Generalized Lists

Lesson 9

Outline

- Basic concepts
- Implementations
  - Linked Implementation
    - Basic
    - List heads
    - Alternative (LISP structure)
  - Operations
- Application: Automatic Storage Reclamation

Outline

- **Basic concepts**
- Implementations
  - Linked Implementation
    - Basic
    - List heads
    - Alternative (LISP structure)
  - Operations
- Application: Automatic Storage Reclamation

Basic Concepts

- ***atom(x)***
- ***null(L)***
- ***head(L)***
- ***tail(L)***
- ***length and depth***
- ***graph/tree representation***
  - ***pure***
  - ***reentrant***
  - ***recursive / cyclic***

$$L = ((a,b),((c,d),e))$$

$$L = ((a,b),c,(),(d,(e)))$$

$$L = (((a,b),P),(P),(P,Q),(Q,h))$$

$$P = (c,d,e)$$

$$Q = (f,(g))$$

$$L = ((a,b,(c,L),S,(d,(L)))$$

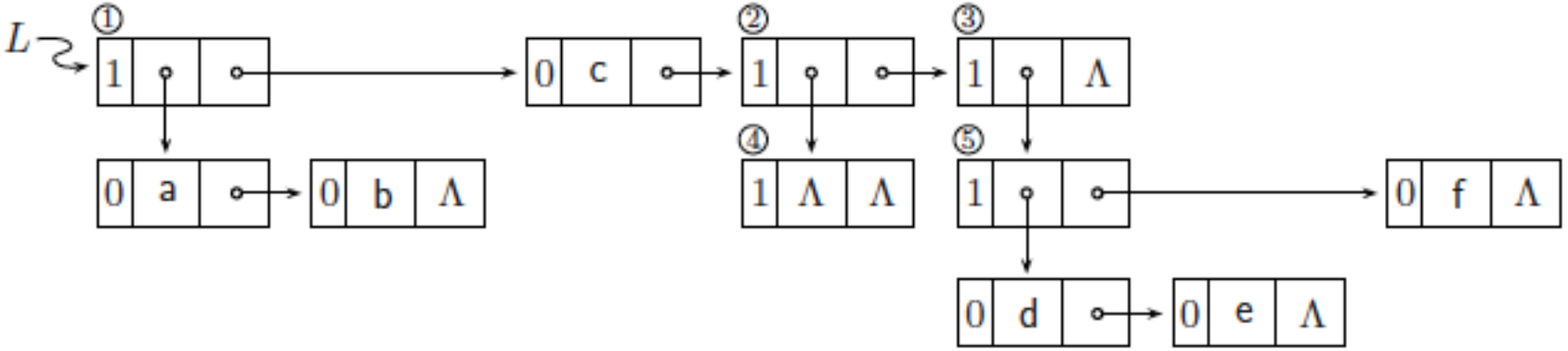
$$S = ((L,p,(q,S)),(r))$$

Outline

- Basic concepts
- **Implementations**
  - **Linked Implementation**
    - **Basic**
    - **List heads**
    - **Alternative (LISP structure)**
  - **Operations**
- Application: Automatic Storage Reclamation

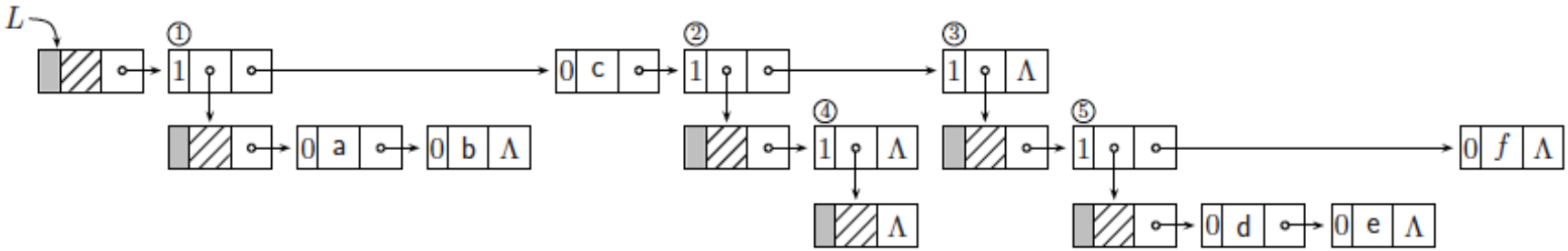
Linked Implementation

$$L = ((a,b),c,(()),((d,e),f))$$

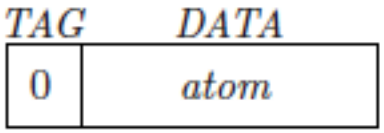


Linked Implementation

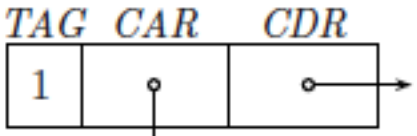
$$L = ((a,b),c,(()),((d,e),f))$$



Linked Implementation

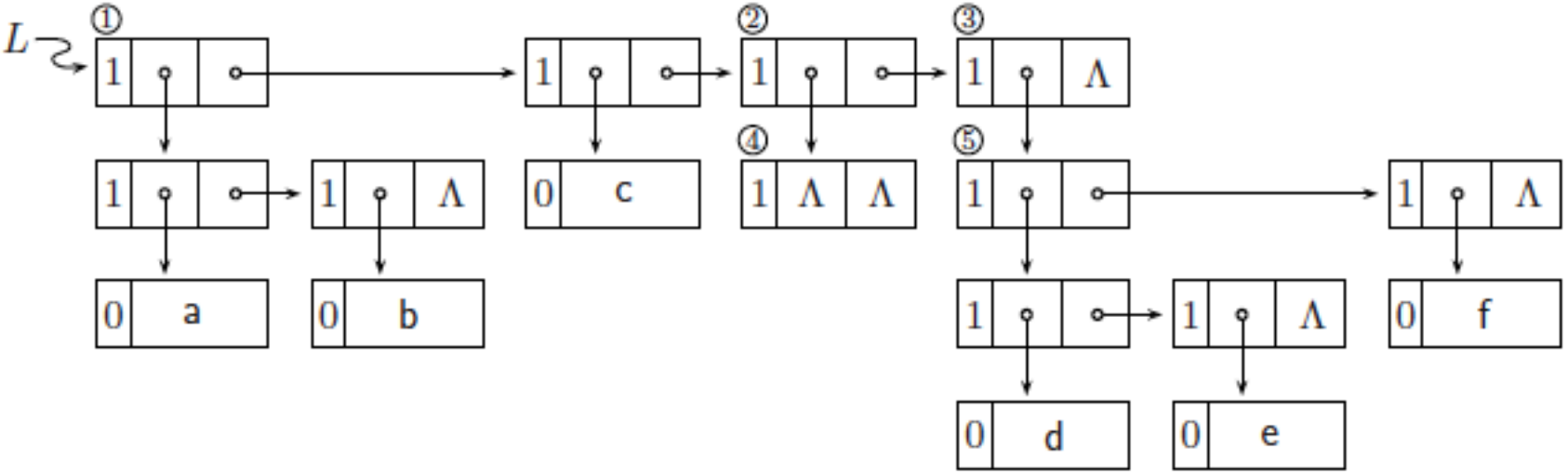


Atomic node



List node

$$L = ((a,b),c,(()),((d,e),f))$$



Operations

- Finding the head of a non-null list

```
procedure HEAD(L)
Returns a pointer to the head of a non-null list L
  if L = Λ or TAG(L) ≠ −1 then [output 'Error: argument is not a valid list'; stop]
  β ← LINK(L)
  case
    : β = Λ : [output 'Error: head is not defined for a null list'; stop]
    : β ≠ Λ : [if TAG(β) = 0 then return(β)
                             else return(DATA(β))]
  endcase
  end HEAD
```

Operations

- Finding the tail of a non-null list

procedure TAIL(L)
Returns a pointer to the tail of a non-null list L
if $L = \Lambda$ **or** $TAG(L) \neq -1$ **then** [**output** 'Error: argument is not a valid list'; **stop**]
$\beta \leftarrow LINK(L)$
case
: $\beta = \Lambda$: [**output** 'Error: tail is not defined for a null list'; **stop**]
: $\beta \neq \Lambda$: [**call** GETNODE(α); $TAG(\alpha) \leftarrow -1$; $LINK(\alpha) \leftarrow LINK(\beta)$; **return**($\alpha$)]
endcase
end TAIL

Operations

- Constructing a list given a head and a tail

procedure CONSTRUCT(H, T)
Constructs a list whose head is H and whose tail is T and returns a pointer to the list head of the constructed list.
if $H = \Lambda$ **or** $T = \Lambda$ **then** [**output** 'Error: invalid pointers'; **stop**]
if $TAG(T) \neq -1$ **then** [**output** 'Error: the tail is not a list'; **stop**]
call GETNODE(β)
case
: $TAG(H) = 0$: [$TAG(\beta) \leftarrow 0$; $DATA(\beta) \leftarrow DATA(H)$] ▷ *head is an atom*
: $TAG(H) = -1$: [$TAG(\beta) \leftarrow 1$; $DATA(\beta) \leftarrow H$] ▷ *head is a list*
endcase
$LINK(\beta) \leftarrow LINK(T)$
call GETNODE(α); $TAG(\alpha) \leftarrow -1$; $LINK(\alpha) \leftarrow \beta$ ▷ *list head of resulting list*
return(α)
end CONSTRUCT

Operations

- Pure list traversal

```
    procedure TRAVERSE_PURE_LIST(L)
    if L = Λ then return
    call InitStack(𝕊)
    α ← L
    loop
1:      if TAG(α) = 0 then call VISIT(DATA(α))
                      else [if DATA(α) = Λ then goto 2
                                           else [call PUSH(𝕊, α)
                                                 α ← DATA(α); goto 1] ]
2:      α ← LINK(α)
        if α = Λ then if IsEmptyStack(𝕊) then return
                                         else [call POP(𝕊, α); goto 2]
    forever
    end TRAVERSE_PURE_LIST
```

Outline

- Basic concepts
- Implementations
  - Linked Implementation
    - Basic
    - List heads
    - Alternative (LISP structure)
  - Operations
- **Application: Automatic Storage Reclamation**

Automatic Storage Reclamation

- Important task in list processing environments (mostly PL compilers)
- Complication: returning freed / unused nodes in reentrant or cyclic lists where several references to that node may exist
- ASR is usually implemented as a system function, e.g. in LISP
- Two principal methods
  - ***Reference-counter technique***
  - ***Garbage-collection technique***

Reference-counter

- A reference counter field (REF) is maintained in each node in which the number of references to the node is stored
- A node whose REF = 0 is returned to the avail list
  - REF value of the nodes pointed to by the freed node are likewise decremented by 1

Garbage-collection

- Unused nodes (i.e. garbage) are allowed to pile up and when avail list is empty, garbage collection begins.
- Two phases: ***marking*** and ***gathering***
  - ***marking: all nodes accessible to running programs are marked "in use"***
  - ***gathering: entire list space is scanned and all nodes not marked "in use" are linked together to comprise the avail list***

Marking algorithms

- Marking a cyclic list using stack of pointers

```
    procedure MARK_LIST_1(L)
    call InitStack(𝕊)
    α ← L
    loop
      MARK(α) ← 1
      if not ATOM(α) then [call PUSH(𝕊, CDR(α)); call PUSH(𝕊, CAR(α))]
1:    if IsEmptyStack(𝕊) then return
                         else [call POP(𝕊, α); if MARK(α) = 1 then goto 1]
    forever
    end MARK_LIST_1
```

Marking algorithms

- Schorr-Waite-Deustch Algorithm

```
procedure MARK_LIST_2(L)
    if ATOM(L) then [MARK(L) ← 1; return]
    β ← Λ; α ← L
Mark node
 1: MARK(α) ← 1
Process CAR path
    σ ← CAR(α)
    case
      : MARK(σ) = 1 : exit
      :    ATOM(σ)  : [MARK(σ) = 1; exit]
      :       else  : [CAR(α) ← β; β ← α; α ← σ; goto 1]
    endcase
Process CDR path
 2: σ ← CDR(α)
    case
      : MARK(σ) = 1 : exit
      :    ATOM(σ)  : [MARK(σ) = 1; exit]
      :       else  : [TAG(α) ← 1; CDR(α) ← β; β ← α; α ← σ; goto 1]
    endcase
Ascend via inverted links
 3: if β = Λ then return
            else if TAG(β) = 0 then [σ ← CAR(β); CAR(β) ← α
                                     α ← β; β ← σ; goto 2]
                                else [σ ← CDR(β); CDR(β) ← α; TAG(β) ← 0;
                                     α ← β; β ← σ; goto 3]
    end MARK_LIST_2
```

Marking algorithms

- Wegbreit Algorithm

```
procedure MARK_LIST_3(L)
    if ATOM(L) then [MARK(L) ← 1; return]
    call InitStack(S)
    β ← Λ; α ← L
Mark node
 1: MARK(α) ← 1
Process CAR path
    σ ← CAR(α)
    case
      : MARK(σ) = 1 : exit
      :   ATOM(σ)   : [MARK(σ) = 1; exit]
      :     else    : [call PUSH(S, 0); CAR(α) ← β; β ← α; α ← σ; goto 1]
    endcase
Process CDR path
 2: σ ← CDR(α)
    case
      : MARK(σ) = 1 : exit
      :   ATOM(σ)   : [MARK(σ) = 1; exit]
      :     else    : [call PUSH(S, 1); CDR(α) ← β; β ← α; α ← σ; goto 1]
    endcase
Ascend via inverted links
 3: if β = Λ then return
            else [call POP(S, flag)
                  if flag = 0 then [σ ← CAR(β); CAR(β) ← α
                                    α ← β; β ← σ; goto 2]
                              else [σ ← CDR(β); CDR(β) ← α
                                    α ← β; β ← σ; goto 3]]
    end MARK_LIST_3
```

Marking Algorithms: Note

	MARK_LIST_1	MARK_LIST_2	MARK_LIST_3
Executes in linear time	Yes	Yes	Yes
Executes in bounded workspace	No	Yes	No
Uses no tag bits	Yes	No	Yes

Gathering

procedure GATHER
$avail \leftarrow \Lambda$
for $\alpha \leftarrow m2 - c + 1$ **to** $m1$ **by** $-c$ **do**
  if $MARK(\alpha) = 0$ **then** [$CDR(\alpha) \leftarrow avail$; $avail \leftarrow \alpha$]
  else $MARK(\alpha) \leftarrow 0$
endfor
end GATHER

Thank you
for your attention.

Questions ...