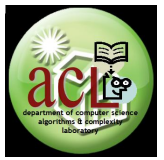
Linear Lists

Lesson 8



Outline

- Linear Lists
  - Basic concepts
  - Basic operations
- Common implementations: linked list
  - Straight singly-linked list (with list head)
  - Circular singly-linked list (with list head)
  - Doubly-linked list
- Applications
  - Polynomial arithmetic
  - Dynamic storage management

Outline

- **Linear Lists**
  - **Basic concepts**
  - **Basic operations**
- Common implementations: linked list
  - Straight singly-linked list (with list head)
  - Circular singly-linked list (with list head)
  - Doubly-linked list
- Applications
  - Polynomial arithmetic
  - Dynamic storage management

Linear List

- A ***list is a finite, ordered set of zero or more elements***
  - ***Elements may be atoms or lists***
- ***A linear list is a list where all elements are atoms***
  - ***A generalized list (or list structure) is a list where an element may either be a list or an element***

$$L = (x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}, x_n)$$

Basic operations

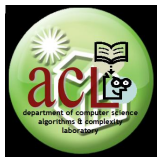
- Initialize a list.
- Determine whether a list is empty.
- Find the length, say n, of a list, i.e., the number of elements in the list.
- Gain access to the ith element of a list, $1 \leq i \leq n$.
- Replace the ith element of a list.
- Delete the ith element of a list.
- Insert a new element into a list.
- Combine two or more lists into a single list.

Basic operations

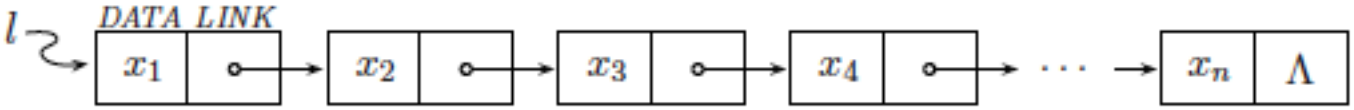
- Split a list into two or more lists.
- Make a copy of a list.
- Erase a list.
- Search a list.
- Sort a list.

Outline

- Linear Lists
  - Basic concepts
  - Basic operations
- **Common implementations: linked list**
  - **Straight singly-linked list (with list head)**
  - **Circular singly-linked list (with list head)**
  - **Doubly-linked list**
- Applications
  - Polynomial arithmetic
  - Dynamic storage management



Straight singly-linked list (SSLL)



procedure LIST_INSERT($\mathbb{L}, w, x$)

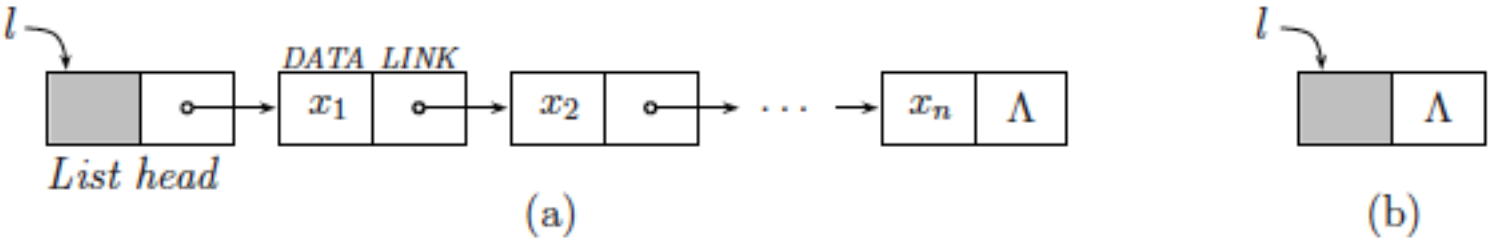
Inserts a new element w before element x in the list; if there is no element x in the list, w is inserted at the tail end of the list.

call GETNODE(ν)
$DATA(\nu) \leftarrow w$
$\alpha \leftarrow l$
loop
  if $\alpha = \Lambda$ **or** $DATA(\alpha) = x$ **then** [**if** $\alpha = l$ **then** $l \leftarrow \nu$
    else $LINK(\beta) \leftarrow \nu$
    $LINK(\nu) \leftarrow \alpha$; **return**]
  else [$\beta \leftarrow \alpha$; $\alpha \leftarrow LINK(\alpha)$]
forever
end LIST_INSERT

Straight singly-linked list (SSLL)


```
procedure LIST_DELETE(𝕃, x)
Deletes element x from the list
    α ← l
    while α ≠ Λ do
      if DATA(α) = x then [if α = l then l ← LINK(l)
                                    else LINK(β) ← LINK(α)
                           call RETNODE(α); return]
                      else [β ← α;  α ← LINK(α)]
    endwhile
    end LIST_DELETE
```

SSLL with list head

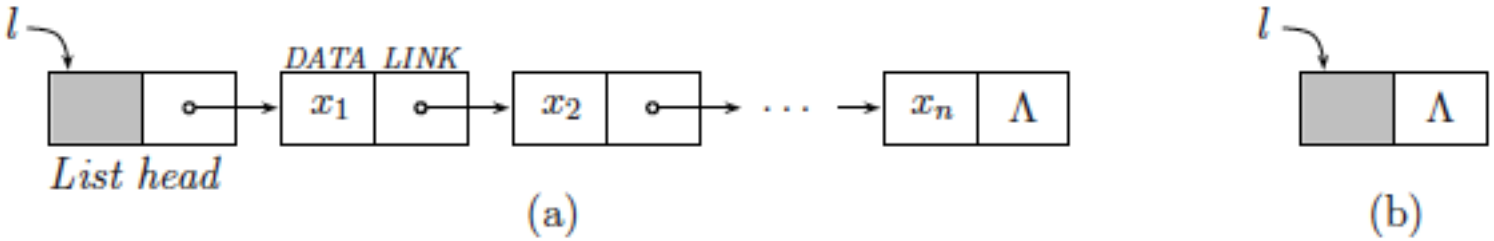


procedure LIST_INSERT($\mathbb{L}, w, x$)

Inserts a new element w before element x in the list; if there is no element x in the list, w is inserted at the tail end of the list.

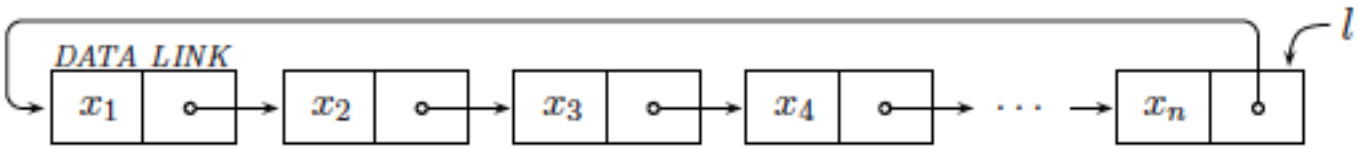
call GETNODE(ν)
$DATA(\nu) \leftarrow w$
$\alpha \leftarrow LINK(l)$; $\beta \leftarrow l$
loop
  if $\alpha = \Lambda$ **or** $DATA(\alpha) = x$ **then** [$LINK(\beta) \leftarrow \nu$; $LINK(\nu) \leftarrow \alpha$; **return**]
  else [$\beta \leftarrow \alpha$; $\alpha \leftarrow LINK(\alpha)$]
forever
end LIST_INSERT

SSLL With List Head




```
procedure LIST_DELETE(𝕃, x)
Deletes element x from the list
    α ← LINK(l); β ← l
    while α ≠ Λ do
      if DATA(α) = x then [LINK(β) ← LINK(α); call RETNODE(α); return]
                     else [β ← α; α ← LINK(α)]
    endwhile
    end LIST_DELETE
```

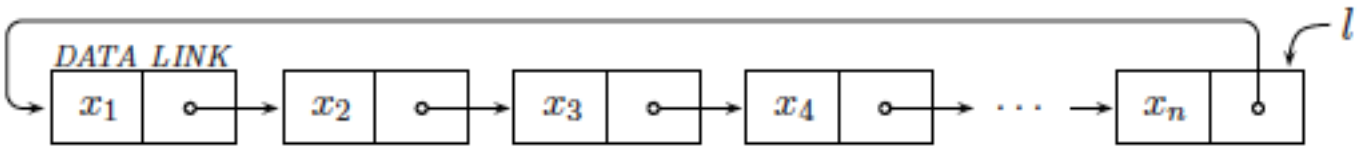
Circular singly-linked list (CSLL)



procedure INSERT_LEFT($\mathbb{L}, x$)
Inserts element x at left end of a circular list
call GETNODE(α)
$DATA(\alpha) \leftarrow x$
if $l = \Lambda$ **then** $[LINK(\alpha) \leftarrow \alpha; l \leftarrow \alpha]$
else $[LINK(\alpha) \leftarrow LINK(l); LINK(l) \leftarrow \alpha]$
end INSERT_LEFT

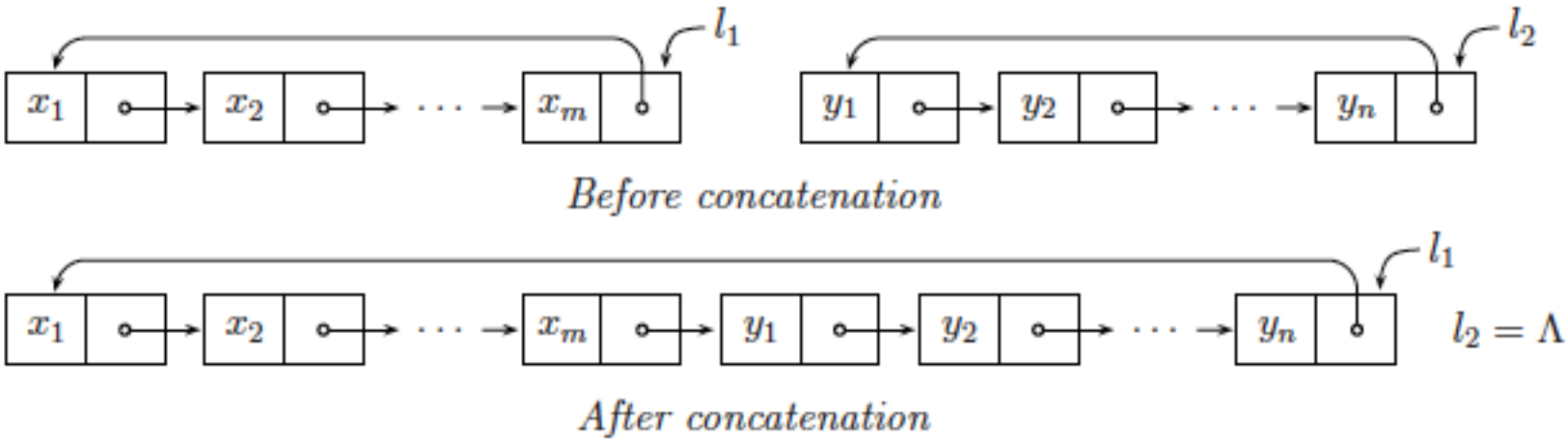
procedure INSERT_RIGHT($\mathbb{L}, x$)
Inserts element x at right end of a circular list
call INSERT_LEFT($\mathbb{L}, x$)
$l \leftarrow LINK(l)$
end INSERT_RIGHT

Circular singly-linked list (CSLL)



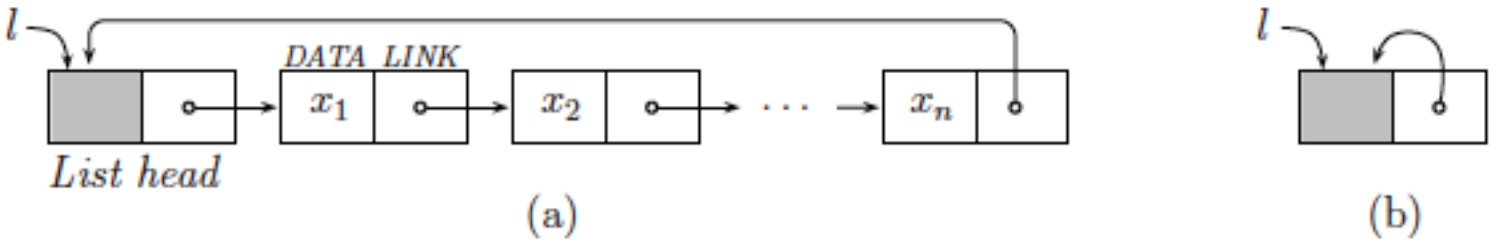

```
procedure DELETE_LEFT(𝕃, x)
Deletes leftmost element of a circular list and returns this in x
  if l = Λ then call CLIST_EMPTY
           else [α ← LINK(l); x ← DATA(α); LINK(l) ← LINK(α)
                 if α = l then l ← Λ
                 call RETNODE(α)]
  end DELETE_LEFT
```

Circular singly-linked list (CSLL)



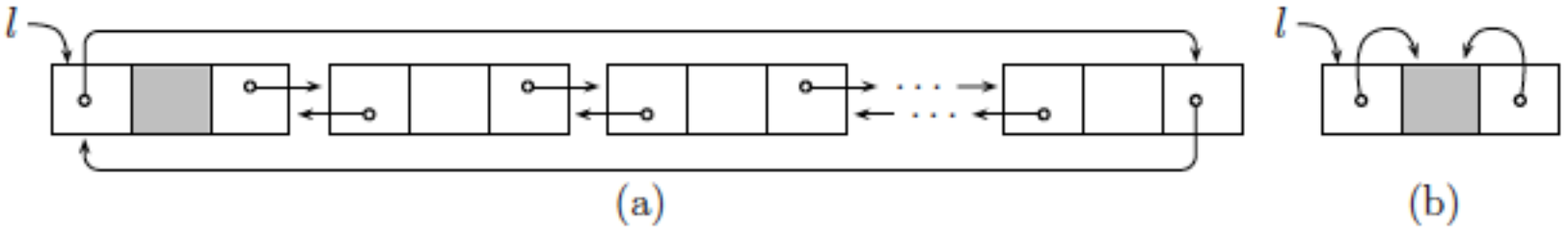
procedure CONCATENATE($\mathbb{L}1, \mathbb{L}2$)
Concatenates two circular lists
if $\mathbb{L}2.l \neq \Lambda$ **then** [**if** $\mathbb{L}1.l \neq \Lambda$ **then** [$\alpha \leftarrow \mathbb{L}1.LINK(\mathbb{L}1.l)$
$\mathbb{L}1.LINK(\mathbb{L}1.l) \leftarrow \mathbb{L}2.LINK(\mathbb{L}2.l)$
$\mathbb{L}2.LINK(\mathbb{L}2.l) \leftarrow \alpha$]
$\mathbb{L}1.l \leftarrow \mathbb{L}2.l$
$\mathbb{L}2.l \leftarrow \Lambda$]
end CONCATENATE

CSLL with list head

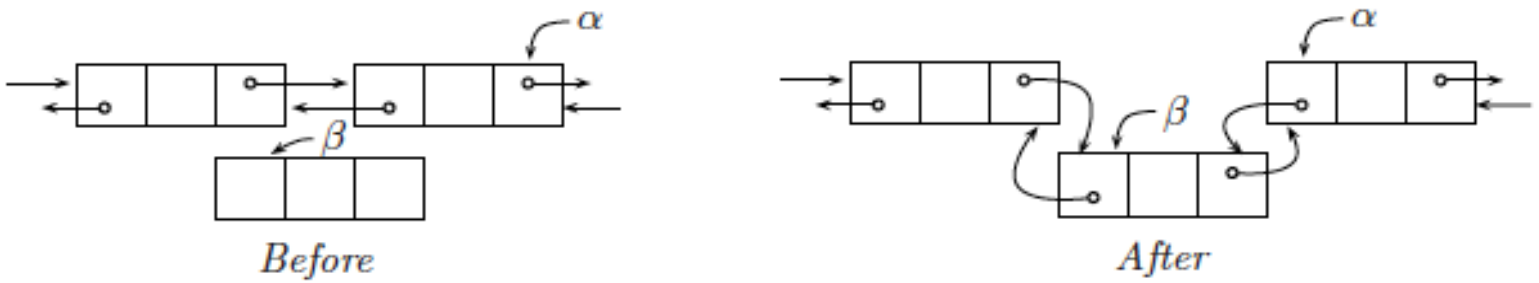


- To be utilized in polynomial arithmetic later

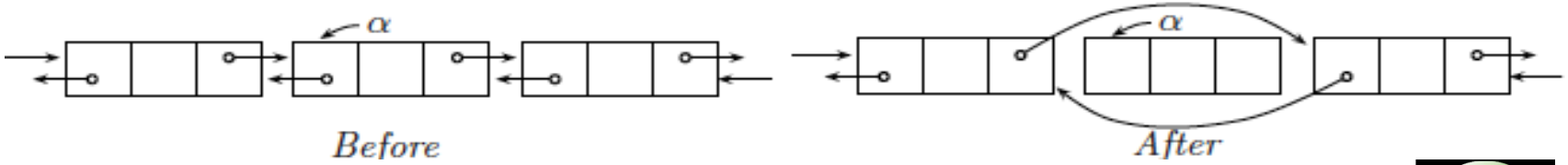
Doubly-linked list



$$RLINK(LLINK(\alpha)) \leftarrow \beta;\ RLINK(\beta) \leftarrow \alpha;\ \ LLINK(\beta) \leftarrow LLINK(\alpha);\ LLINK(\alpha) \leftarrow \beta$$



$$RLINK(LLINK(\alpha)) \leftarrow RLINK(\alpha);\ \ LLINK(RLINK(\alpha)) \leftarrow LLINK(\alpha)$$

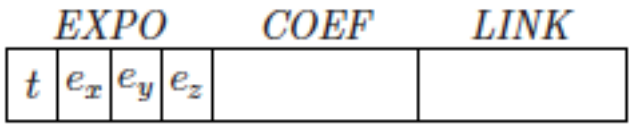


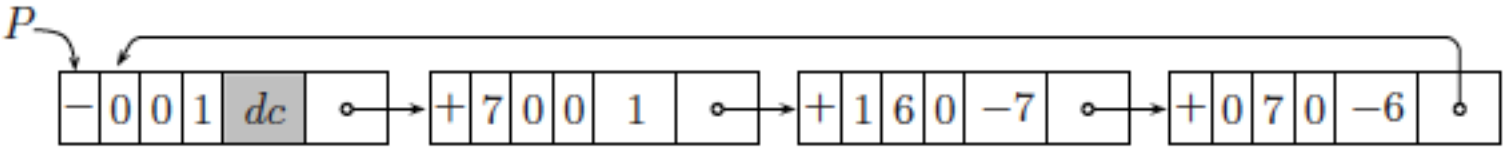
Outline

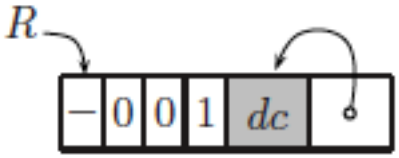
- Linear Lists
  - Basic concepts
  - Basic operations
- Common implementations: linked list
  - Straight singly-linked list (with list head)
  - Circular singly-linked list (with list head)
  - Doubly-linked list
- **Applications**
  - **Polynomial arithmetic**
  - **Dynamic storage management**

Polynomial arithmetic

$$P(x,y) = x^5 - 2x^4y + 3x^3y^2 - 4x^2y^3 + 5xy^4 - 6y^5 \qquad Q(x,y) = x^2 + 2xy + y^2$$







Polynomial arithmetic

- Internal representation of polynomials

procedure POLYREAD($\mathbb{P}$, *name*)
call ZEROPOLY($\mathbb{P}$, *name*)
while not EOI **do**
  input *ex*, *ey*, *ez*, *coef*
  call GETNODE(ν)
  $\mathbb{P}.EXPO(\nu) \leftarrow (+, ex, ey, ez)$
  $\mathbb{P}.COEF(\nu) \leftarrow coef$
  $\beta \leftarrow \mathbb{P}.l$; $\alpha \leftarrow \mathbb{P}.LINK(\beta)$
  while $\mathbb{P}.EXPO(\alpha) > \mathbb{P}.EXPO(\nu)$ **do**
    $\beta \leftarrow \alpha$; $\alpha \leftarrow \mathbb{P}.LINK(\alpha)$
  endwhile
  $\mathbb{P}.LINK(\beta) \leftarrow \nu$
  $\mathbb{P}.LINK(\nu) \leftarrow \alpha$
endwhile
end POLYREAD

procedure ZEROPOLY($\mathbb{P}$, *name*)
call GETNODE(τ)
$\mathbb{P}.l \leftarrow \tau$
$\mathbb{P}.name \leftarrow name$
$\mathbb{P}.EXPO(\tau) \leftarrow (-, 0, 0, 1)$
$\mathbb{P}.LINK(\tau) \leftarrow \tau$
end ZEROPOLY

Polynomial arithmetic

- Polynomial addition

```
procedure POLYADD(ℙ, ℚ, name)
α ← ℙ.LINK(ℙ.l)          ▷ pointer to current term (node) in P
β ← ℚ.LINK(ℚ.l)          ▷ pointer to current term (node) in Q
σ ← ℚ.l                  ▷ a pointer that is always one node behind β
loop
  case
    :ℙ.EXPO(α) < ℚ.EXPO(β) : [σ ← β; β ← ℚ.LINK(β);]
    :ℙ.EXPO(α) = ℚ.EXPO(β) : [if ℙ.EXPO(α) < 0 then [ℚ.name ← name; return]
                              ℚ.COEF(β) ← ℚ.COEF(β) + ℙ.COEF(α)
                              if ℚ.COEF(β) = 0 then [τ ← β
                                                     ℚ.LINK(σ) ← ℚ.LINK(β)
                                                     β ← ℚ.LINK(β)
                                                     call RETNODE(τ)]
                                               else [σ ← β
                                                     β ← ℚ.LINK(β)]
                              α ← ℙ.LINK(α)]
    :ℙ.EXPO(α) > ℚ.EXPO(β) : [call GETNODE(τ)
                              COEF(τ) ← ℙ.COEF(α); EXPO(τ) ← ℙ.EXPO(α)
                              ℚ.LINK(σ) ← τ; LINK(τ) ← β
                              σ ← τ; α ← ℙ.LINK(α)]
  endcase
forever
end POLYADD
```

Polynomial arithmetic

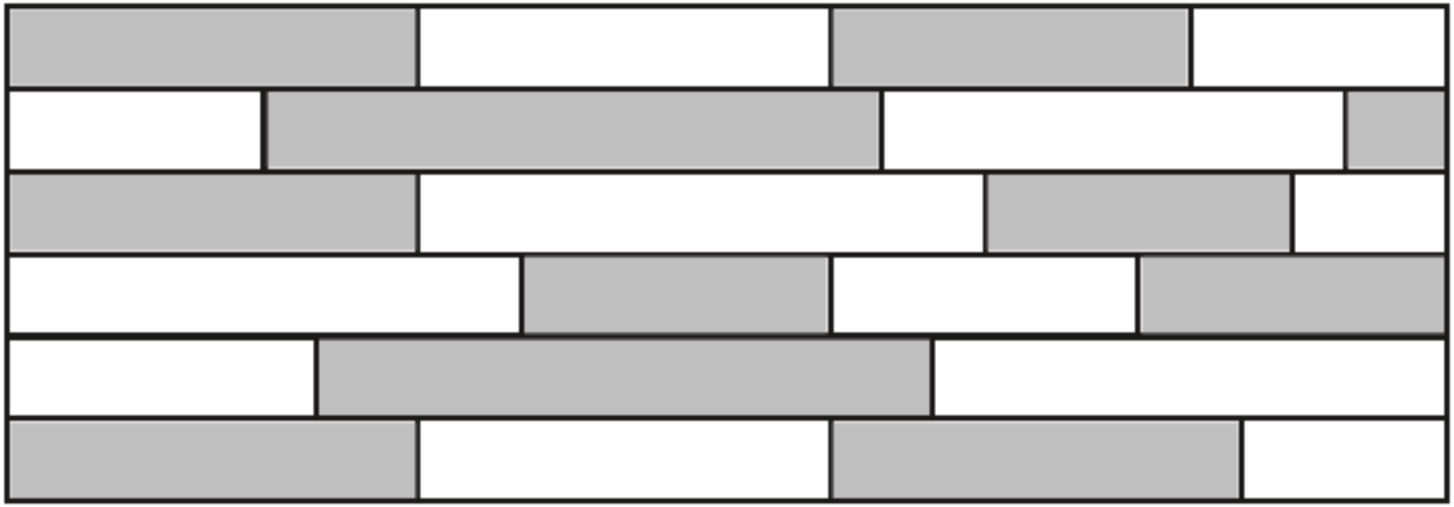
- Polynomial multiplication

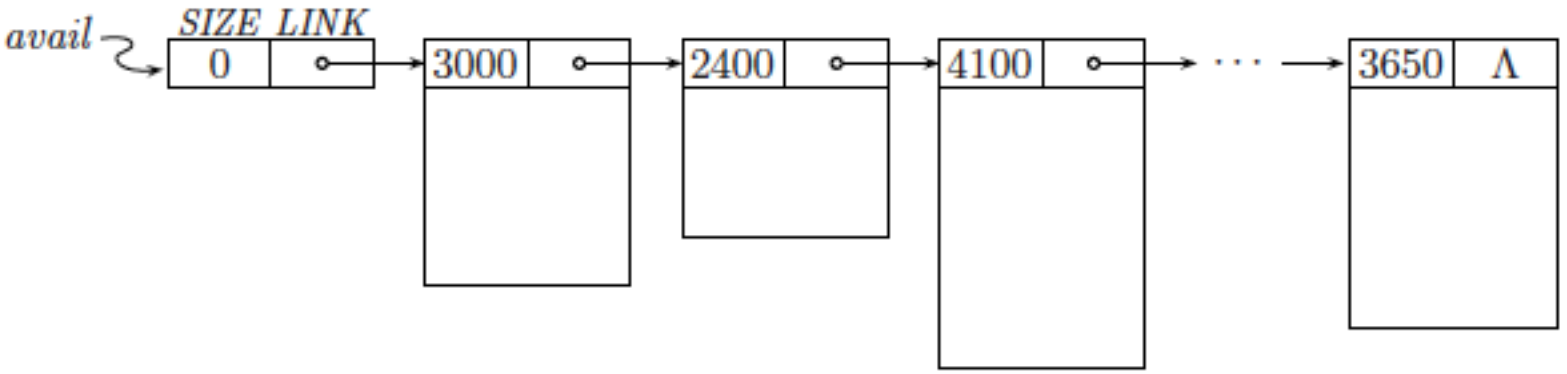
```
    procedure POLYMULT(ℙ, ℚ, ℝ, name)
Create temporary polynomial 𝕋 to contain product term.
    call ZEROPOLY(𝕋,' ')
    call GETNODE(τ)
    𝕋.LINK(𝕋.l) ← τ
    LINK(τ) ← 𝕋.l
Perform multiplication
    α ← ℙ.LINK(ℙ.l)
    while ℙ.EXPO(α) ≠ −001 do
      β ← ℚ.LINK(ℚ.l)
      while ℚ.EXPO(β) ≠ −001 do
        𝕋.COEF(τ) ← ℙ.COEF(α) * ℚ.COEF(β)
        𝕋.EXPO(τ) ← ℙ.EXPO(α) + ℚ.EXPO(β)
        call POLYADD(𝕋, ℝ, name)
        β ← ℚ.LINK(β)
      endwhile
      α ← ℙ.LINK(α)
    endwhile
    call RETPOLY(𝕋)
    end POLYMULT
```

Dynamic storage management

- ***Dynamic storage management*** (***DSM***) is the management of a contiguous area of memory, called the ***memory pool*** through various techniques for allocating and deallocating blocks in the pool.
  - The memory pool consists of individually addressable units called ***words*** and that block sizes are measured in number of words
- DSM operations
  - ***Reservation (or allocation)***
  - ***Liberation (or deallocation)***

DSM: Basics

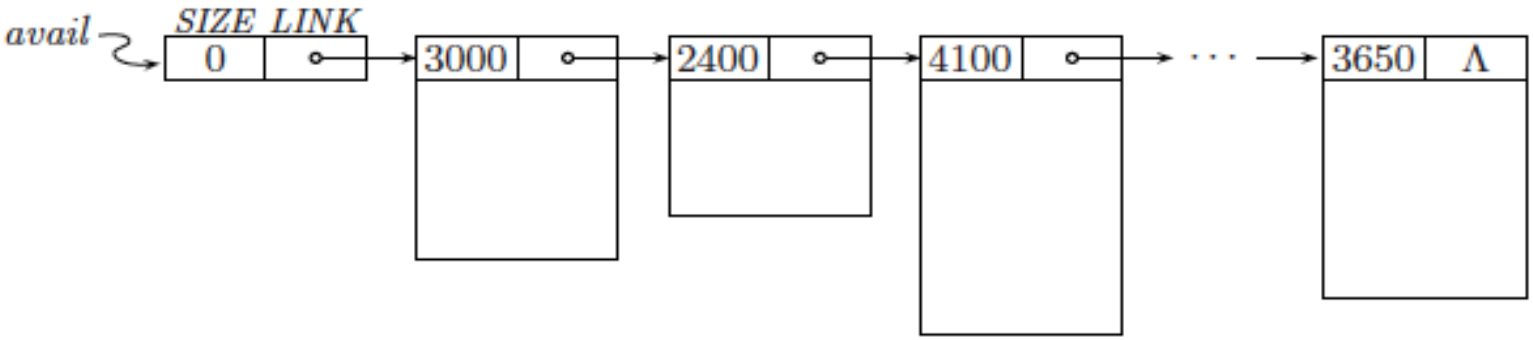




DSM: Reservation

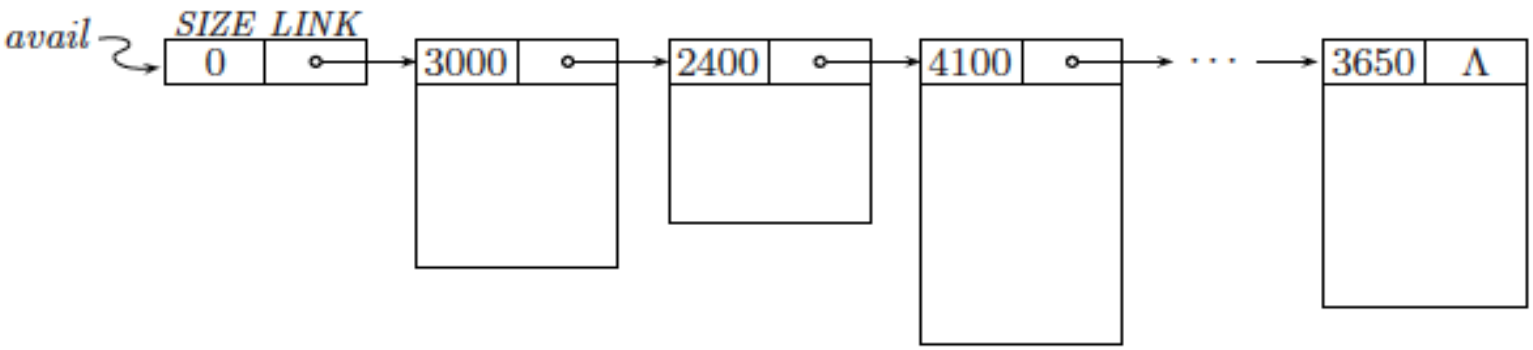
- To satisfy a request for n words, we scan the avail list and choose a block based on a 'fitting' criterion
  - ***First-fit method***
  - ***Best-fit method***
  - ***Optimal-fit method***
  - ***Worst-fit method***

First-fit method (version 1)



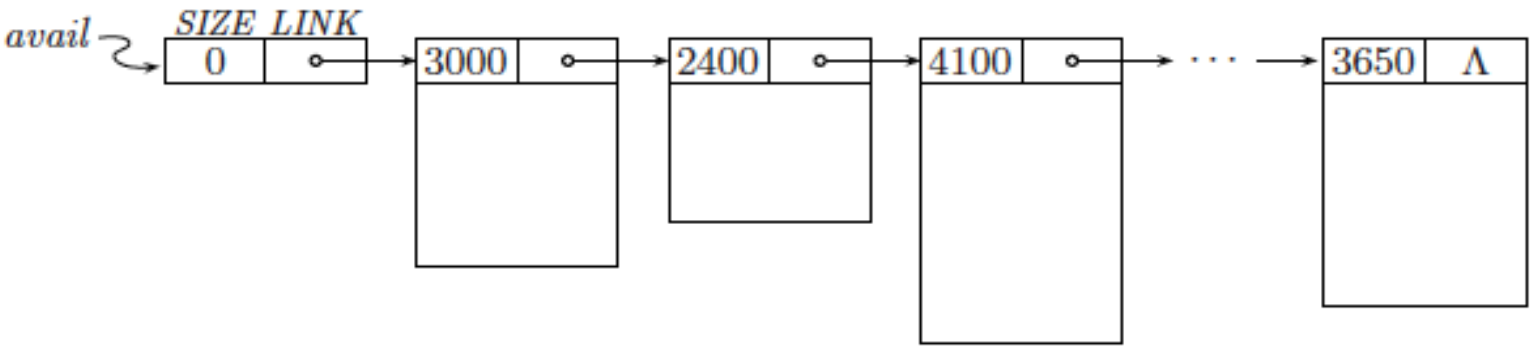

```
procedure FIRST_FIT_1(n)
alpha ← LINK(avail)        ▷ pointer to current block
β ← avail                  ▷ trailing pointer
while α ≠ Λ
   if SIZE(α) ≥ n then [SIZE(α) ← SIZE(α) − n
                        if SIZE(α) = 0 then LINK(β) ← LINK(α)
                                       else  α ← α + SIZE(α)
                        return(α)]
                  else  [β ← α;  α ← LINK(α)]
endwhile
return(Λ)
end FIRST_FIT_1
```

First-fit method (version 2)



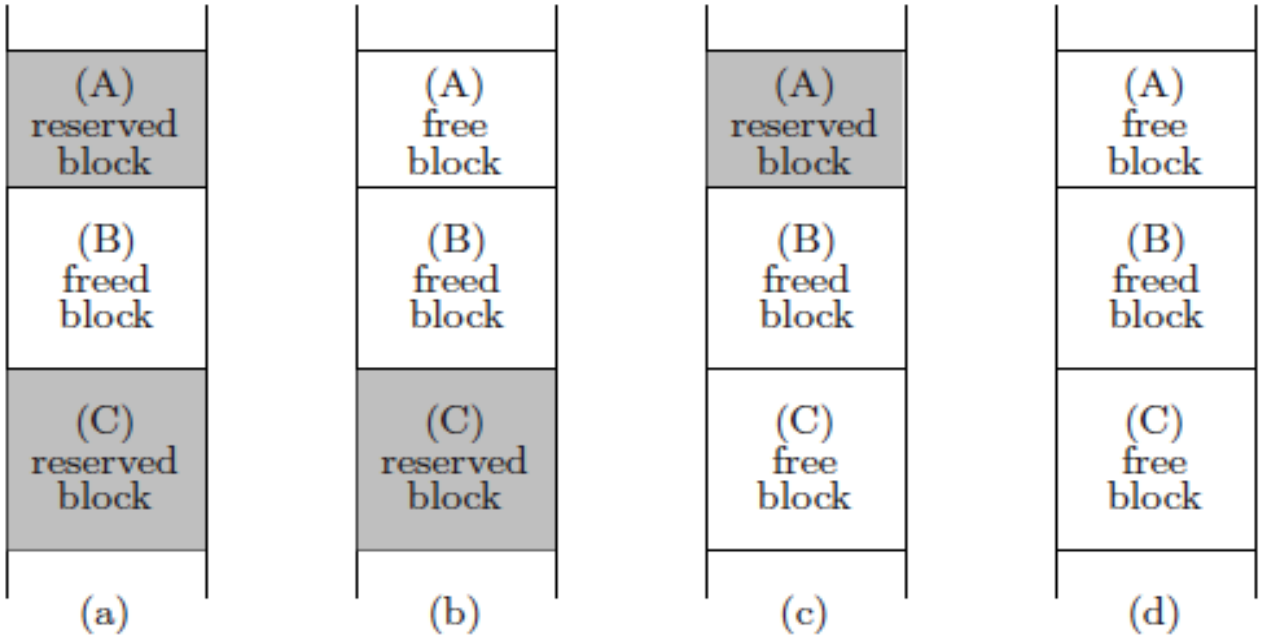

```
procedure FIRST_FIT_2(n, ρ, minsize)
alpha ← LINK(ρ)
β ← ρ
flag ← 0
while β ≠ ρ or flag = 0 do
   if α = Λ then [α) ← LINK(avail); β ← avail; flag ← 1]
   if SIZE(α) ≥ n then [excess ← SIZE(α) − n
                        if excess < minsize then [LINK(β) ← LINK(α)
                                                  ρ ← β]
                                            else [SIZE(α) ← excess
                                                  ρ ← α
                                                  α ← α + excess
                                                  SIZE(α) ← n]
                        return(α)]
                  else [β ← α; α ← LINK(α)]
endwhile
return(Λ)
end FIRST_FIT_2
```

Best-fit method




```
procedure BEST_FIT(n, minsize)
alpha ← LINK(avail)
β ← avail
bestsize ← ∞
while α ≠ Λ do
   if SIZE(α) ≥ n then if bestsize > SIZE(α) then [bestsize ← SIZE(α); τ ← β]
   β ← α
   α ← LINK(α)
endwhile
if bestsize = ∞ then return(Λ)
                else [α ← LINK(τ)      ▷ best-fitting block
                      excess ← bestsize − n
                      if excess < minsize then LINK(τ) ← LINK(α)
                                          else [SIZE(α) ← excess
                                                α ← α + excess
                                                SIZE(α) ← n]
                      return(α)]
end BEST_FIT
```

DSM: Liberation

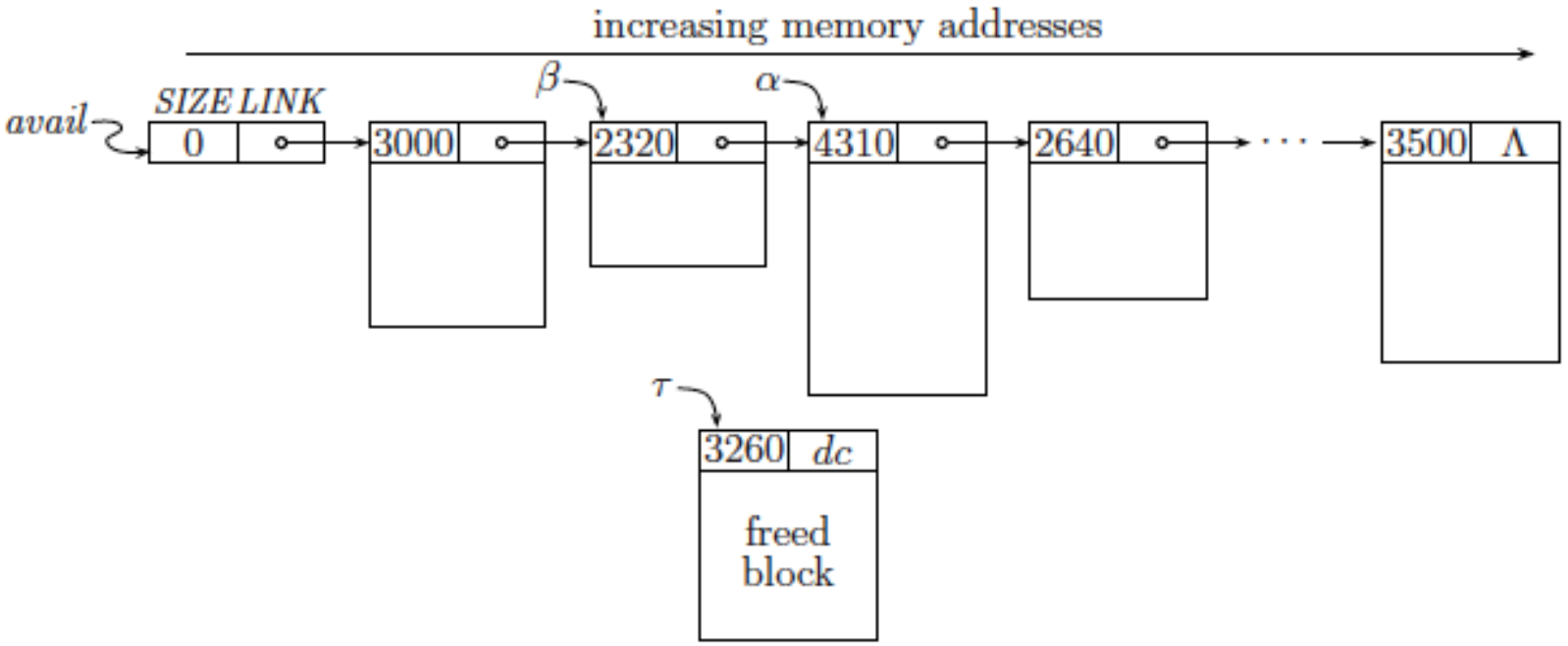


- Two techniques
  - ***Sorted-list technique***
  - ***Boundary-tags***

Sorted-list technique

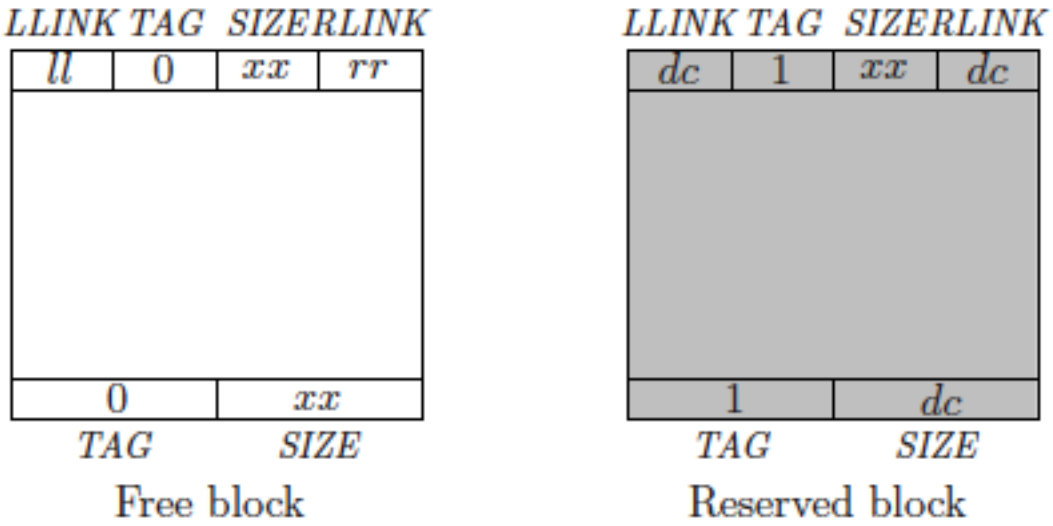
- Avail list is assumed to be in order of increasing memory address.
- Avail list is traversed, and position of the freed block (w.r.t. the blocks in avail) is determined
- Collapsing is possible if the freed block is adjacent to the block/s adjacent to its position
  - start address of left block + size of left block = start address of freed block
  - start address of freed block + size of freed block = start address of right block
  - both

Sorted-list technique



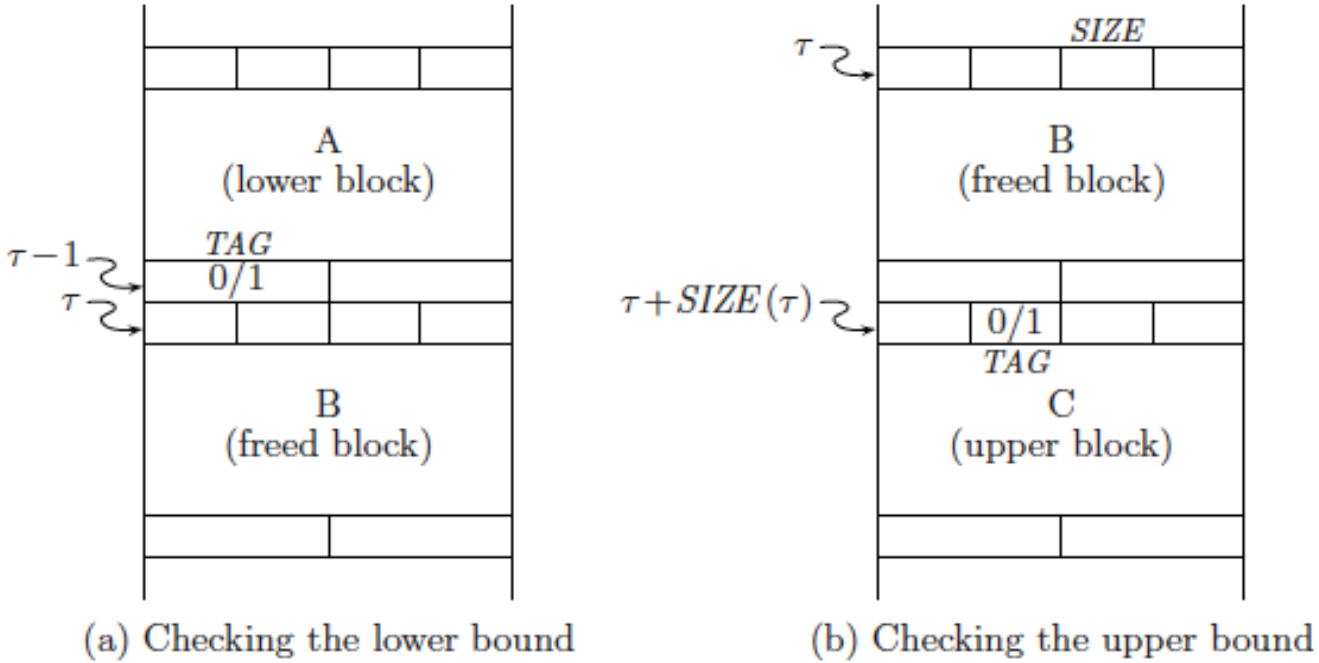
Boundary-tag technique

- Each block is bounded by two control words
- **Avail list is a _____ list**



Boundary-tag technique

- If tag of lower (upper) block is 0, that block is free and thus we collapse the freed block and the lower (upper) block



(a) Checking the lower bound

(b) Checking the upper bound

Thank you
for your attention.

Questions ...