Lists. Stacks. Queues

Mădălina Răschip, Cristian Gațu

Faculty of Computer Science
"Alexandru Ioan Cuza" University of Iași, Romania

DS 2018/2019

Content

Abstract data types: LLin, LLinOrd, Stacks, Queues

- Linear lists
- Array implementation
- Linked list implementation
- Ordered linear lists
- Stacks
- Queues

Application – arithmetic expression conversion

Content

- Abstract data types: LLin, LLinOrd, Stacks, Queues
    - Linear lists
    - Array implementation
    - Linked list implementation
    - Ordered linear lists
    - Stacks
    - Queues
- Application – arithmetic expression conversion

Linear lists – examples

- Students
  - *(Adriana, George, Luiza, Maria, Daniel)*
- Exams
  - *(Math, Logic, DS, CAOS, IP, ENG)*
- Week days
  - *(M, T, W, T, F, S, S)*
- Months
  - *(Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)*

Abstract data type LLin

- OBJECTS: $L = (e_0, \cdots, e_{n-1}), \quad n \geq 0$
- $e_i \in$ Elt (element abstract data type)
- Relations:

–	e_0	first element of the list;
–	e_{n-1}	last element of the list;
–	e_i	predecessor of e_{i+1}.

LLin – operations

- `emptyList()`
  - input: empty
  - output: $L = ()$ (empty list)
- `insert()`
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad k \in \mathtt{Nat}, \quad e \in \mathtt{Elt}$
  - output:
    - $L = (\cdots, e_{k-1}, e, e_k, \cdots)$, if $0 \leq k \leq n$
    - error otherwise

`insert()` – examples

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

- $\texttt{insert}(L, 0, x) \quad \Rightarrow \quad L = (x, a, b, c, d, e, f, g)$

  Obs. the index of elements $a, \cdots, g$ is incremented by 1.

- $\texttt{insert}(L, 2, x) \quad \Rightarrow \quad L = (a, b, x, c, d, e, f, g)$

- $\texttt{insert}(L, 7, x) \quad \Rightarrow \quad L = (a, b, c, d, e, f, g, x)$

- $\texttt{insert}(L, 10, x) \quad \Rightarrow$ error

- $\texttt{insert}(L, -7, x) \quad \Rightarrow$ error

`LLin` – operations

- `delete()`
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad k \in \texttt{Nat}$
  - output:
    - $L = (\cdots, e_{k-1}, e_{k+1}, \cdots)$, if $0 \leq k \leq n-1$
    - error otherwise

LLin – operations

- delete()
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad k \in \mathtt{Nat}$
  - output:
    - $L = (\cdots, e_{k-1}, e_{k+1}, \cdots)$, if $0 \leq k \leq n-1$
    - error otherwise

Examples:

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

- delete$(L, 2) \quad \Rightarrow \quad L = (a, b, d, e, f, g)$
  Obs. the index of elements $d, \cdots, g$ is decremented by 1.
- delete$(L, 10) \quad \Rightarrow \quad$ error
- delete$(L, -7) \quad \Rightarrow \quad$ error

`LLin` – operations

- `theKth()`
    - input:
        - $L = (e_0, \cdots, e_{n-1}), \quad k \in \mathtt{Nat}$
    - output:
        - e_k, if $0 \leq k \leq n-1$
        - error otherwise

LLin – operations

- theKth()
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad k \in \texttt{Nat}$
  - output:
    - e_k, if $0 \leq k \leq n-1$
    - error otherwise

Examples:

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

- $\texttt{theKth}(L, 0) \quad \Rightarrow \quad a$
- $\texttt{theKth}(L, 2) \quad \Rightarrow \quad c$
- $\texttt{theKth}(L, 6) \quad \Rightarrow \quad g$
- $\texttt{theKth}(L, 20) \quad \Rightarrow$ error
- $\texttt{theKth}(L, -2) \quad \Rightarrow$ error

LLin – operations

- deleteALLe()
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad e \in \mathtt{Elt}$
  - output:
    - The L list where all occurrences of e have been deleted.

LLin – operations

- deleteALLe()
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad e \in \texttt{Elt}$
  - output:
    - The L list where all occurrences of e have been deleted.

Examples:

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

- deleteALLe$(L, a) \quad \Rightarrow \quad (b, c, b, c)$
- deleteALLe$(L, c) \quad \Rightarrow \quad (a, b, a, b, a)$
- deleteALLe$(L, d) \quad \Rightarrow \quad (a, b, c, a, b, c, a)$

`LLin` – operations

- `iterate()`
  - input:
    - $L = (e_0, \cdots, e_{n-1})$, a procedure / function `visit()`
  - output:
    - The L list where all the elements have been processed by `visit()`

LLin – operations

- `iterate()`
  - input:
    - $L = (e_0, \cdots, e_{n-1})$, a procedure / function `visit()`
  - output:
    - The L list where all the elements have been processed by `visit()`

Examples:

$$L = (1, 2, 3, 1, 2, 3)$$

- $\texttt{iterate}(L, \texttt{twoTimes()}) \quad \Rightarrow \quad (2, 4, 6, 2, 4, 6)$
- $\texttt{iterate}(L, \texttt{increment()}) \quad \Rightarrow \quad (2, 3, 4, 2, 3, 4)$

LLin – operations

- pos()
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad e \in \texttt{Elt},$
  - output:
    - the position of the first occurence of e in L or
    - -1 if e does not appear in L.

$\texttt{LLin}$ – operations

- $\texttt{pos()}$
  - input:
    - $L = (e_0, \cdots, e_{n-1}), \quad e \in \texttt{Elt},$
  - output:
    - the position of the first occurence of e in L or
    - -1 if e does not appear in L.

Examples:

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

- $\texttt{pos}(L, a) \quad \Rightarrow \quad 0$
- $\texttt{pos}(L, c) \quad \Rightarrow \quad 2$
- $\texttt{pos}(L, d) \quad \Rightarrow \quad 6$
- $\texttt{pos}(L, x) \quad \Rightarrow \quad -1$

LLin – operations

- `length()`
  - input:
    - $L = (e_0, \cdots, e_{n-1})$,
  - output:
    - n – the length of L list.

`LLin` – operations

- `length()`
    - input:
        - $L = (e_0, \cdots, e_{n-1})$,
    - output:
        - n – the length of L list.

Example:

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

- $\texttt{length}(L) \quad \Rightarrow \quad 7$

Content

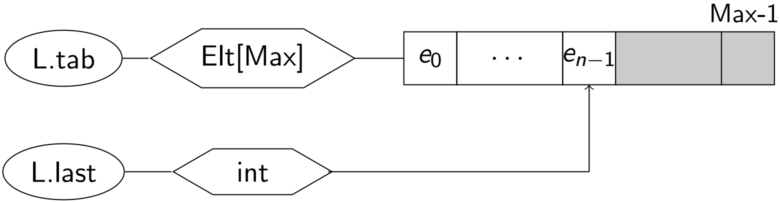
Abstract data types: LLin, LLinOrd, Stacks, Queues

- Linear lists
- Array implementation
- Linked list implementation
- Ordered linear lists
- Stacks
- Queues

Application – arithmetic expression conversion

LLin – array implementation

- Object representation: $L = (e_0, \cdots, e_{n-1})$



- L is a *structure*
  - $L.tab$ – an array field that stores the list elements;
  - $L.last$ – a numeric field that stores the last element position.

LLin – array implementation

- `insert()`
  - shift right by one position the elements of indices $k, k+1, \ldots;$
  - insert e on the position k;
  - exceptions:
    - $k < 0, \quad k > L.last + 1 \quad (n)$
    - $L.last = Max - 1.$

LLin – array implementation

```
procedure insert(L, k, e)
begin
    if (k < 0 or k > L.last + 1) then
        throw "error-wrong position"
    if (L.last >= Max − 1) then
        throw "error-not enough memory"
    for j ← L.last downto k do
        L.tab[j + 1] ← L.tab[j]
    L.tab[k] ← e
    L.last ← L.last + 1
end
```

LLin – array implementation

```
procedure insert(L, k, e)
begin
    if (k < 0 or k > L.last + 1) then
        throw "error-wrong position"
    if (L.last >= Max − 1) then
        throw "error-not enough memory"
    for j ← L.last downto k do
        $L.tab[j+1] \leftarrow L.tab[j]$
    $L.tab[k] \leftarrow e$
    $L.last \leftarrow L.last + 1$
end
```

- Running time: $O(n)$.

LLin – array implementation

- iterate()

```
procedure iterate(L, visit())
begin
    for $i \leftarrow 0$ to $L.last$ do
        visit($L.tab[i]$)
end
```

- If $visit()$ processes one element in $O(1)$, then $iterate()$ processes the list in $O(n)$ (n the list length).

Content

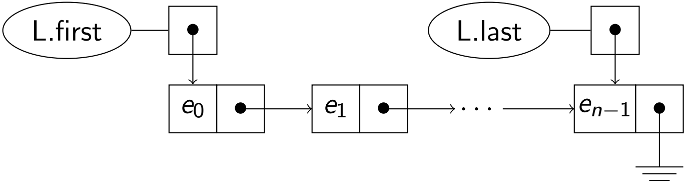
Abstract data types: LLin, LLinOrd, Stacks, Queues

- Linear lists
- Array implementation
- Linked list implementation
- Ordered linear lists
- Stacks
- Queues

Application – arithmetic expression conversion

LLin – linked list implementation

- Object representation: $L = (e_0, \cdots, e_{n-1})$



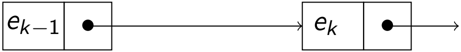
- *L* is a *structure* with two fields
  - *L.first* – pointer to the first element;
  - *L.last* – pointer to the last element.
- a `nod * p` (stored at the address in `p`) has two fields:
  - $p->elt(= e_i)$ – stores the node information;
  - $p->succ$ – stores the address of the next node.

LLin – linked list implementation

- insert()
  - iterate elements of position $0, 1, \ldots, k-1$;
  - insert a new element afther the $(k-1)$-th;
    - create a new node; **new**(q)
    - fill the information;
    - update links.
  - exceptions:
    - empty list;
    - $k = 0$;
    - $k = n$;
    - $k < 0, k > n$.

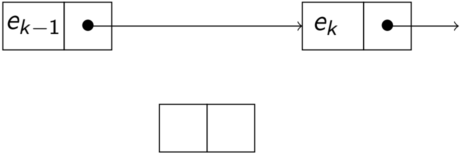
LLin – linked list implementation

- General case



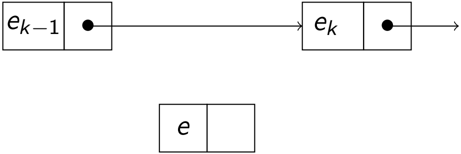
LLin – linked list implementation

- General case



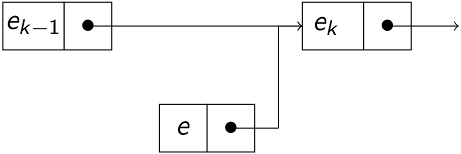
LLin – linked list implementation

- General case



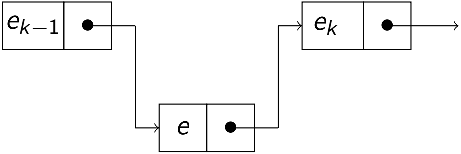
LLin – linked list implementation

- General case



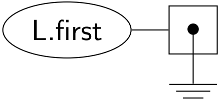
LLin – linked list implementation

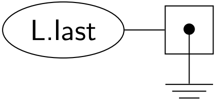
- General case



LLin – linked list implementation

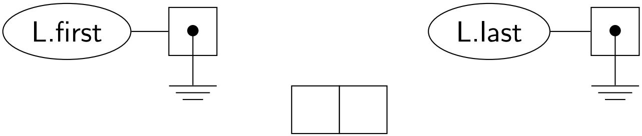
- Special case: empty list





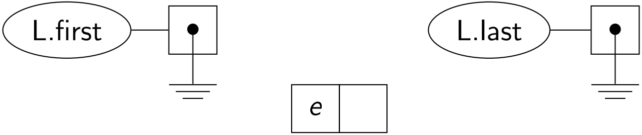
LLin – linked list implementation

- Special case: empty list



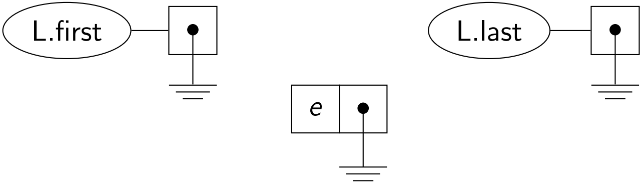
LLin – linked list implementation

- Special case: empty list



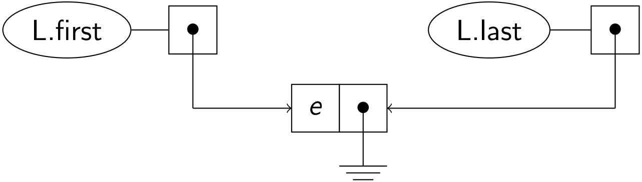
LLin – linked list implementation

- Special case: empty list



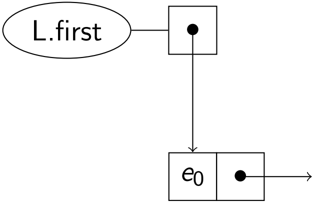
LLin – linked list implementation

- Special case: empty list



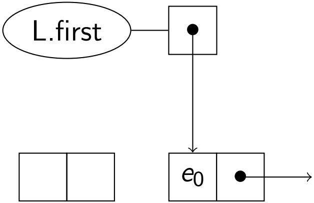
LLin – linked list implementation

- Special case: insert as first element



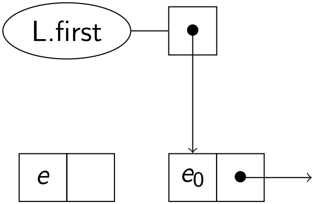
LLin – linked list implementation

- Special case: insert as first element



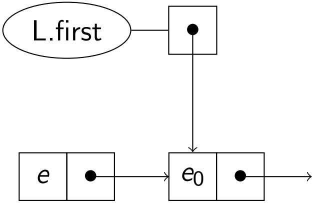
LLin – linked list implementation

- Special case: insert as first element



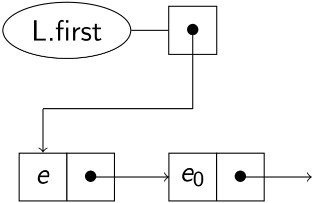
LLin – linked list implementation

- Special case: insert as first element



`LLin` – linked list implementation

- Special case: insert as first element



LLin – linked list implementation

```
procedure $insert(L, k, e)$
begin
    if $(k < 0)$ then
        throw "error-wrong position"
    new($q$);    $q->elt \leftarrow e$
    if $(k == 0$ or $L.first == NULL)$ then
        $q->succ \leftarrow L.first$;    $L.first \leftarrow q$
        if $(L.last == NULL)$ then
            $L.last \leftarrow q$
    else
        $p \leftarrow L.first$;    $j \leftarrow 0$
        while $(j < k - 1$ and $p! = L.last)$ do
            $p \leftarrow p->succ$;    $j \leftarrow j + 1$
        if $(j < k - 1)$ then
            throw "error-wrong position"
        $q->succ \leftarrow p->succ$;    $p->succ \leftarrow q$
        if $(p == L.last)$ then
            $L.last \leftarrow q$
end
```

LLin – application

- Polygonal line.
    - Point: structure with two fields x and y;
    - list building

```
procedure buildList(L)
begin
    $L \leftarrow emptyList()$
    /* read $n$ */
    for $i \leftarrow 0$ to $n - 1$ do
        /* read $p.x$, $p.y$ */
        $insert(L, 0, p)$
end
```

- Obs. Running time depends on the implementation.

LLin – application

- Muliply by 2 one point coordinates:

```
procedure times2Point(p)
begin
    $p.x \leftarrow p.x * 2$
    $p.y \leftarrow p.y * 2$
end
```

- Muliply by 2 the coordinates of a polygonal line:

```
procedure times2Line(p)
begin
    $iterate(L, times2Point())$
end
```

LLin – application

- translate point:

```
procedure trPoint(p, dx, dy)
begin
    $p.x \leftarrow p.x + dx$
    $p.y \leftarrow p.y + dy$
end
```

- translate polygonal line:

```
procedure trLine(L, dx, dy)
begin
    $iterate(L, trPoint())$
end
```

Content

Abstract data types: LLin, LLinOrd, Stacks, Queues

- Linear lists
- Array implementation
- Linked list implementation
- Ordered linear lists
- Stacks
- Queues

Application – arithmetic expression conversion

Ordered linear lists: LLinOrd

- OBJECTS:
  $L = (e_0, \cdots, e_{n-1}), \quad n \geq 0, \quad e_i \in \texttt{Elt}, \quad e_0 \leq e_1 \leq \cdots \leq e_{n-1}$
- Operations:
  - emptyList()
    - input: empty
    - output: $L = ()$ (empty list)
  - insert()
    - input: $L = (e_0, \ldots, e_{n-1}), \quad e \in \texttt{Elt}$
    - output: $L = (\cdots, e_{k-1}, e, e_k, \cdots)$, if $e_{k-1} \leq e \leq e_k$
      $(e_{-1} = -\infty, e_n = +\infty)$

Ordered linear lists: `LLinOrd`

- `delete()`
    - input: $L = (e_0, \ldots, e_{n-1}), \quad e \in \texttt{Elt}$
    - output: $L = (\cdots, e_{k-1}, e_{k+1}, \cdots)$, if $e = e_k$
      error otherwise
- `theKth()`
- `iterate()`
- `pos()`

LLinOrd – array implementation

```
function $pos(L, e)$
begin
    $p \leftarrow 0$;    $q \leftarrow L.last$
    $m \leftarrow (p + q)/2$
    while $(L.tab[m]! = e$ and $p < q)$ do
        if $(e < L.tab[m])$ then
            $q \leftarrow m$–$1$
        else
            $p \leftarrow m + 1$
        $m \leftarrow (p + q)/2$
    if $(L.tab[m] == e)$ then
        return $m$
    else
        return $-1$
end
```

LLinOrd – searching complexity

- Array implementation: $O(\log_2 n)$;
- Linked list implementation: $O(n)$ (linear search).

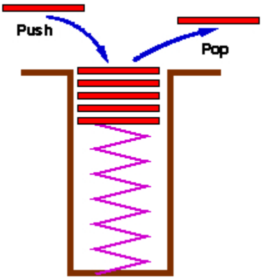
Content

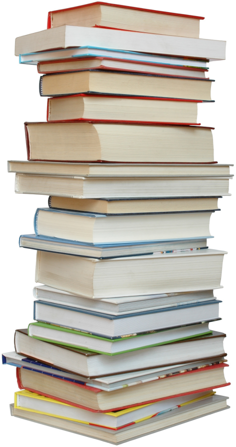
Abstract data types: LLin, LLinOrd, Stacks, Queues

- Linear lists
- Array implementation
- Linked list implementation
- Ordered linear lists
- Stacks
- Queues

Application – arithmetic expression conversion

Stack





Stack – applications

- Direct applications
  - web browser page history;
  - “undo” sequence in a text editor;
  - recursive calls of a subprogram.
- Indirect applications
  - Auxiliary data structure in certain algorithms;
  - Other data structures component.

Abstract data type `Stack`

- Objects:
  Lists where the element age is known:
  LIFO lists (*Last-In-First-Out*).

- Operations:
  - `emptyStack()`
    - input: empty
    - output: $S = ()$ (empty stack)
  - `isEmpty()`
    - input: $S \in$ `Stack`
    - output:
      – **true** if S is empty;
      – **false** if S is not empty.

Abstract data type Stack

- Operations:
  - push()
    - input: $S \in$ Stack, $e \in$ Elt
    - output: S where e has been added as the last element (the newest).
  - pop()
    - input: $S \in$ Stack
    - output:
      – S where the last introduced element has been deleted (the newest);
      – error if S is empty.
  - top()
    - input: $S \in$ Stack
    - output:
      – the last introduced in S element (the newest);
      – error if S is empty.

Stack – list implementation

ADT Stack		ADT LLin
$push(S, e)$	$=$	$insert(S, 0, e)$
$pop(S, e)$	$=$	$delete(S, 0)$
$top(S)$	$=$	$theKth(S, 0)$

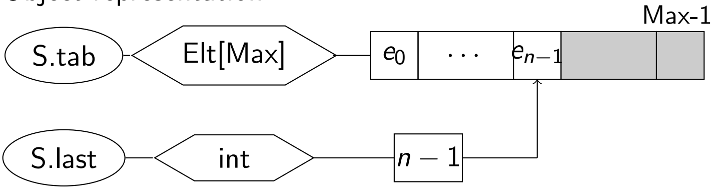
or

ADT Stack		ADT LLin
$push(S, e)$	$=$	$insert(S, length(S), e)$
$pop(S, e)$	$=$	$delete(S, length(S) - 1)$
$top(S)$	$=$	$theKth(S, length(S) - 1)$

ADT — Abstract Data Type

Stack – array implementation

- Object representation

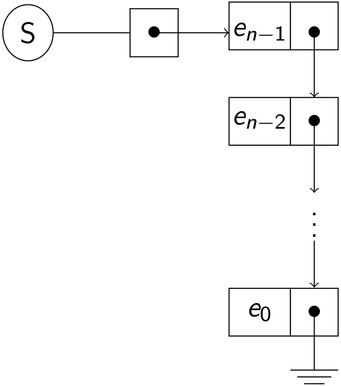


- implementation

```
procedure $push(S, e)$
begin
    if $S.last == Max - 1$ then
        throw "error"
    else
        $S.last \leftarrow S.last + 1$
        $S.tab[last] \leftarrow e$
end
```

Stack – linked list implementation

- Object representation



Stack – linked list implementation

- Implementation
  - push()

```
procedure push(S, e)
begin
    new(q)
    $q->elt \leftarrow e$
    $q->succ \leftarrow S$
    $S \leftarrow q$
end
```

  - pop()

```
procedure pop(S)
begin
    if $S == NULL$ then
        throw "error"
    $q \leftarrow S$
    $S \leftarrow S->succ$
delete(q) end
```

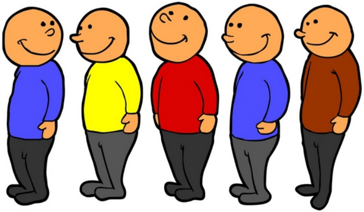
Content

Abstract data types: LLin, LLinOrd, Stacks, Queues

- Linear lists
- Array implementation
- Linked list implementation
- Ordered linear lists
- Stacks
- Queues

Application – arithmetic expression conversion

Queues



Queue – applications

- Direct applications
  - waiting lists / threads;
  - shared resources access.
    Example: printers.
- Indirect applications
  - Auxiliary data structure in certain algorithms.

Tipul abstract Queue

- OBJECTS:
  Lists where the age elements is known:
  FIFO lists (*First-In-First-Out*).

- Operations:
  - emptyQueue()
    - input: empty
    - output: $C = ()$ (empty queue)
  - isEmpty()
    - input: $C \in$ Queue
    - output:
      – **true** if C is empty;
      – **false** if C is not empty.

Abstract data type Queue

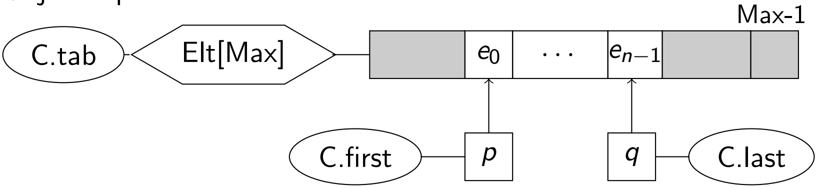
- Operations:
  - `insert()`
    - input: $C \in \texttt{Queue}, \quad e \in \texttt{Elt}$
    - output: C where e has been added as the last element (newest).
  - `delete()`
    - input: $C \in \texttt{Queue}$
    - output:
      – C whehre the first added element has been deleted (oldest);
      – error if C is empty.
  - `read()`
    - input: $C \in \texttt{Queue}$
    - output:
      – the first introduced element in C (oldest);
      – error if C is empty.

`Queue` – list implementation

ADT `Queue`		ADT `LLin`
$insert(C, e)$	$=$	$insert(C, length(C), e)$
$delete(C)$	$=$	$delete(C, 0)$
$read(S)$	$=$	$theKth(C, 0)$

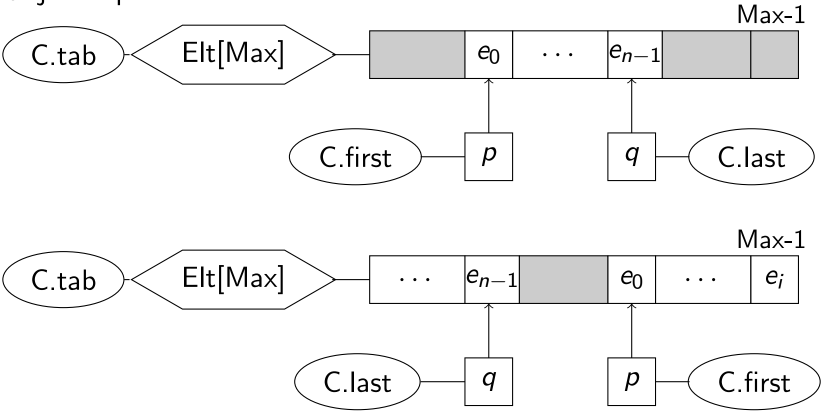
Queue – array implementation

- Object representation



Queue – array implementation

- Object representation



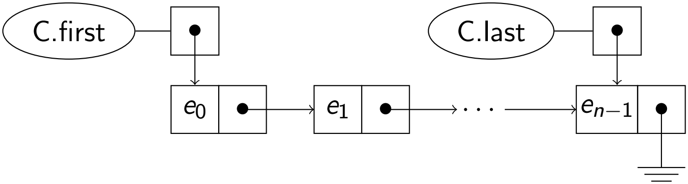
Queue – array implementation

- Implementation
  - `insert()`

```
procedure insert(C, e)
begin
    if $(C.last + 1)\%Max == C.first$ then
        throw "error"
    else
        $C.last \leftarrow (C.last + 1)\%Max$
        $C.tab[last] \leftarrow e$
end
```

Queue – linked list implementation

- Object representation



Queue – linked list implementation

- Implementation
    - insert()

```
procedure $insert(C, e)$
begin
    new(q)
    $q- > elt \leftarrow e$
    $q- > succ \leftarrow NULL$
    if $C.last == NULL$ then
        $C.first \leftarrow q$
        $C.last \leftarrow q$
    else
        $C.last- > succ \leftarrow q$
        $C.last \leftarrow q$
end
```

Content

Abstract data types: LLin, LLinOrd, Stacks, Queues

- Linear lists
- Array implementation
- Linked list implementation
- Ordered linear lists
- Stacks
- Queues

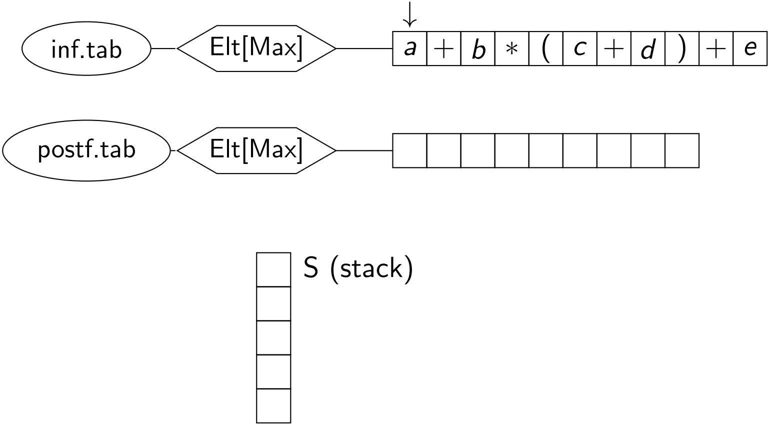
Application – arithmetic expression conversion

Application – postfix notation

- infix notation
  - $a + b$
  - $a + (b * 2)$
- postfix notation
  - $a\ b\ +$
  - $a\ b\ 2\ \ * +$
- priority rules
  - $a + b * 2$
- association rules: $7/3 * 2$
  - left: $(7/3) * 2$
  - right: $7/(3 * 2)$

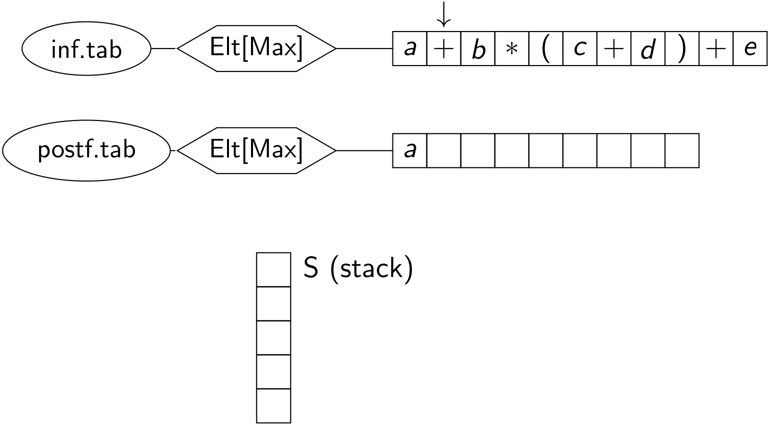
Infix → postix conversion

Example: $a + b * (c + d) + e$



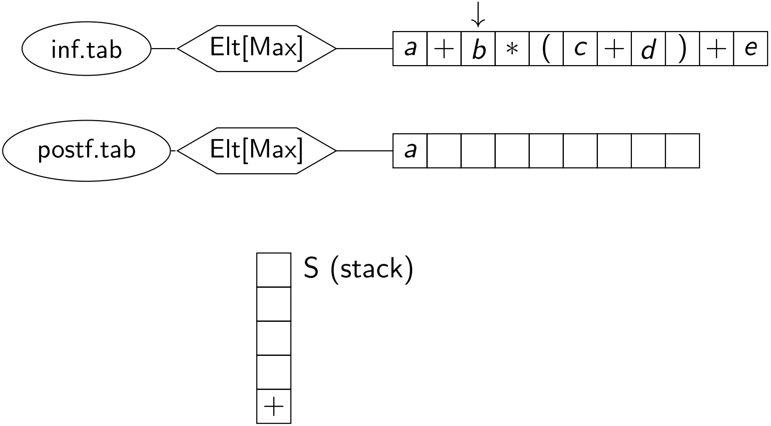
Infix → postix conversion

Example: $a + b * (c + d) + e$



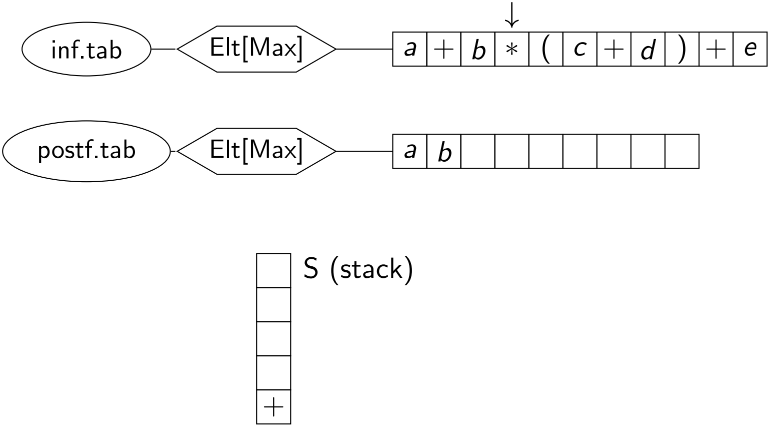
Infix $\rightarrow$ postix conversion

Example: $a + b * (c + d) + e$



Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab — Elt[Max]

a	+	b	*	(	c	+	d	)	+	e

postf.tab — Elt[Max]

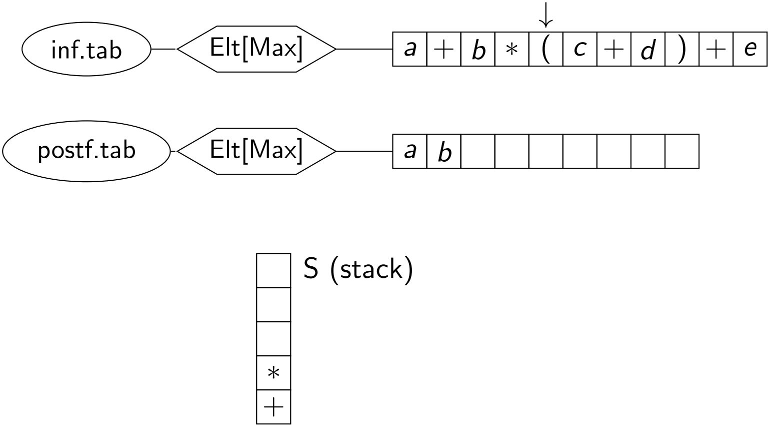
a	b							

S (stack)

+

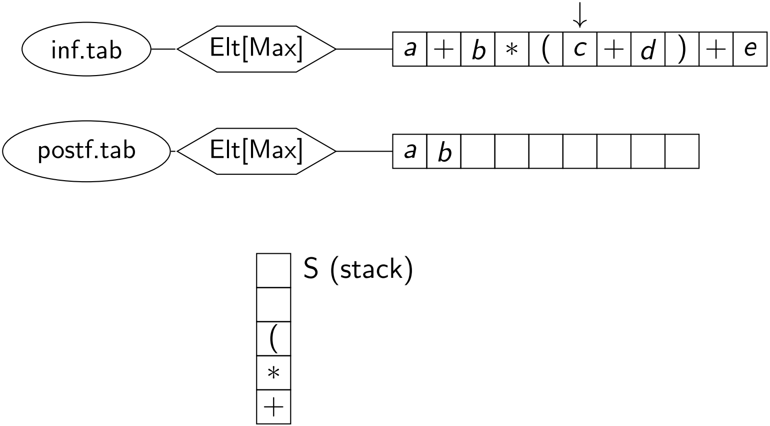
Infix → postix conversion

Example: $a + b * (c + d) + e$



Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab — Elt[Max] —

a	$+$	b	$*$	(	c	$+$	d	)	$+$	e

postf.tab — Elt[Max] —

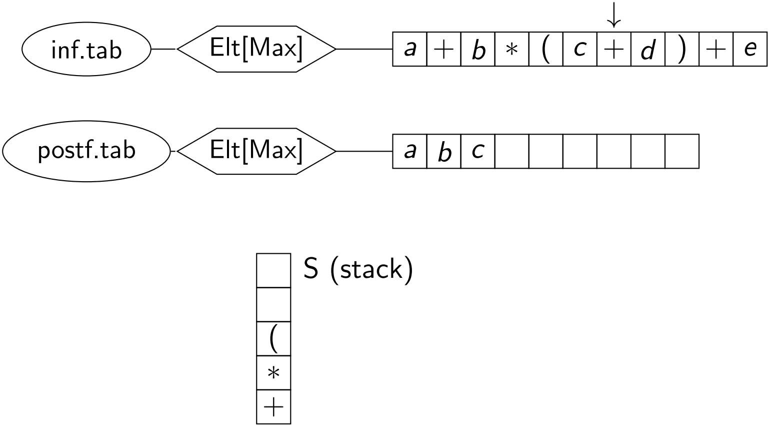
a	b							

S (stack)

(
$*$
$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab — Elt[Max] — | a | $+$ | b | $*$ | $($ | c | $+$ | d | $)$ | $+$ | e |

postf.tab — Elt[Max] — | a | b | c | | | | | | |

S (stack)

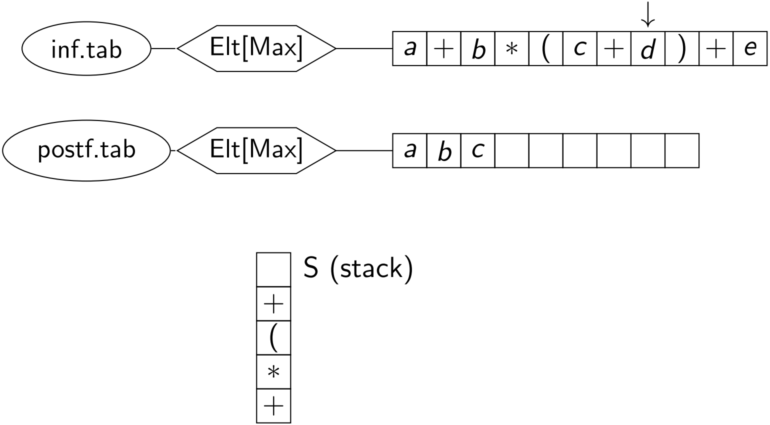
$($

$*$

$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab

Elt[Max]

a	$+$	b	$*$	$($	c	$+$	d	$)$	$+$	e

postf.tab

Elt[Max]

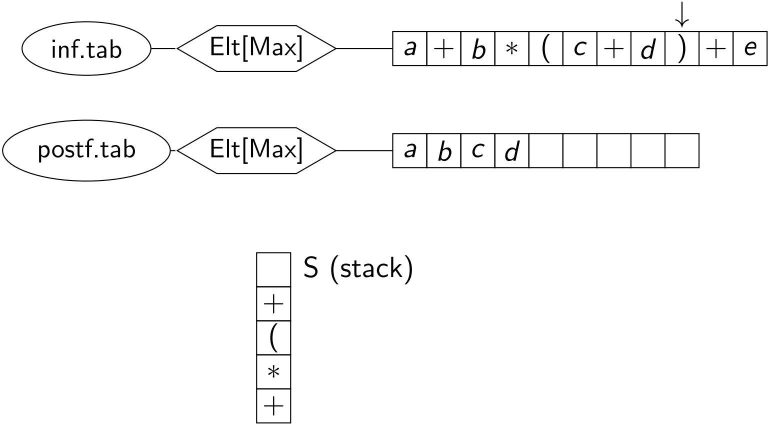
a	b	c						

S (stack)

$+$
$($
$*$
$+$

Infix $\rightarrow$ postix conversion

Example: $a + b * (c + d) + e$



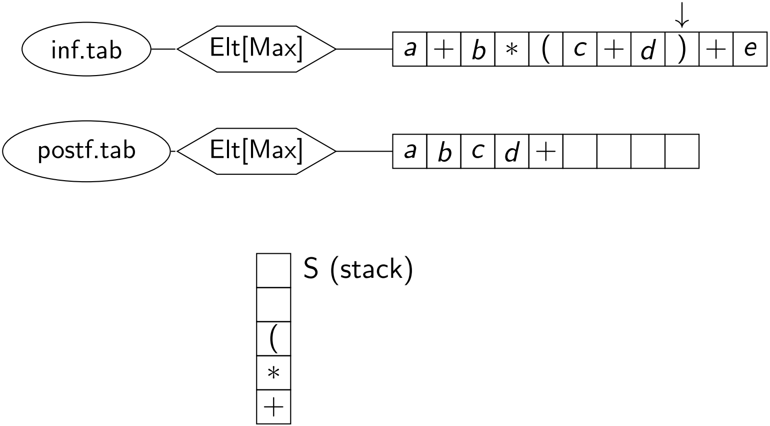
inf.tab — Elt[Max] — | a | $+$ | b | $*$ | $($ | c | $+$ | d | $)$ | $+$ | e |

postf.tab — Elt[Max] — | a | b | c | d | | | | | |

S (stack): | | $+$ | $($ | $*$ | $+$ |

Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab

Elt[Max]

↓ at)

a	$+$	b	$*$	$($	c	$+$	d	$)$	$+$	e

postf.tab

Elt[Max]

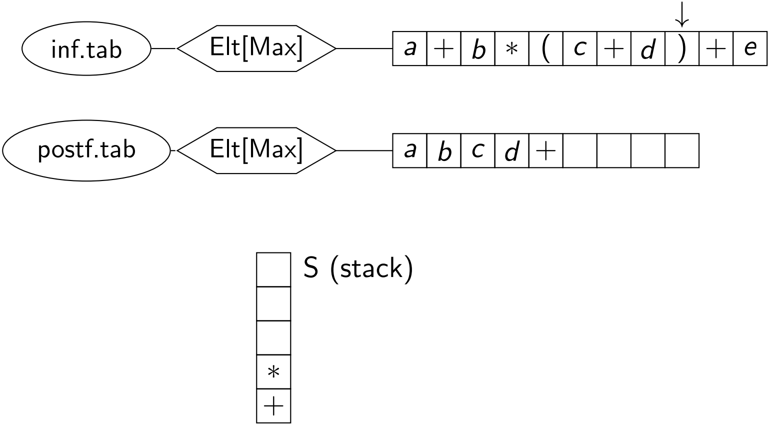
a	b	c	d	$+$				

S (stack)

$($
$*$
$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



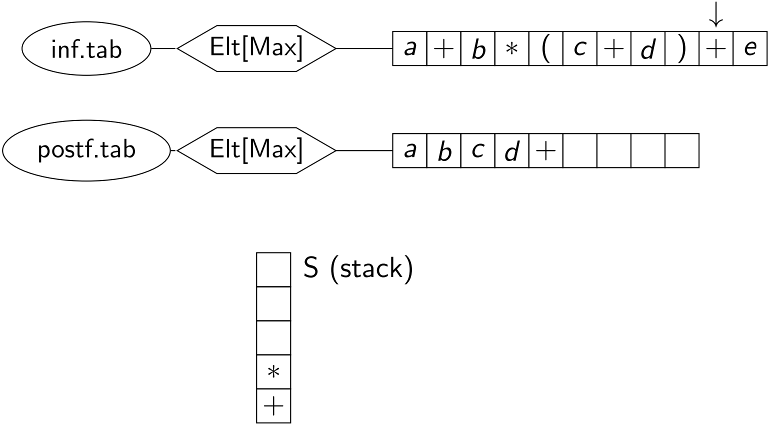
a	$+$	b	$*$	(	c	$+$	d	)	$+$	e

a	b	c	d	$+$				

$*$
$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab — Elt[Max] — a | $+$ | b | $*$ | (| c | $+$ | d |) | $+$ | e

↓

postf.tab — Elt[Max] — a | b | c | d | $+$ | | | |

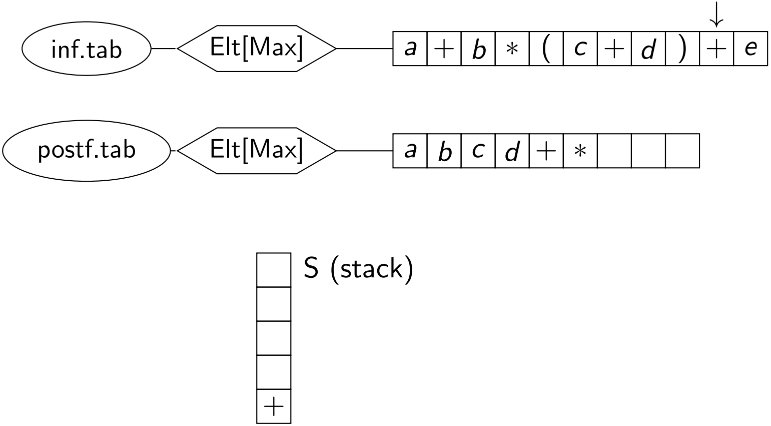
S (stack)

$*$

$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab — Elt[Max] — a | $+$ | b | $*$ | $($ | c | $+$ | d | $)$ | $+$ | e

$\downarrow$

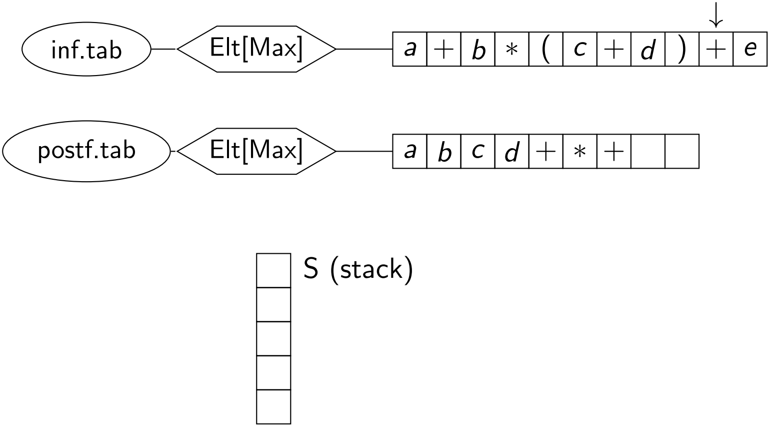
postf.tab — Elt[Max] — a | b | c | d | $+$ | $*$ | | |

S (stack)

$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab

Elt[Max]

									↓	
a	$+$	b	$*$	(	c	$+$	d	)	$+$	e

postf.tab

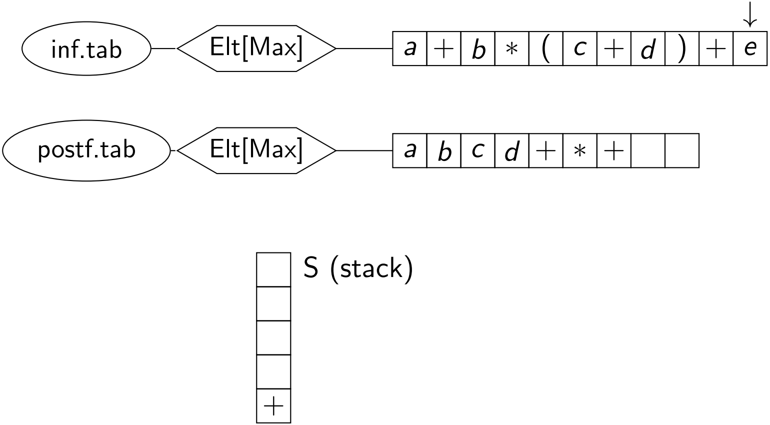
Elt[Max]

a	b	c	d	$+$	$*$	$+$		

S (stack)

Infix → postix conversion

Example: $a + b * (c + d) + e$



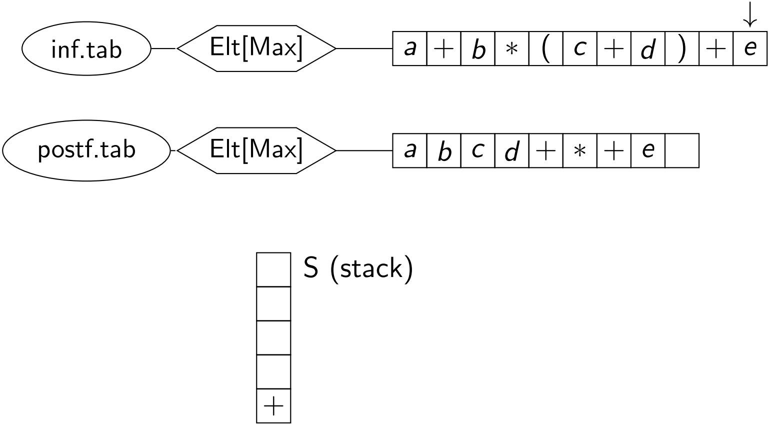
a	$+$	b	$*$	$($	c	$+$	d	$)$	$+$	e

a	b	c	d	$+$	$*$	$+$		

$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



inf.tab — Elt[Max] — a | $+$ | b | $*$ | (| c | $+$ | d |) | $+$ | e

↓

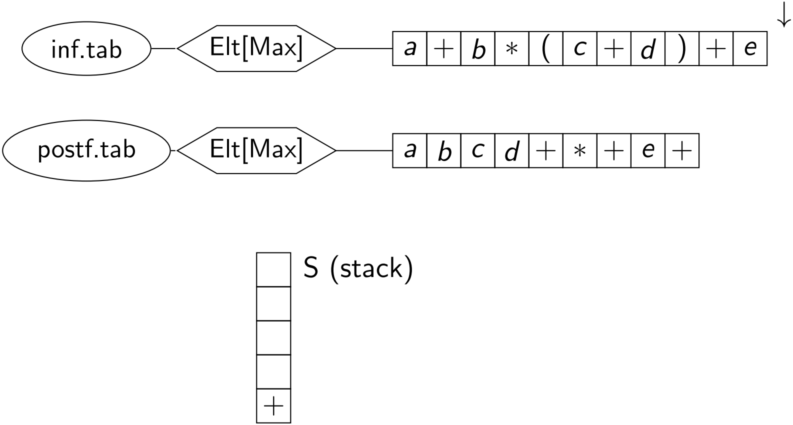
postf.tab — Elt[Max] — a | b | c | d | $+$ | $*$ | $+$ | e |

S (stack)

$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



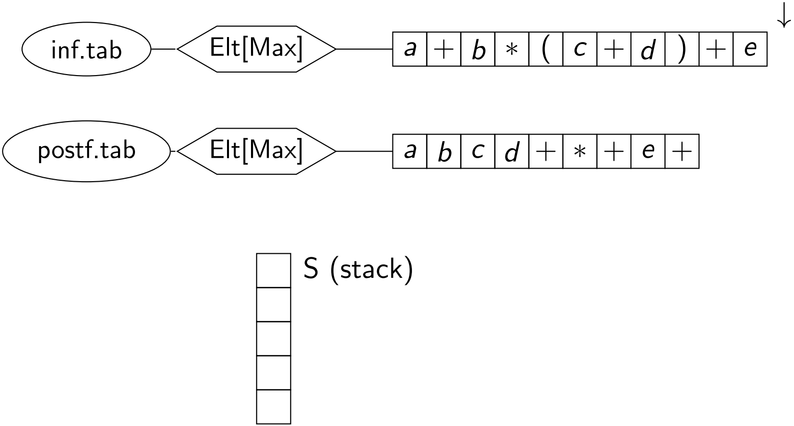
a	$+$	b	$*$	(	c	$+$	d	)	$+$	e

a	b	c	d	$+$	$*$	$+$	e	$+$

$+$

Infix → postix conversion

Example: $a + b * (c + d) + e$



a	$+$	b	$*$	(	c	$+$	d	)	$+$	e

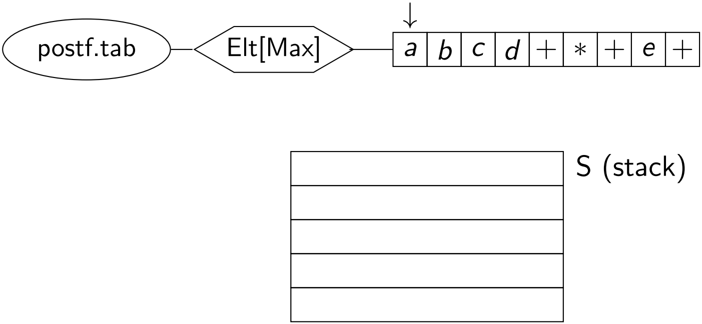
a	b	c	d	$+$	$*$	$+$	e	$+$

Infix → postfix conversion

```
procedure convInfix2Postfix(infix, postfix)
/* infix și postfix sunt cozi*/
begin
    $S \leftarrow emptyStack()$
    while (not $isEmpty(infix)$) do
        $x \leftarrow read(infix)$;    $delete(infix)$
        if $(operand(x))$ then
            $insert(postfix, x)$
        else
            if $(x ==')')$ then
                while $(top(s)! =' (')$ do
                    $insert(postfix, top(S))$;    $pop(S)$
                $pop(S)$
            else
                while (not $isEmpty(S)$ and $top(S)! =' ('$ and
                  $priorit(top(S)) >= priorit(x)$) do
                    $insert(postfix, top(S))$;    $pop(S)$
                $push(S, x)$
    while (not $isEmpty(S)$) do
        $insert(postfix, top(S))$;    $pop(S)$
end
```

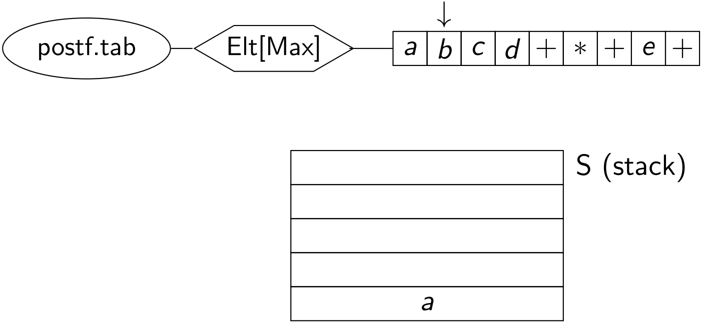
Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



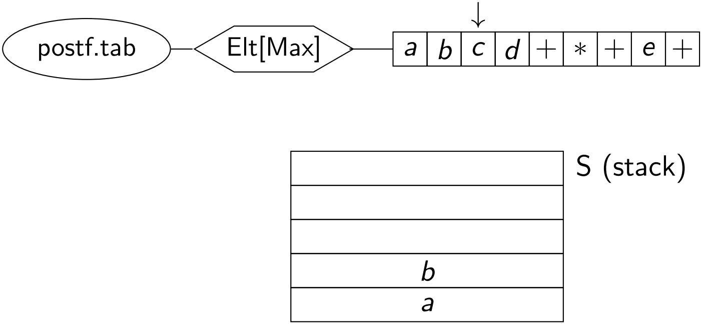
Postfix expression evaluation

Example: $a\ b\ c\ d\ + * + e +$



Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



postf.tab

Elt[Max]

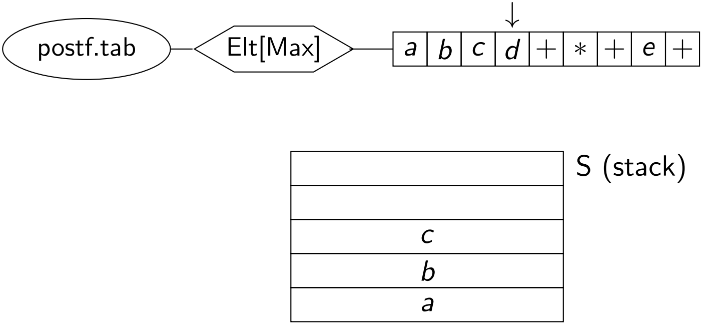
a	b	c	d	$+$	$*$	$+$	e	$+$

S (stack)

b
a

Postfix expression evaluation

Example: $a\ b\ c\ d\ + * + e +$



postf.tab

Elt[Max]

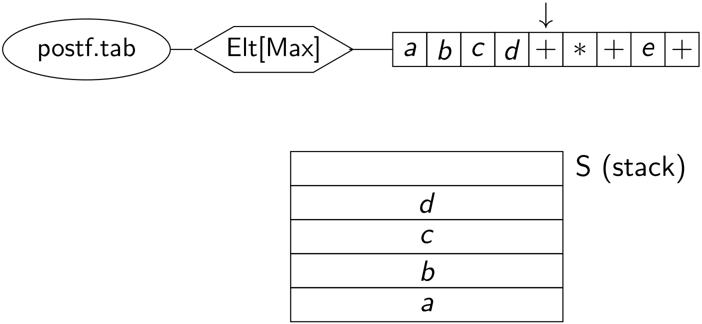
a	b	c	d	$+$	$*$	$+$	e	$+$

S (stack)

c
b
a

Postfix expression evaluation

Example: $a\ b\ c\ d\ + * + e +$



postf.tab — Elt[Max] —

a	b	c	d	$+$	$*$	$+$	e	$+$

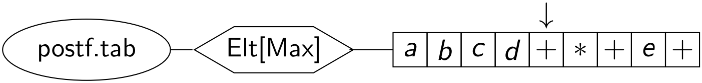
↓ (over the 5th cell, $+$)

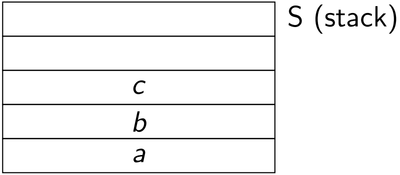
S (stack)

d
c
b
a

Postfix expression evaluation

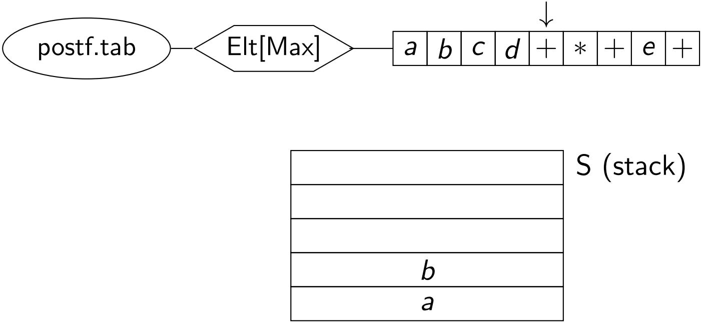
Example: $a\ b\ c\ d\ + * + e +$





Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e+$



postf.tab

Elt[Max]

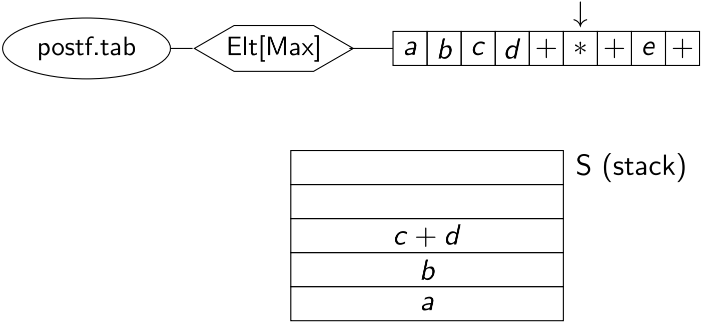
a	b	c	d	$\downarrow$ $+$	$*$	$+$	e	$+$

S (stack)

b
a

Postfix expression evaluation

Example: $a\ b\ c\ d\ + * + e +$

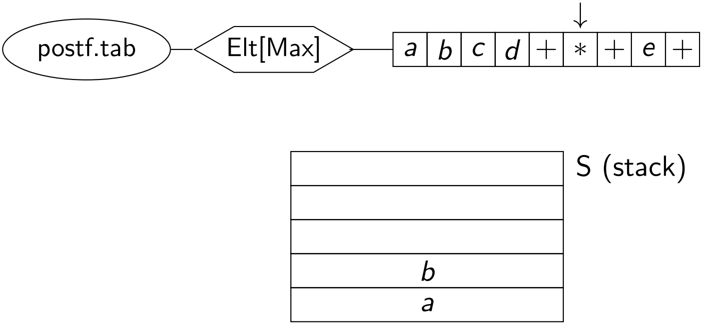


a	b	c	d	$+$	$*$	$+$	e	$+$

$c + d$
b
a

Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



postf.tab

Elt[Max]

a	b	c	d	$+$	$*$	$+$	e	$+$

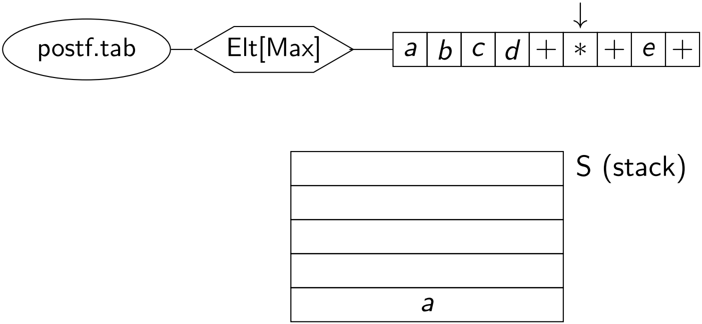
S (stack)

b

a

Postfix expression evaluation

Example: $a \ b \ c \ d \ \ + * + e +$



postf.tab

Elt[Max]

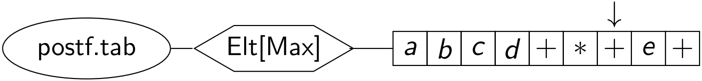
a	b	c	d	$+$	$*$	$+$	e	$+$

S (stack)

a

Postfix expression evaluation

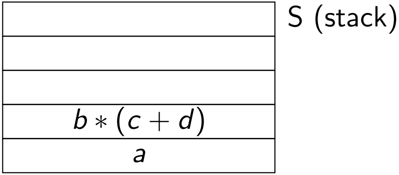
Example: $a \ b \ c \ d \ \ + * + \ e +$



postf.tab

Elt[Max]

a	b	c	d	$+$	$*$	$+$	e	$+$

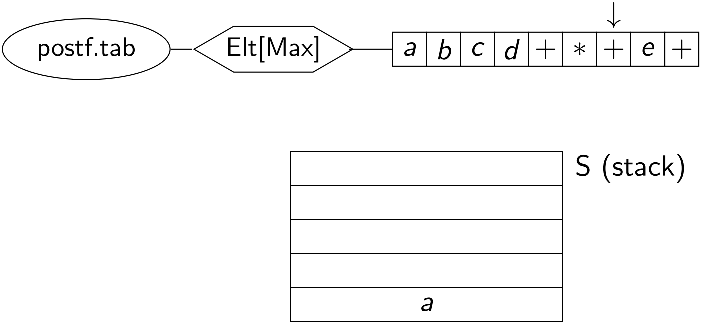


S (stack)

$b * (c + d)$
a

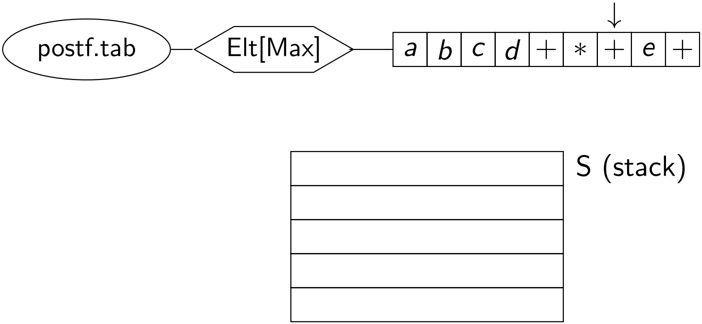
Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



Postfix expression evaluation

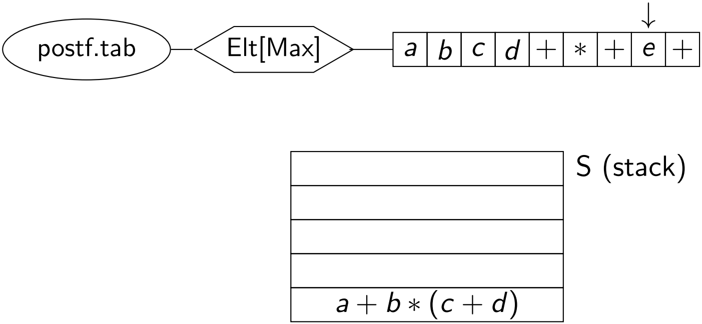
Example: $a\ b\ c\ d\ \ + * + e +$



a	b	c	d	$+$	$*$	$+$	e	$+$

Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



postf.tab

Elt[Max]

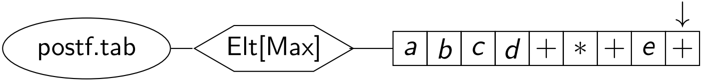
a	b	c	d	$+$	$*$	$+$	e	$+$

S (stack)

$a + b * (c + d)$

Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



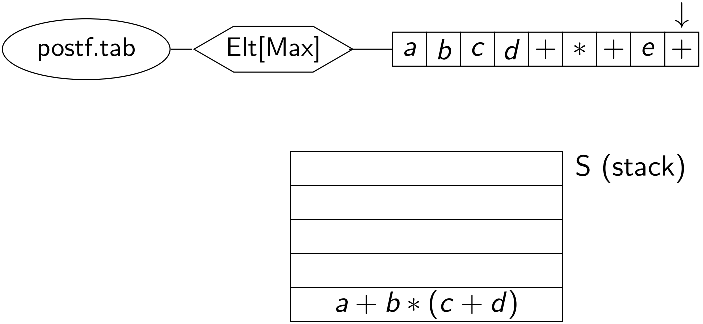
a	b	c	d	$+$	$*$	$+$	e	$+$

S (stack)

e
$a + b * (c + d)$

Postfix expression evaluation

Example: $a\ b\ c\ d\ + * + e +$



postf.tab

Elt[Max]

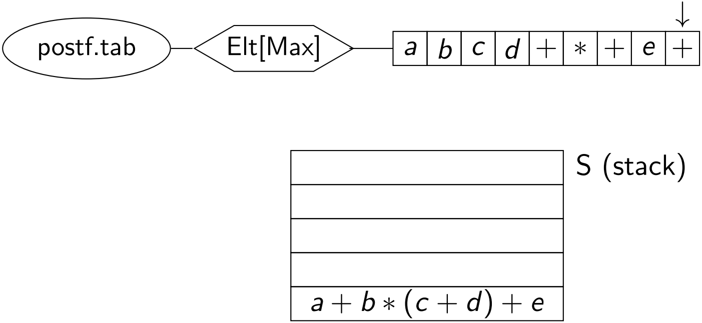
a	b	c	d	$+$	$*$	$+$	e	$+$

S (stack)

$a + b * (c + d)$

Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



postf.tab

Elt[Max]

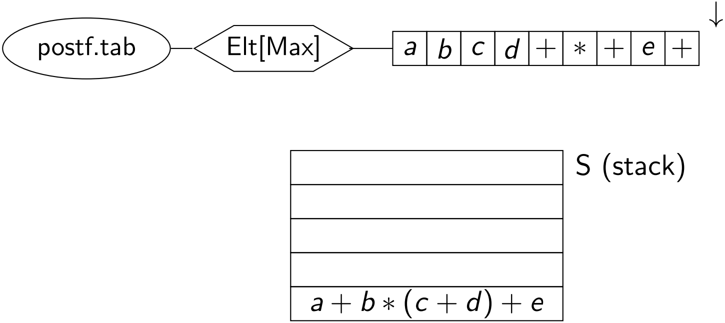
a	b	c	d	$+$	$*$	$+$	e	$\overset{\downarrow}{+}$

S (stack)

$a + b * (c + d) + e$

Postfix expression evaluation

Example: $a\ b\ c\ d\ \ + * + e +$



a	b	c	d	$+$	$*$	$+$	e	$+$

$a + b * (c + d) + e$

Postfix expression evaluation

```
function valPostfix(postfix)
begin
    $S \leftarrow emptyStack()$
    while (not $isEmpty(postfix)$) do
        $x \leftarrow read(postfix)$;    $delete(infix)$
        if ($operand(x)$) then
            $push(S, x)$
        else
            $right \leftarrow top(S)$;    $pop(S)$
            $left \leftarrow top(S)$;    $pop(S)$
            $val \leftarrow left\ op(x)\ right$
            $push(S, val)$
    $val = top(S)$;    $pop(S)$
    return $val$
end
```