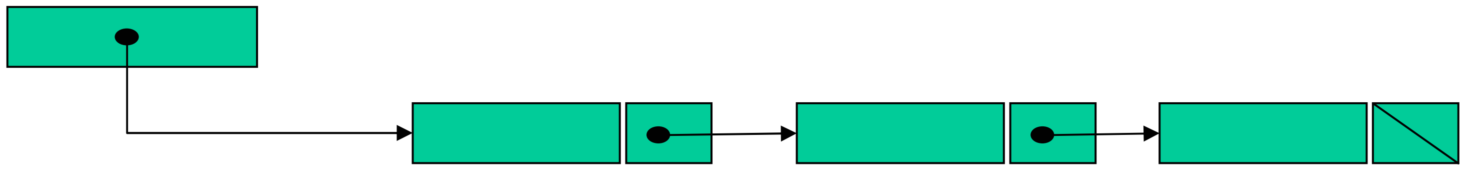
INDEX

WHAT IS A LINKED LIST?

- A linked (or linear) list is a data structure that stores a collection of objects of a certain type, usually denoted as nodes (or elements).
- The nodes are ordered in a linear sequence. It means that except the first one, the other nodes have a predecessor one.
- The number of nodes can change along a process, increasing (by insertion) or decreasing (by deletion) according to the necessities.
- Their implementation is achieved using pointers and dynamic variables (pointer-based data structure) .
- Linked lists are usually simply denoted as *lists.*

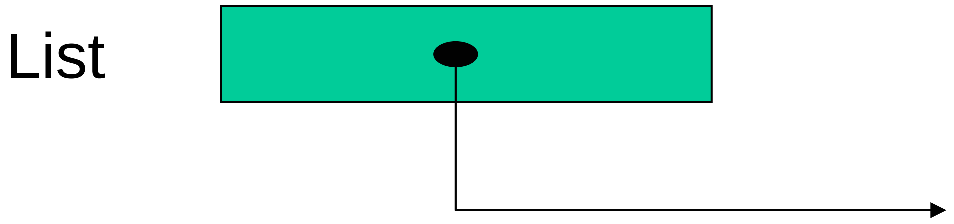
A LIST IMPLEMENTATION USING POINTERS AND RECORDS IN PASCAL

- Each node is represented by a record of at least two fields. In that case, the first one stores a piece of information of a certain type and the second one is a pointer that will point to the next node if that exists and NIL if not.



A LIST IMPLEMENTATION USING POINTERS AND RECORDS IN PASCAL

```
TYPE
tElem=<type>
PNode=^tNode
tNode=record
    info:tElem;
    next:PNodo;
End; {tNode}

VAR
list:PNode;
```



BASIC LIST OPERATION

- There are a set of basic operation for manipulating lists. These group of operations are related to the concept of abstract data type that will be studied later.

BASIC LIST OPERATION

- **Createlist(List):** initializes **List** to empty state.
- **Emptylist(List):** Determines whether **List** is empty.
- **Insertfirst(x,List):** Inserts in the first position of **List** a node containing “x”.
- **Insert(x,P,List):** Inserts in **List** a node containing “x” there where P is pointing.
- **Insertlast(x,List):** Inserts in the last position of **List** a node containing “x”.
- **Find(x,List):** Returns a pointer pointing to the first node of **List** that contains “x” if this one exists and NIL if not.
- **Exist(x,List):** Determines whether a node, containing the “x” piece of information, exists on **List.**

(“x” is piece of information of a certain type).

BASIC LIST OPERATION

- **Delete(x,List):** Deletes a node of **List** containing “x”, releasing the corresponding allocated memory space.
- **Deleteaddress(P,List):** Deletes the node of **List** that is pointed by “P”, releasing the corresponding allocated memory space.
- **Next(P,List):** Given a pointer “P” pointing to a node of **List,** this function returns another one that points to the next node.
- **Previous(P,List):** Given a pointer “P” that is pointing to a node of **List,** this function returns another one that points the previous node.
- **Lastnode(List):** Returns a pointer pointing to the last position of **List**.
- **Empty(List):** Empties **List** releasing all the allocated memory space.
- **View(List):** Shows all the contents of **List.**

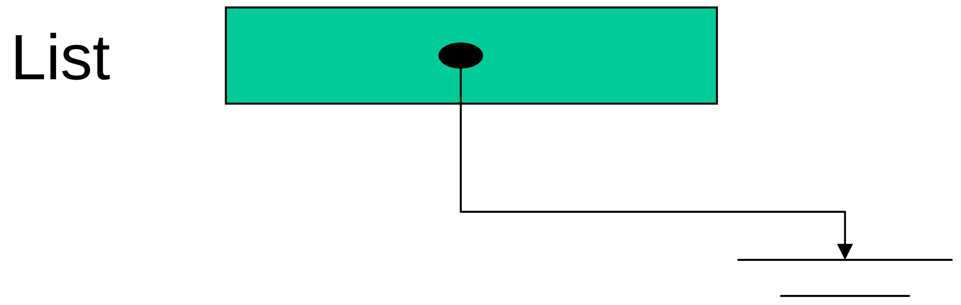
(“x” is piece of information of a certain type).

INITIALIZING A LIST

- The basic initialization operations are:
  - **Createlist(List):** Procedure that initializes **List** to empty state.
  - **Emptylist(List):** Boolean function that Determines whether **List** is empty.

PROCEDURE CREATELIST IMPLEMENTED IN PASCAL

```
PROCEDURE Createlist( var List:PNode);
    BEGIN
          List:=nil
    END; {Createlist}
```



FUNCTION EMPTYLIST IMPLEMENTED IN PASCAL

```
FUNCTION Emptylist(L:PNode):boolean;
   BEGIN
         Emptylist:=(List=nil)
   END; {Emptylist}
```

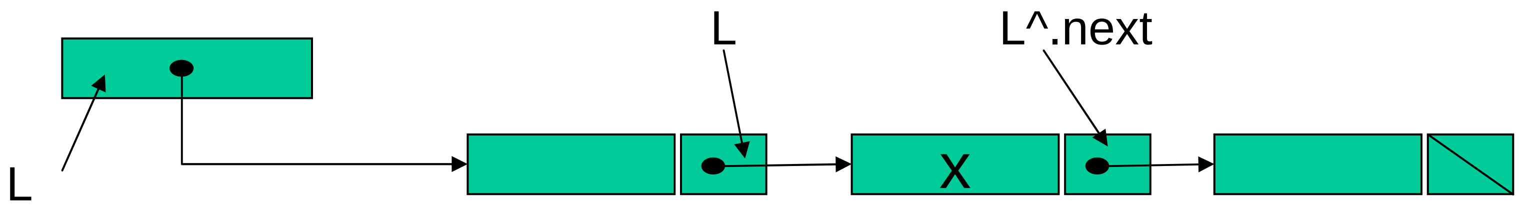
A SEARCH ON A LIST

- The basic search operations are:
  - **Find(x,List):**Returns a pointer pointing to the first node of **List** that contains “x” if this one exists and NIL if not.

  - **Exist(x,List):** Determines whether a node, that contains “x”, exists on **List.**

FUNCTION FIND IMPLEMENTED IN PASCAL

```
FUNCTION Find (x:tElem;List:PNode):PNode;
    {Returns a pointer pointing to the first node of List that contains “x” if
    this one exists and NIL if not.}

BEGIN
    WHILE (List^.next<>nil) AND (List^.info<>x) DO
            List:=List^.next; {move to the next position}
    IF List^.info<>x THEN
            Find:=nil
    ELSE
            Find:=L
END; {Find}
```



FUNCTION EXIST IMPLEMENTED IN PASCAL

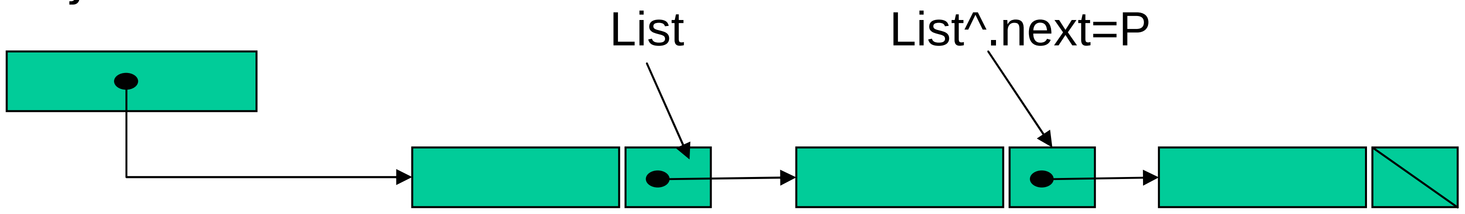
```
FUNCTION Exist (x:tElem;List:PNode):Boolean;
{Returns true if a node containing “x” exists and false if not}

BEGIN
    IF not Emptylist(List) THEN BEGIN
        WHILE (List^.next<>nil) AND (List^.info<>x) DO
          List:=List^.next; {move to the next position}
          Exist:=(List^.info=x)
        END; {IF}
    ELSE
         Exist:=false
    END; {Exist}
```

SUPPORTING OPERATION

- Using these kind of operations a particular node of a list can be located by means of pointers.
  - **Previous(P,List):** Given a pointer “P” pointing to a node of **List,** this function returns another pointer pointing to the previous node.
  - **Next(P,List):** Given a pointer “P” pointing to a node of **List,** this function returns another pointer pointing to the next node.
  - **Lastnode(List):** Returns a pointer pointing to the last position of **List .**

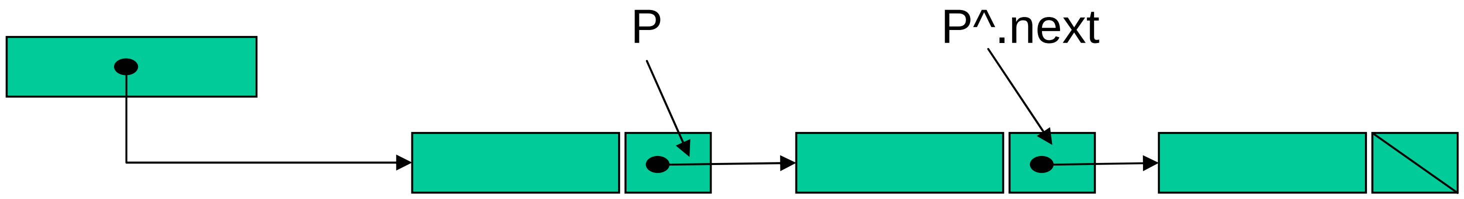
FUNCTION PREVIOUS IMPLEMENTED IN PASCAL

```
FUNCTION Previous (P,List:PNode):PNode;
    {Given a pointer “P”  pointing to a node of List,  this function returns
    another one pointing to the previous node if this node exist and NIL if it
    doesn´t exist or if List is empty or if (List=P)}.
BEGIN
    IF Emptylist(List)  OR (List=P) THEN
          Previous:=nil
    ELSE BEGIN
       WHILE (List^.next<>P) AND (List^.next<>nil) DO
          List:=List^.next; {move to the next position}
       IF List^.next=P THEN
          Previous:=List
       ELSE
          Previous:=nil
    END {ELSE}
END; {Previous}
```



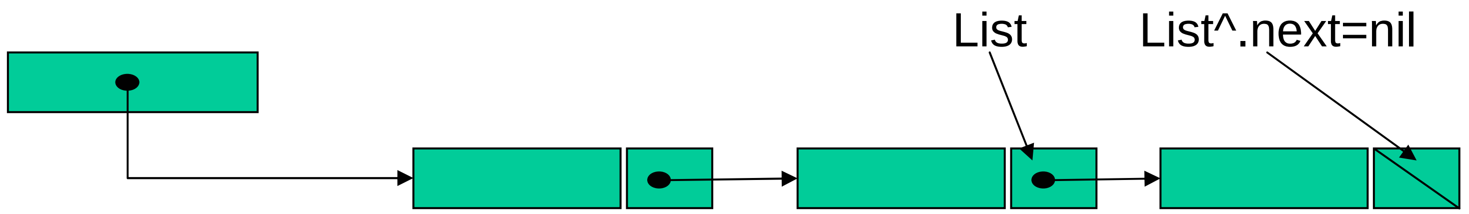
FUNCTION NEXT IMPLEMENTED IN PASCAL

```
FUNCTION Next (P,List:PNode):PNode;
   {Given a pointer "P"  pointing to a node of List,  this function returns
    another one pointing to the next node if List is not empty and NIL if it is
    empty or if (P=Nil)}
BEGIN
    IF Emptylist(List)  OR (P=Nil) THEN
          Next:=nil
    ELSE
          Next:=P^.next
END; {Next}
```



FUNCTION LASTNODE IMPLEMENTED IN PASCAL

```
FUNCTION LastNode (List:PNode):PNode;
   {Returns NIL if List is empty and a pointer pointing to the
   last position of List if not}
BEGIN
   IF Emptylist(List)  THEN
        LastNode:=nil
   ELSE  BEGIN
      WHILE  (List^.next<>Nil) DO
         List:=List^.next; {move to the next position}
         LastNode:=List
     END {ELSE}
END; {LastNode}
```

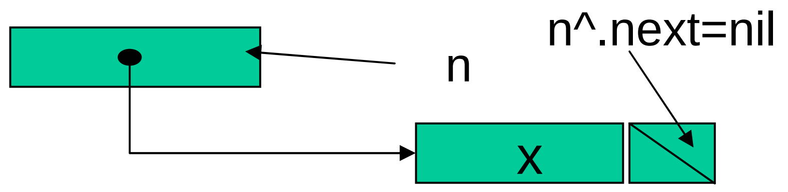


INSERTION NODE OPERATIONS

- These operations add new elements to the list and
- involves to create a node containing the piece of information and after finding a particular position on the list and inserting the node there.
  - **CreateNode(x):** Creates a node containing “x” and returns a pointer pointing to the node.
  - **Insertfirst(x,List):** Inserts a node containing “x” in the first position of **List.**
  - **Insert(x,P,List):** Inserts in **List** a node containing “x” there where P is pointing.
  - **Insertlast(x,List):** Inserts a node containing “x” in the last position of **List**

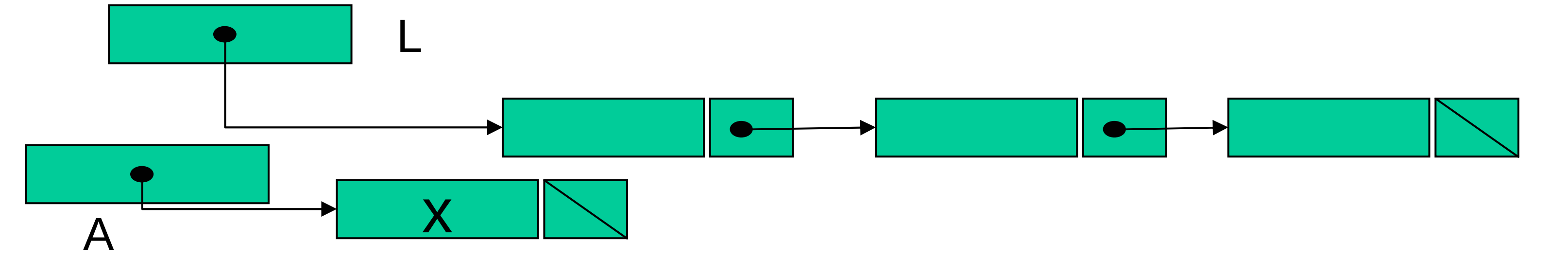
FUNCTION CREATENODE IMPLEMENTED IN PASCAL

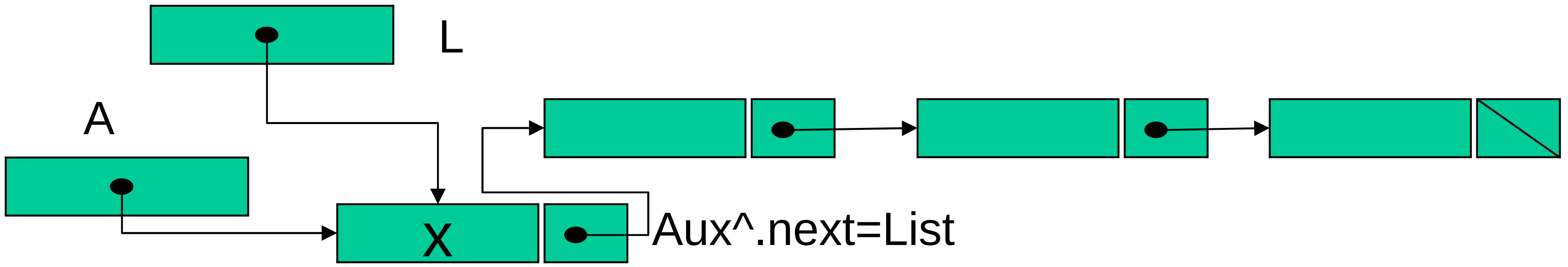
```
FUNCTION CreateNode (x:tElem):PNode;
{Creates a node containing “x” and returns a pointer pointing to this node}
VAR
    n:PNode;
BEGIN
    New(n);
    n^.info:=x;
    n^.next:=nil;
    CreaNode:=n
END; {CreateNode}
```



PROCEDURE INSERTFIRST IMPLEMENTED IN PASCAL

```
PROCEDURE Insertfirst (x:tElem; var List:PNode);
{Inserts a node containing “x” in the first position of List.}
VAR
    Aux:PNode;
BEGIN
    Aux:=CreateNode(x);
    Aux^.next:=List;
    List:=Aux
END; {Insertfirst}
```



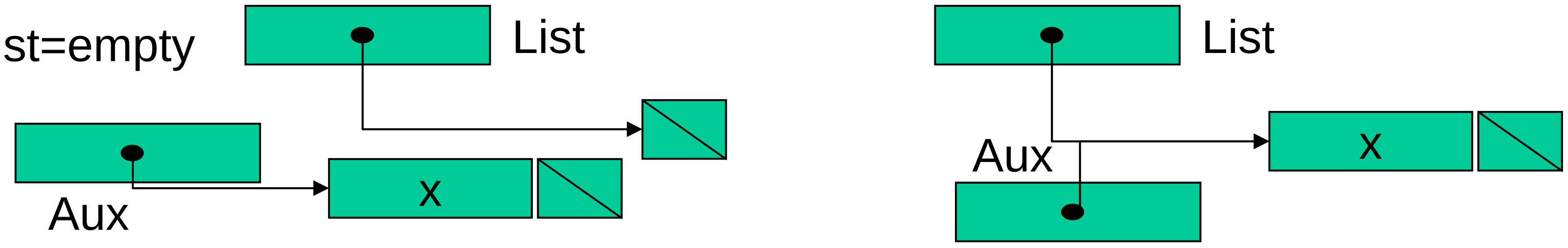


PROCEDURE INSERT IMPLEMENTED IN PASCAL

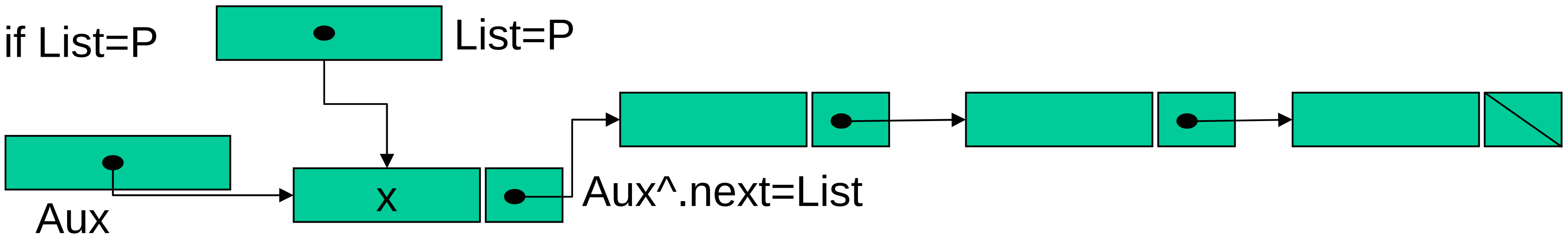
```
PROCEDURE Insert (x:tElem; P:PNode; var List:PNode);
{Inserts in List a node containing “x” there where P is pointing. }
VAR
    Aux:PNode;
BEGIN
    Aux:=CreateNode(x);
    IF  Emptylist(List) THEN
        List:=Aux
        ELSE IF P=List THEN BEGIN
          Aux^.next:=P;
          Lixt:=Aux
        END {Else if}
          ELSE BEGIN
             Anterior(P,List)^.next:=Aux;
             Aux^.next:=P
          END {Else}
END; {Insert}
```

PROCEDURE INSERT

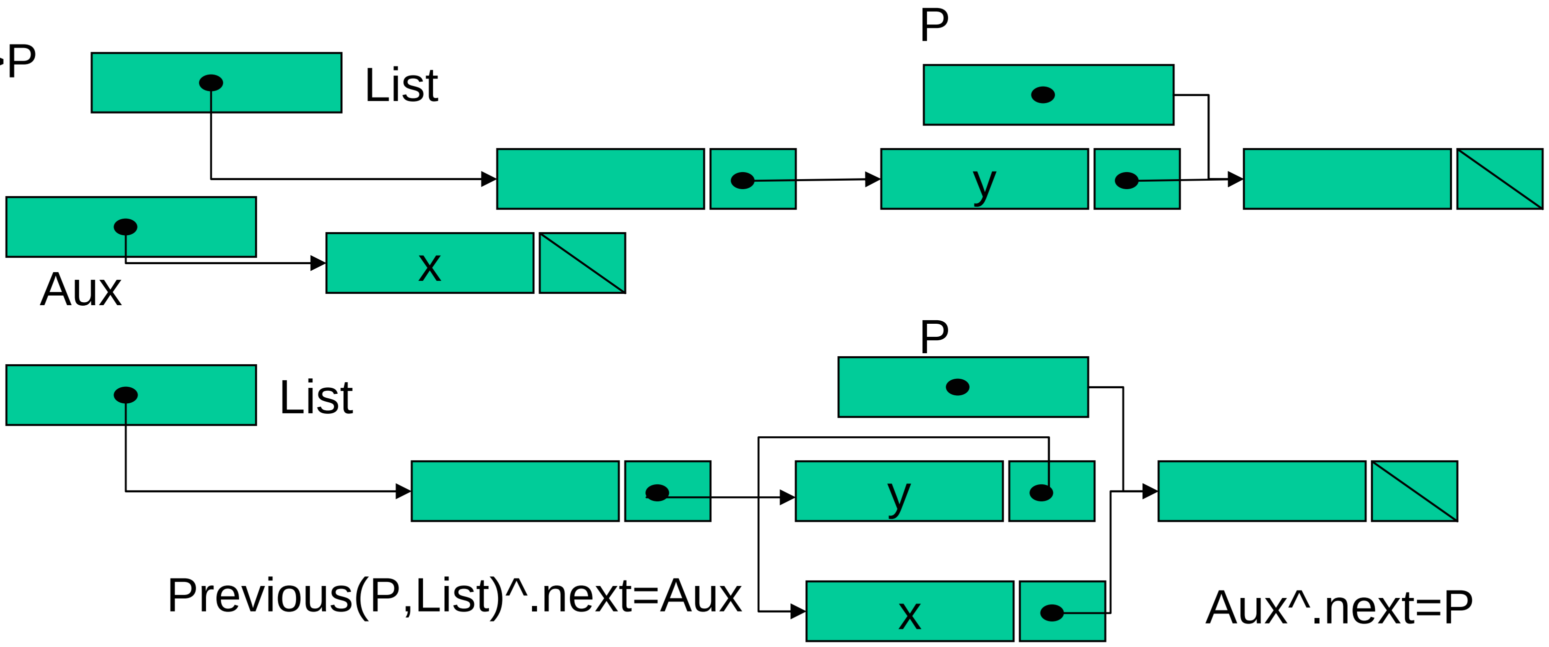
if List=empty



if List=P



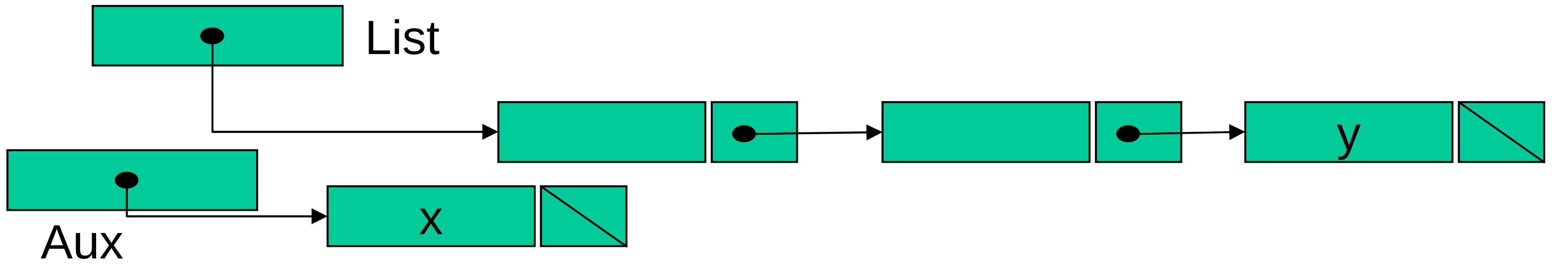
if List<>P

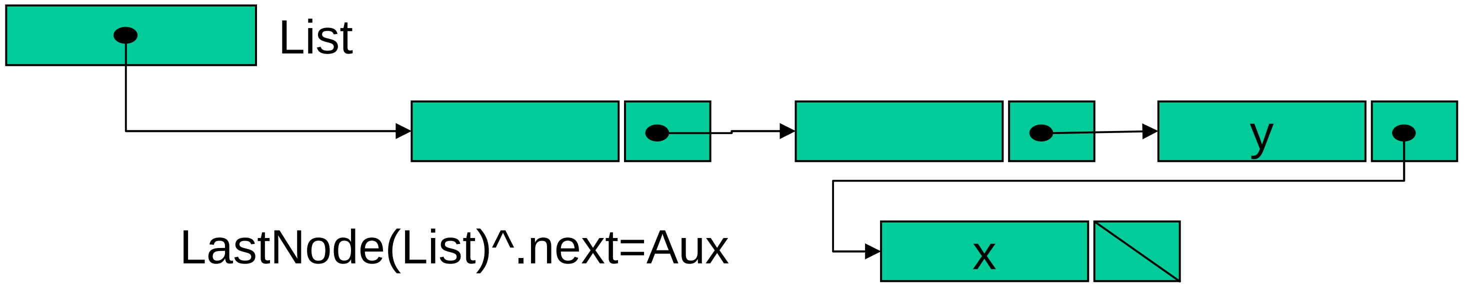


PROCEDURE INSERTLAST IMPLEMENTED IN PASCAL

```
PROCEDURE InsertLast (x:tElem;  var L:PNode);
{Inserts a node containing “x” in the last position of List}
VAR
    Aux:PNode;
BEGIN
    A:=CreateNode(x);
    IF  Emptylist(L) THEN
        List:=Aux
      ELSE
         LastNode(List)^.next:=Aux;
END; {InsertLast}
```

PROCEDURE INSERTLAST





LastNode(List)^.next=Aux

DELETION NODE OPERATIONS

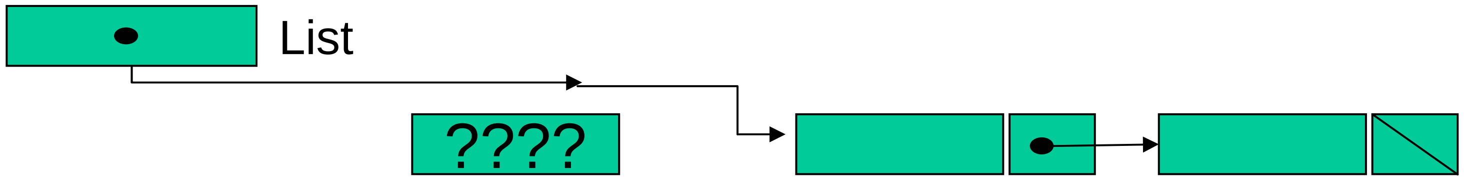
- These operations remove elements of a list releasing the allocated memory space.
- Involves to find the node in the list, remove it and finally link the previous node to the next one.
  - **Delete(x,List):** Deletes a node of **List** containing “x”.
  - **Deleteaddress(P,List):** Deletes the node of **List** that is pointed by “P”.
  - **Empty(List):** Empties **List,** releasing all allocated memory space.

PROCEDURE DELETE IMPLEMENTED IN PASCAL

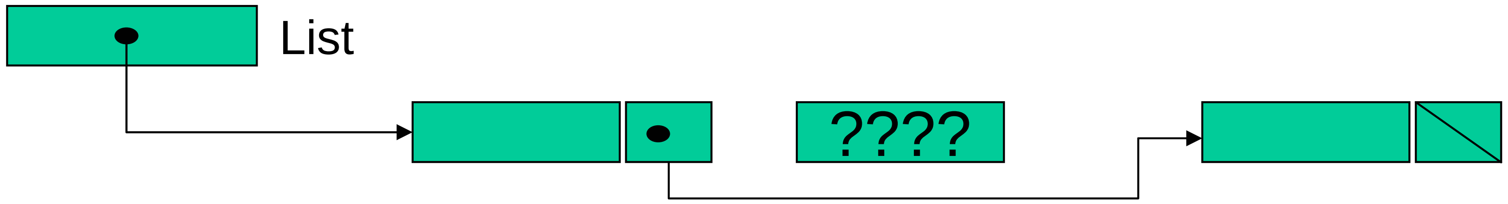
```
PROCEDURE Delete (x:tElem;  var List:PNode);
{Deletes a node of List containing “x”}
VAR
    Aux:PNode;
BEGIN
    Aux:=Find(x,List);
    IF  Aux<>nil THEN BEGIN
          IF Aux=List THEN {if List points to first node}
                    List:=List^.next
           ELSE
                    Previous(Aux,List)^.next:=Aux^.next;
          Dispose(Aux)
    END {If}
END; {Delete}
```

PROCEDURE DELETE

If Aux points to first node of List)



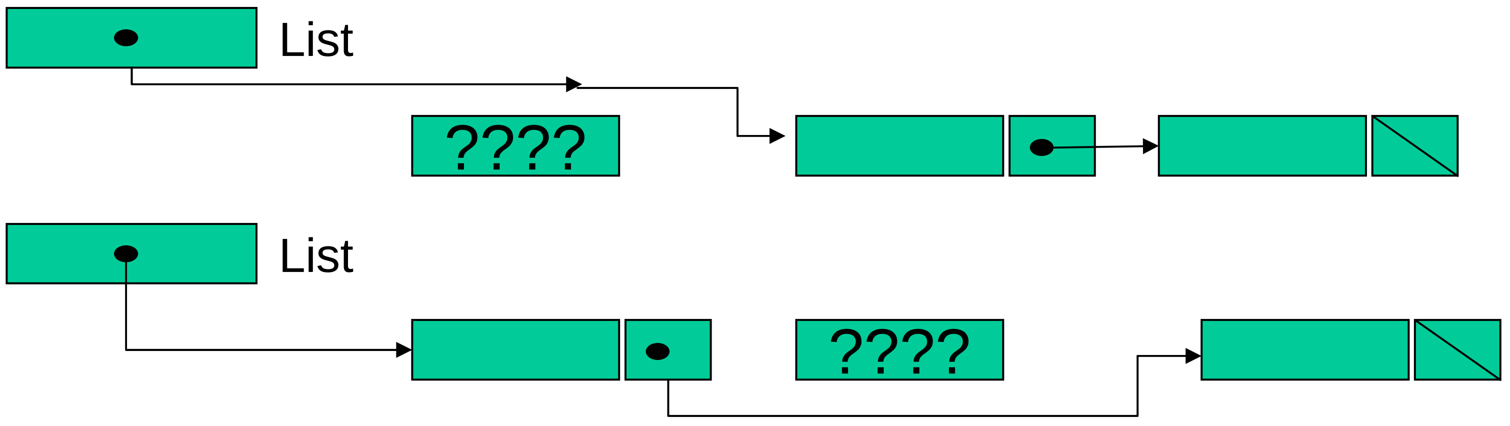
if Aux points to any other node of List)



PROCEDURE DELETEADDRESS IMPLEMENTED IN PASCAL

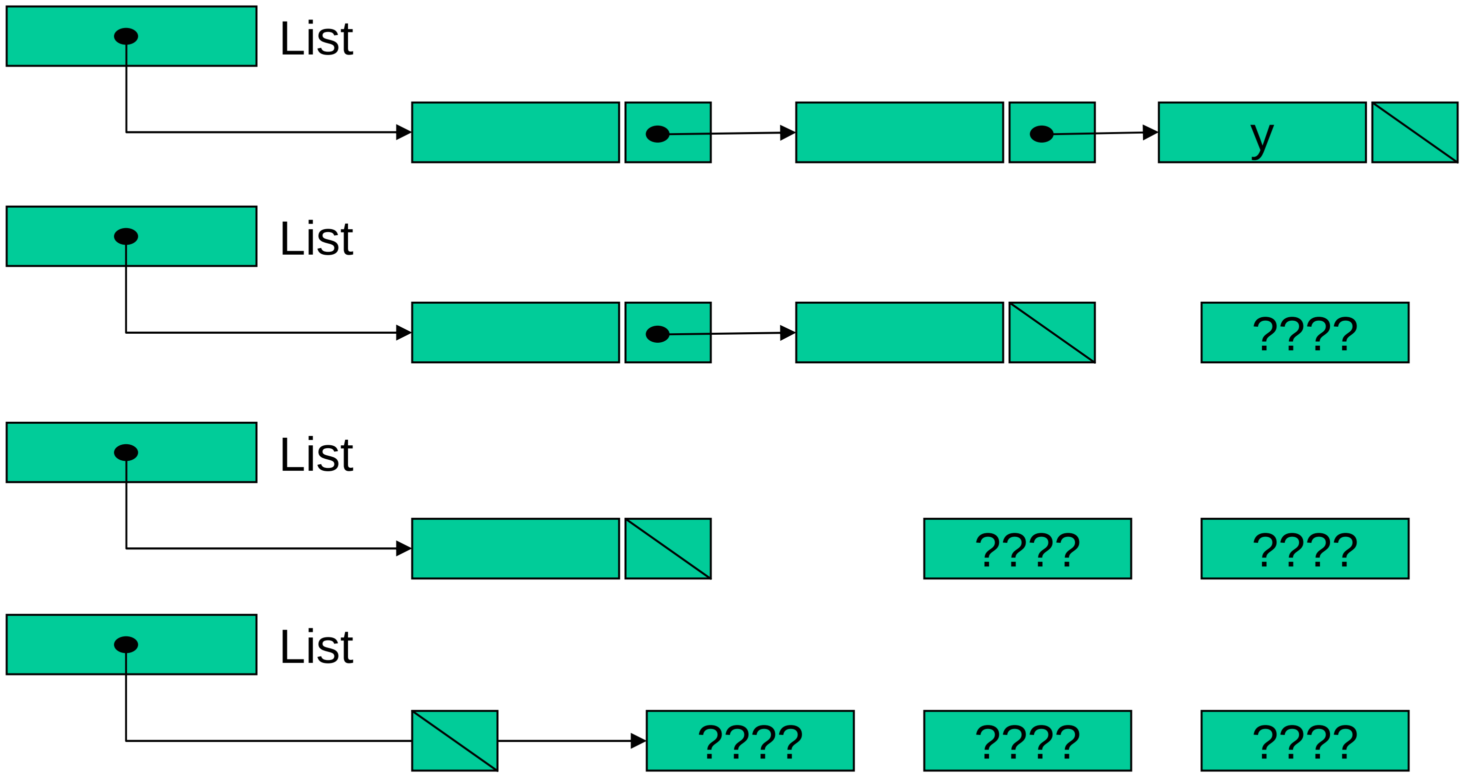
```
PROCEDURE DeleteAddress(P:PNode;  var List:PNode);
{Deletes the node of List that is pointed by “P”.}
BEGIN
    IF P=List THEN BEGIN {First Node of List}
           List:=List^.next;
           Dispose(P)
    END {If}
           ELSE
               IF  Previous(P,List)<>nil THEN BEGIN
                     Previous(P,List)^.next:=Next(P,List);
                     Dispose(P)
               END {Else if}
END; {DeleteAddress}
```

PROCEDURE DELETEADDRESS



PROCEDURE EMPTY IMPLEMENTED IN PASCAL

```
PROCEDURE Empty (var List:PNode);
{Empties List, releasing all the allocated memory space}
BEGIN
   WHILE not Emptylist(List)  DO
         DeleteAddress(LastNode(List),List)
END; {Anula}
```



SHOWING ALL THE CONTENTS OF A LIST

– **View(List):** Shows all the contents in a List

PROCEDURE VIEW IMPLEMENTED IN PASCAL

```
PROCEDURE View ( List:PNode);
{Shows up all the contents in List.}
BEGIN
    WHILE List<>nil DO BEGIN
            write(List^.info,' ');
            List:=List^.next
    END {While}
END; {View}
```

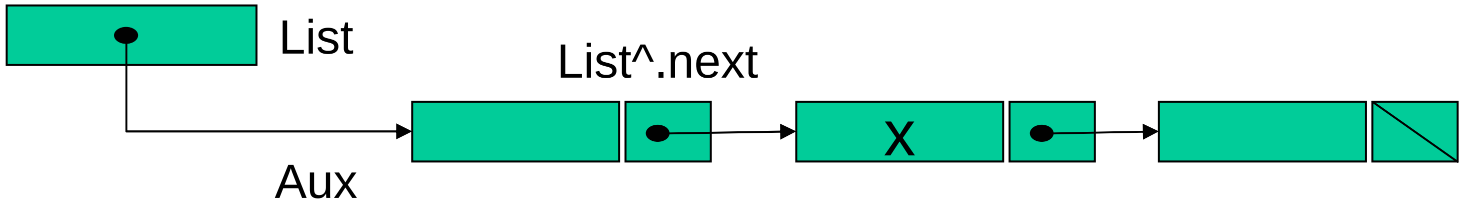
SORTED LISTS

- A sorted list is a list which nodes are ordered by a value .
- When the nodes in a sorted list are represented by records then their logical order is determined by one of their fields, called key record.
- Such value-ordered list are also called key-ordered lists.
- In our case the lists will be sorted in increasing order.
- To manage a sorted list is neccessary defined the next operations:
  - **orderedFind(x,List):** Returns a pointer pointing to the previous node with respect to that one which contains “x” and Nil if this last one is the first node of **List**.
  - **orderedinsert(x,List):** if **List** is empty then Inserts the node containing “x” at the first position and if not then inserts this one taking into account the established order.

FUNCTION ORDEREDFIND IMPLEMENTED IN PASCAL

```
FUNCTION OrderedFind (x:tElem; List:PNode):PNode;
    {Returns a pointer pointing to the previous node with respect to that one which contains
    “x” and Nil if this last one is the first node of the list.}
VAR
    Aux:PNode;
BEGIN
    Aux:=nil;
    IF  Not Emptylist(List) THEN BEGIN
          WHILE (x≥List^.info) AND (List^.next<>nil) DO BEGIN
                Aux:=List;
                 List:=List^.next
           END; {While}
           IF x≥List^.info THEN
                 Aux:=List
    END; {If}
    OrderedFind:=Aux
END; {OrderedFind}
```

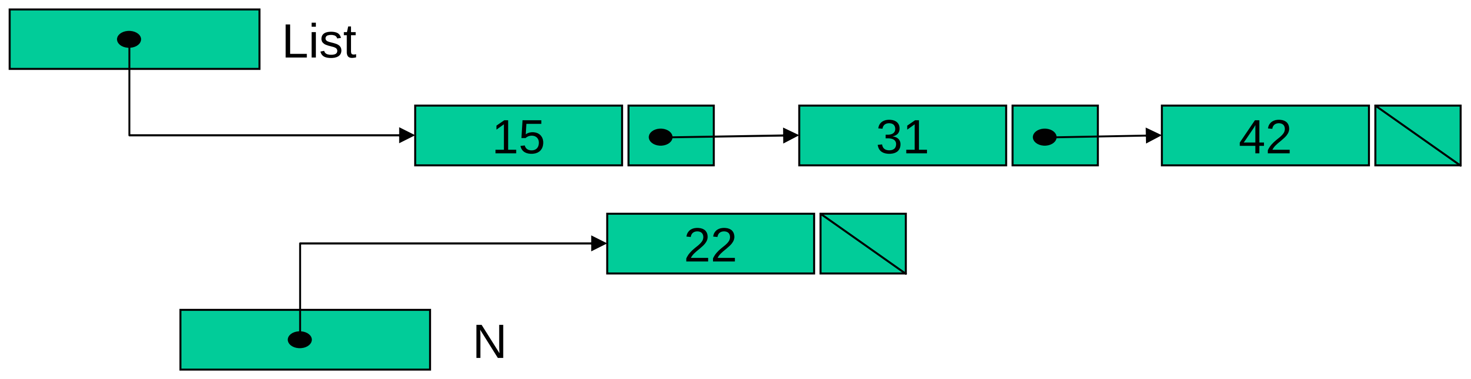
FUNCTION ORDEREDFIND IMPLEMENTED IN PASCAL

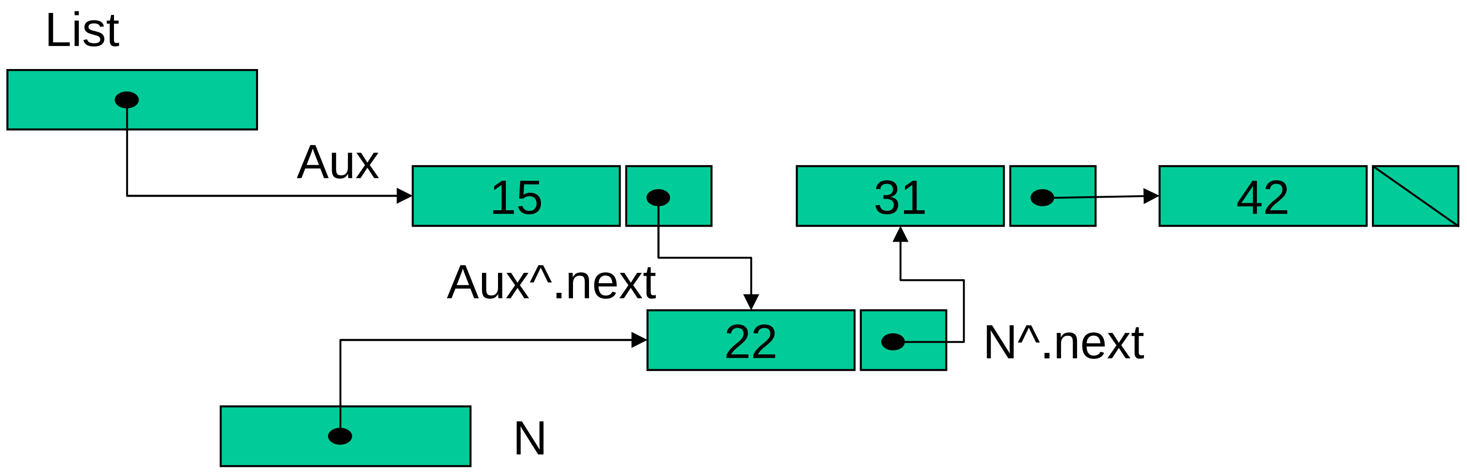


PROCEDURE ORDEREDINSERT IMPLEMENTED IN PASCAL

```
PROCEDURE Orderedinsert(x:tElem;  var L:PNode);
      {if List is empty then Inserts the node containing "x" at the first position and if not then
      inserts this one taking into account the established order}
VAR
      Aux,N:PNode;
BEGIN
      N:=CreateNode(x);
      IF  Emptylist(L) THEN
                List:=N
           ELSE BEGIN
                     Aux:=Posinser(x,List);
              IF Aux=nil THEN BEGIN {First position of List}
                       N^.next:=List;
                       List:=N
              END; {If}
                     ELSE BEGIN {any other position of List}
                                N^.next:=Aux^.next;
                                Aux^.next:=N
                     END {Else}
            END {Else}
END; {OrderedInsert}
```

PROCEDURE ORDEREDINSERT





A SEARCH ON A SORTED LIST

- This operation is more efficient since it is only neccessary to find the element or at least one greater.
  - **OrderedFind (x,L):** Returns a pointer pointing to the node of **List** containing “x” if this one exists and NIL if not

FUNCTION ORDEREDFIND IMPLEMENTED IN PASCAL

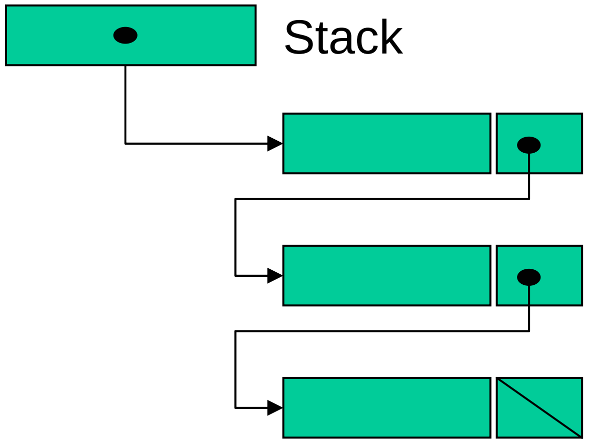
```
FUNCTION OrderedFind (x:tElem;   List:PNode):PNode;
   {Returns a pointer pointing to the node of List containing “x” if this one
    exists and NIL if not}
BEGIN
     WHILE (List^.next<>nil) AND (List^.info<x) DO
       List:=List^.next;
     IF List^.info=x THEN
            OrderedFind :=List
     ELSE
            OrderedFind :=nil
END; {OrderedFind}
```

THE STACK DATA STRUCTURE

- Stack is a common data structure that allows adding and removing elements of a certain type in a particular order. Every time an element is added, it goes on the top of the stack; the only element that can be removed is the element that was at the top of the stack.

- Consequently, a stack is said to have " last in, first out " behavior (LIFO). The first item added to an stack will be the last item removed from an stack.

- Remember: Stack is the underlying data structure for implementing recursion.



AN STACK IMPLEMENTATION USING POINTERS AND RECORDS (linked list scheme) IN PASCAL

```
TYPE
tElem=<type>
tStack=^tStackNode
tStackNode=record
    info:tElem;
    next:tStack;
End; {tStackNode}

VAR
Stack:tStack;
```

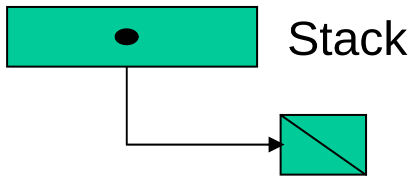
BASIC STACK OPERATION

- **CreateStack(Stack):** Initializes **Stack** to an empty state.
- **Emptystack(Stack):** Boolean function that **t**ests whether **stack** is empty.
- **Top(Stack):** Returns the content of the top of **Stack**
- **CreateNode(x):** Creates a node containing “x” for being stacked and returns a pointer pointing to this node.
- **Push(x,Stack):** Adds a new node containing “x” to the top of **Stack**
- **Pop(Stack):** Removes top element of **Stack.**

(“x” is a piece of information of certain type)

PROCEDURE CREATESTACK IMPLEMENTED IN PASCAL

```
PROCEDURE CreateStack(var Stack:tStack);
{Initializes Stack to an empty state.}
Begin
        Stack:=nil
End; {CreateStack}
```

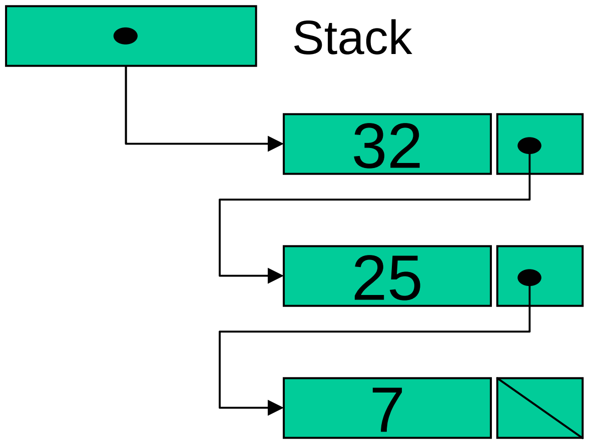


FUNCTION EMPTYSTACK IMPLEMENTED IN PASCAL

```
FUNCTION Emptystack(Stack:tStack):Boolean;
{Tests whether stack is empty.}
Begin
        Emptystack:=(Stack=nil)
End; {EmptyStack}
```

FUNCTION TOP IMPLEMENTED IN PASCAL

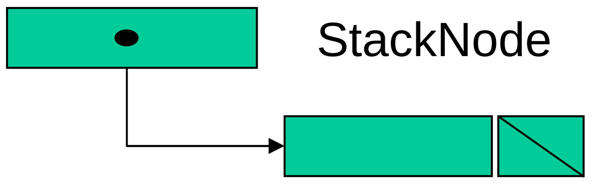
```
FUNCTION Top(Stack:tStack):tElem;
{Prec. Stack is not empty}
{Returns the content of the top of Stack}
Begin
        Top:=Stack^.info
End; {Top}
```



Top=32

FUNCTION CRETENODE IMPLEMENTED IN PASCAL

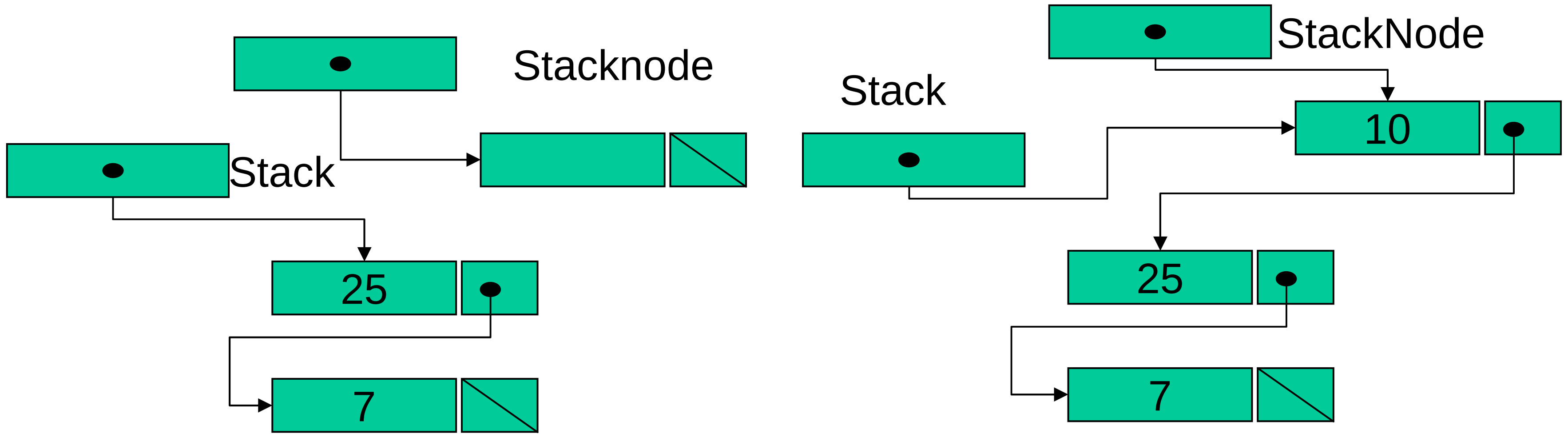
```
FUNCTION CreateNode(x:tElem):tStack;
   {Creates a node containing “x” for being stacked and returns a
    pointer pointing to this node}
VAR
    StackNode:tStack;
Begin
          New(StackNode);
          StackNode^.info:=x;
          StackNode^.next:=nil;
          CreateNode:=StackNode
End; {CreateNode}
```



PROCEDURE PUSH IMPLEMENTED IN PASCAL

```
PROCEDURE Push(x:telem; var Stack:tStack);
{Adds a new node containing “x” to the top of Stack}
VAR
   Aux:tStack;
Begin
        Aux:=CreateNode(x);
        Aux^.next:=Stack;
        Stack:=Aux;
End; {Push}
```

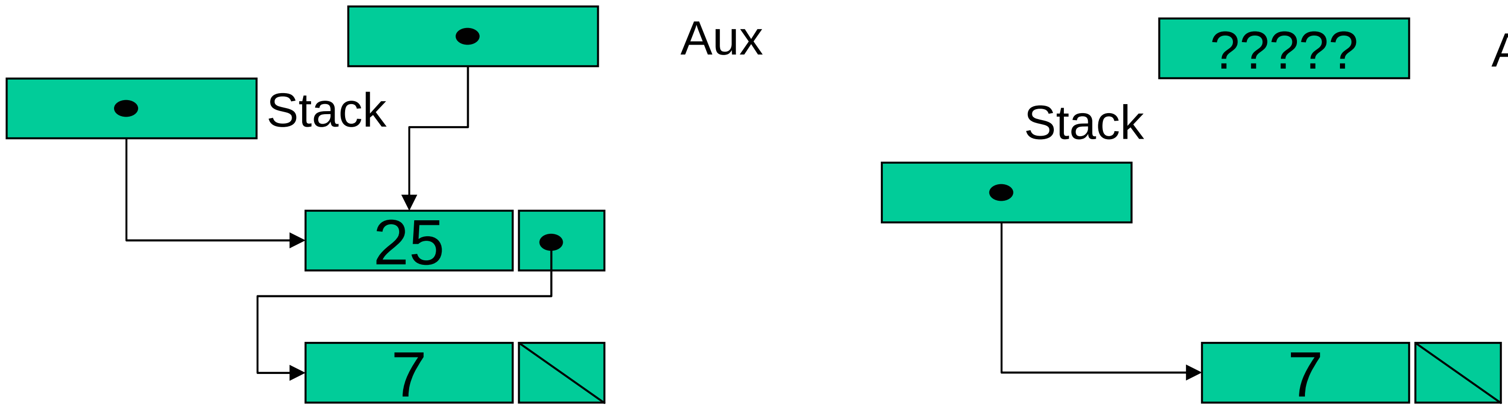
PROCEDURE PUSH



PROCEDURE POP IMPLEMENTED IN PASCAL

```
PROCEDURE Pop(var Stack:tStack);
{Prec. Stack is not empty}
{Removes top element of Stack}
VAR
   Aux:tStack;
Begin
        Aux:=Stack;
        Stack:=Aux^.next;
        Dispose(Aux)
End; {Pop}
```

PROCEDURE POP



STACK DATA STRUCTURE APPLICATION

Recursion:

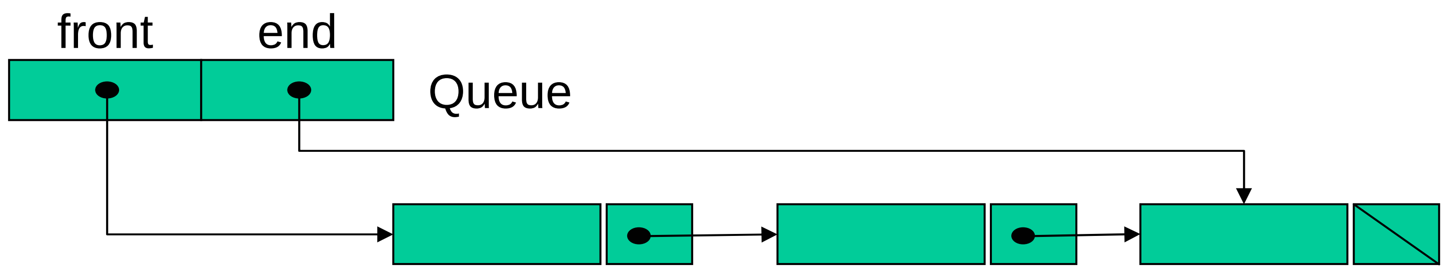
- An iterative way to represent recursion process involves an explicit implementation of a stack data structure.
- Under this approach recursion may be implemented by means of two consecutive loops. The first one stacks and the second one pops up the stored elements.

EXAMPLE: FACTORIAL OF AN INTEGER

```
FUNCTION Faciter(num:integer):integer;
{Prec. num≥0}
{Returns num!}
VAR
    Stack:tStack;
    n,fac:integer;
BEGIN
    createStack(Stack);
    {First loop: Stacks the elements}
    FOR  n:=num DOWNTO 1 DO
            Push(n,Stack);
    {Second loop:  Removes elements and works out factorial function }
    fac:=1 {base case}
    WHILE not EmptyStack(Stack) DO BEGIN
            fac:=Top(Stack)*fac;
            Pop(Stack)
    END; {while}
    Faciter:=fac
END; {Faciter}
```

THE QUEUE DATA STRUCTURE

- A queue data structure is an homogeneus group of elements in which new elements are added at its rear-end and elements are removed from its front-end (First In-First Out access).
- Applications: Printer network management.



A QUEUE IMPLEMENTATION USING POINTERS AND RECORDS (linked list scheme) IN PASCAL

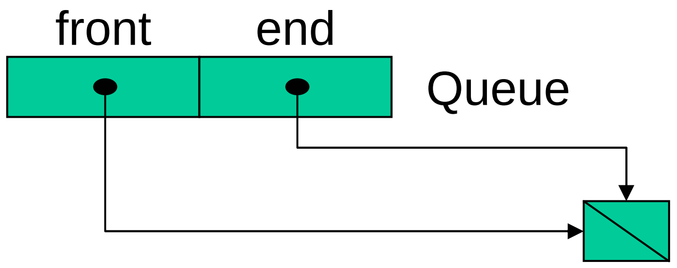
```
TYPE
tElem=<type>
tPNode=^tQueueNode
tQueueNode=record
    info:tElem;
    next:tPNode;
End; {tQueueNode}

tQueue=record
    ini:tPNode;
    fin:tPNode
end;{tQueue}

VAR
Queue:tQueue;
```

BASIC QUEUE OPERATION

- **Createqueue(Queue):** Initializes **Queues** to empty state.
- **EmptyQueue(Queue):** Determines whether queue is empty.
- **Front(Queue):** Returns the content of the front-end node of **Queue**
- **CreateNode(x):** Creates a node containing “x” for being added and returns a pointer pointing to this node.
- **Enqueue(x, Queue):** Adds a new node containing “x” at rear-end of **Queue**.
- **Dequeue(Queue):** Removes nodes from front-end of **Queue**.

PROCEDURE CREATEQUEUE IMPLEMENTED IN PASCAL

```
PROCEDURE CreateQueue(var Queue:tQueue);
{Initializes Queues to empty state}
Begin
        Queue.ini:=nil;
        Queue.fin:=nil
End; {CreateQueue}
```



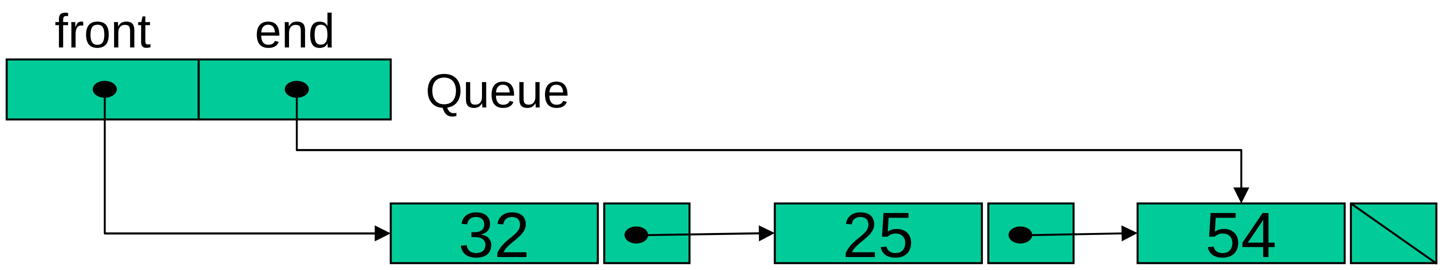
FUNCTION EMPTYQUEUE IMPLEMENTED IN PASCAL

```
FUNCTION EmptyQueue(Queue:tQueue):Boolean;
{Determines whether queue is empty}
Begin
    EmptyQueue:=(Queue.ini=nil)
End; {EmptyQueue}
```

FUNCTION FRONT IMPLEMENTED IN PASCAL

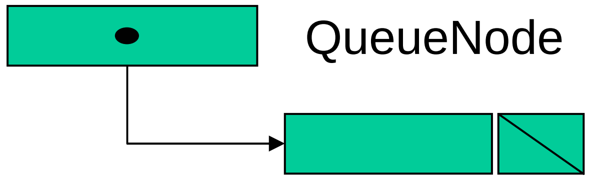
```
FUNCTION Front(Queue:tQueue):tElem;
{Prec. Queue is not empty}
{Returns the content of the front-end node of Queue}
Begin
        Front:=Queue.ini^.info
End; {Front}
```

Front=32



FUNCTION CREATENODE IMPLEMENTED IN PASCAL

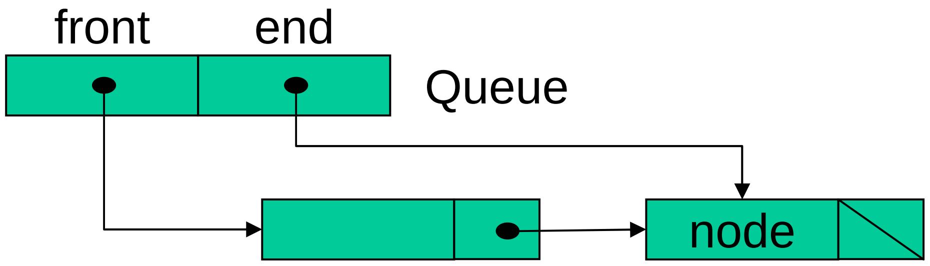
```
FUNCTION CreateNode(x:tElem):tPNode;
    {Creates a node containing “x” for being added and
    returns a pointer pointing to this node}
VAR
    QueueNode:tPNode;
Begin
        New(QueueNode);
        QueueNode^.info:=x;
        QueueNode^.next:=nil;
        CreateNode:=QueueNode
End; {CreateNode}
```

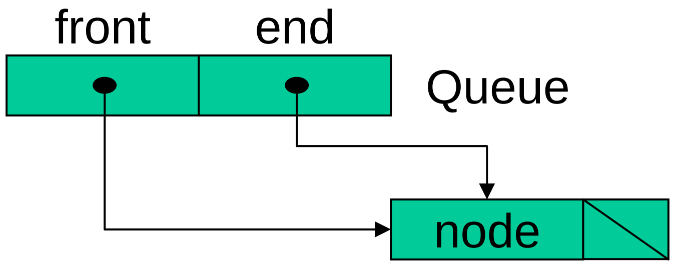


PROCEDURE ENQUEUE IMPLEMENTED IN PASCAL

```
PROCEDURE Enqueue(x:telem; var Queue:tQueue);
{Adds a new node containing “x” at rear-end of Queue}
VAR
    node:tPNode;
Begin
           node:=CreateNode(x);
           if  not EmptyQueue then begin
                      Queue.fin^.next:=node;
                      Queue.fin:=node;
                end; {if}
           else begin {Empty Queue}
                      Queue.fin:=node;
                      Queue.ini:=node
                 end{else}
End; {Enqueue}
```

PROCEDURE ENQUEUE

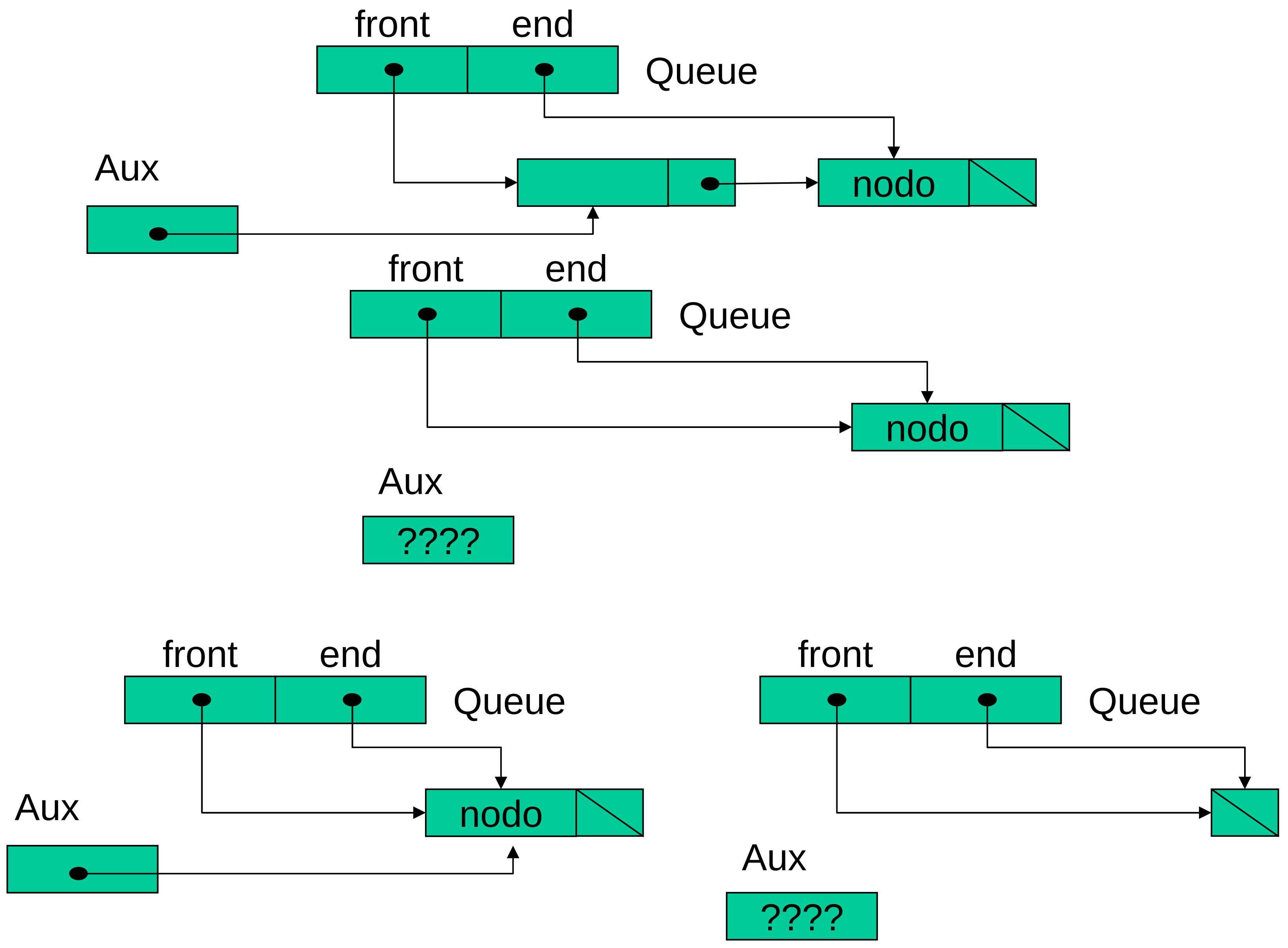




PROCEDURE DEQUEUE IMPLEMENTED INPASCAL

```
PROCEDURE DeQueue(var Queue:tQueue);
{Prec. Queue is not empty}
{Removes nodes from front-end of Queue}
VAR
    Aux:tPNode;
Begin
          Aux:=Queue.ini
          if  not (Queue.ini=Queue.fin) then begin
                    Queue.ini:=Aux^.next;
              end; {if}
          else begin {one element Queue}
                    Queue.fin:=nil;
                    Queue.ini:=nil;
              end{else}
          dispose(Aux)
End; {DeQueue}
```

PROCEDURE DEQUEUE


front
end
Queue
Aux
nodo
front
end
Queue
nodo
Aux
????
front
end
Queue
Aux
nodo
front
end
Queue
Aux
????