elementary data structures

overview:

- stacks
- queues
- linked lists
- trees

Stacks

with elements in M

configuration: $s \in M^*$

operations:

- push(x,s)
  - $s' = x \circ s$
- pop(s)
  - $s' = tail(s)$
  - output $hd(s)$
- isempty(s)

Stacks

with elements in M

configuration: $s \in M^*$

operations:

- push(x,s)
  - $s' = x \circ s$
- pop(s)
  - $s' = tail(s)$
  - output $hd(s)$
- isempty(s)

implementation

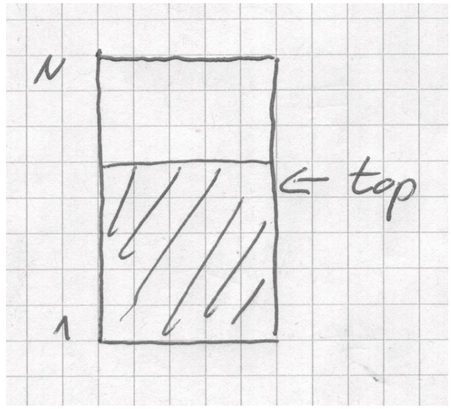
- array S[1:N]
- variable top

Algorithm 16 IsEmpty

Input: stack S

Output: 1 if S is empty, 0 otherwise

```
1: if top = 0 then
2:     return 1
3: else
4:     return 0
```



Stacks

with elements in M

configuration: $s \in M^*$

operations:

- push(x,s)
  - $s' = x \circ s$
- pop(s)
  - $s' = tail(s)$
  - output $hd(s)$
- isempty(s)

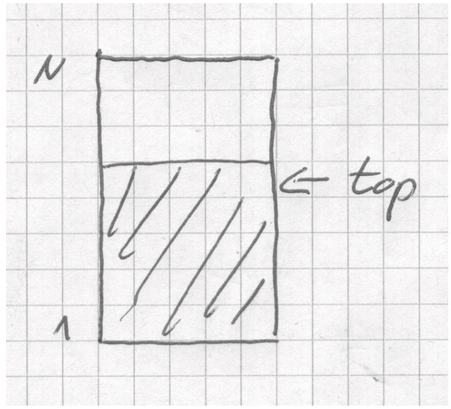
implementation

- array S[1:N]
- variable top

Algorithm 16 IsEmpty

Input: stack S
Output: 1 if S is empty, 0 otherwise

```
1: if top = 0 then
2:     return 1
3: else
4:     return 0
```



Algorithm 17 Push

Input: stack S, element x
Output: adds x to S

```
1: if top ≥ N then
2:     "error"
3: else
4:     top = top + 1;
5:     S[top] = x;
```

Algorithm 18 Pop

Input: stack S
Output: returns and removes the top element of S

```
1: if IsEmpty(S) then
2:     "error"
3: else
4:     top := top − 1
5:     return S[top+1]
```

queues

with elements in M

configuration: $s \in M^*$

operations:

- enqueue(x,s)
  - $s' = s \circ x$
- dequeue(s)
  - $s' = tail(s)$
  - output $hd(s)$
- isempty(s)

queues

with elements in M

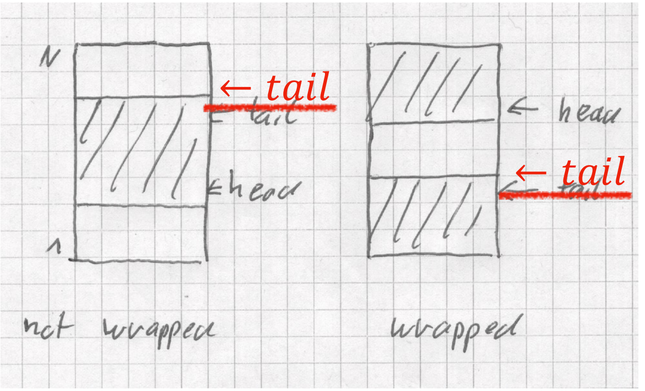
configuration: $s \in M^*$

operations:

- enqueue(x,s)
  - $s' = s \circ x$
- dequeue(s)
  - $s' = tail(s)$
  - output $hd(s)$
- isempty(s)

implementation

- array Q[1:N]
- variables head, tail



Algorithm 19 IsEmpty

Input: queue Q
Output: 1 if Q is empty, 0 otherwise

```
1: if head = tail then
2:     return 1
3: else
4:     return 0
```

Algorithm 20 Enqueue

Input: queue Q, element x
Output: adds x to Q

```
1: if head = tail + 1 or head = 1 and tail = N then
2:     "error"
3: else
4:     Q[tail] = x
5:     if tail = N then
6:         tail = 1
7:     else
8:         tail = tail + 1
```

Algorithm 21 Dequeue

Input: queue Q
Output: the first element of Q

```
1: if head = tail then
2:     "error"
3: else
4:     x := Q[head]
5:     if head = N then
6:         head := 1
7:     else
8:         head := head + 1
9:     return x
```

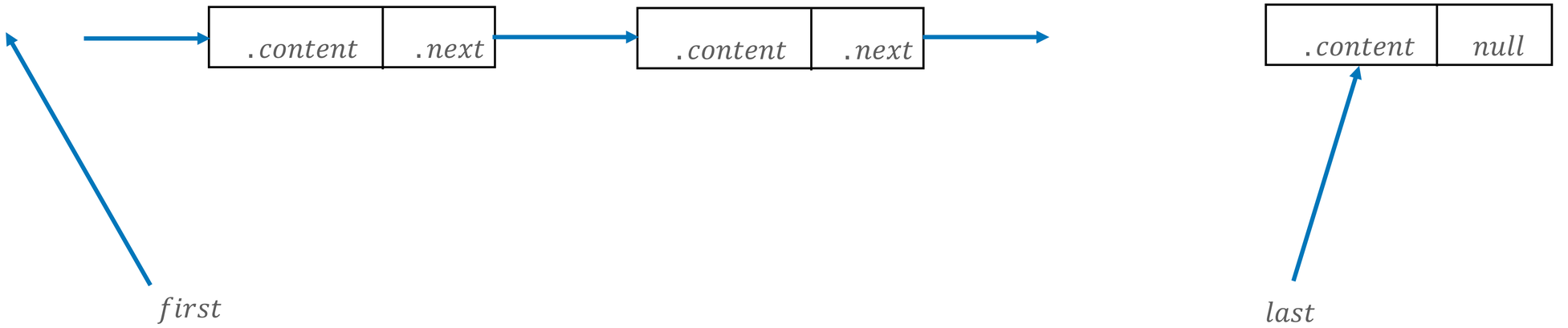
linked lists

creating a linked list

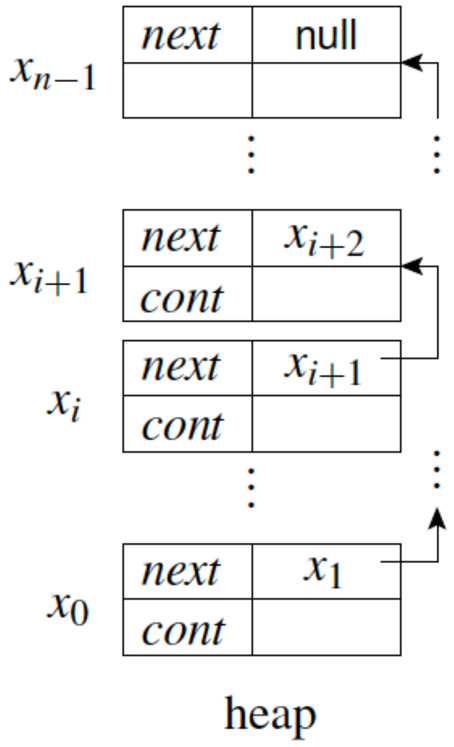
from I2OS set of slides 14

types

```
typedef LEL* u;
typedef struct {int content; u next} LEL;
```



creating a linked list of length n on the heap



$$llist(x[0:n-1], c)$$

```
typedef LEL* u;
typedef struct {int content; u next} LEL;
u first;
u last;
int n;
int main()
{
  n = N;
  first = new LEL*;
  last = first;
  n = n - 1;
  while n>0
  {
    last*.next = new LEL*;
    last = last*.next;
    n = n - 1
  }
```

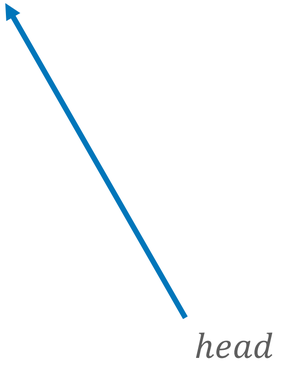
creating a linked list

from I2OS set of slides 14

doubly linked lists

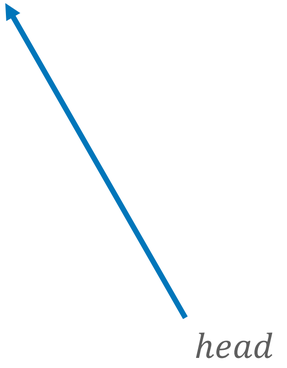
types

```
typedef LEL* u;
typedef struct {int content; u next} LEL;
                                    ;u prev
```



creating a linked list

from I2OS set of slides 14



doubly linked lists

types

```
typedef LEL* u;
typedef struct {int content; u next} LEL;
```

;u prev

Algorithm 23 List-insert

Input: a list L, an element x

Output: appends x to the front of the list

1: Next(x) := *head*
2: **if** *head* $\neq$ NULL **then**
3: Prev(*head*) := x
4: *head* := x
5: Prev(x) := NULL

$next(x) \equiv x.next*$

Algorithm 24 List-delete

Input: a list L, an element x

Output: removes x from the list

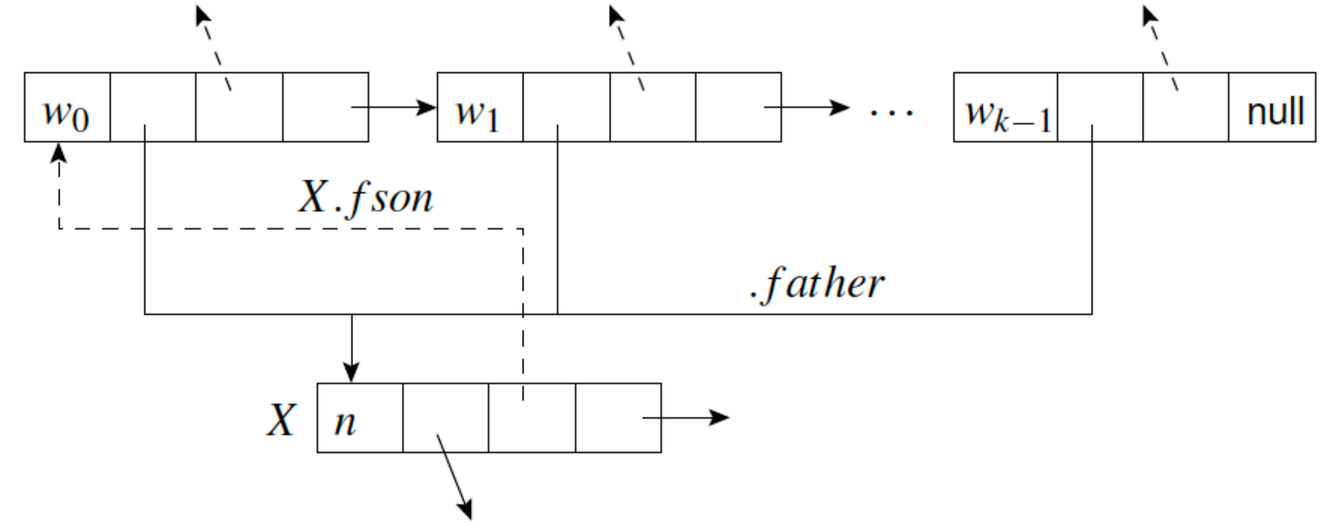
1: **if** Prev(x) $\neq$ NULL **then**
2: Next(Prev(x)) := Next(x)
3: **else**
4: *head* := Next(x)
5: **if** Next(x) $\neq$ NULL **then**
6: Prev(Next(x)) := Prev(x)

$prev(y) \equiv y.prev*$

trees

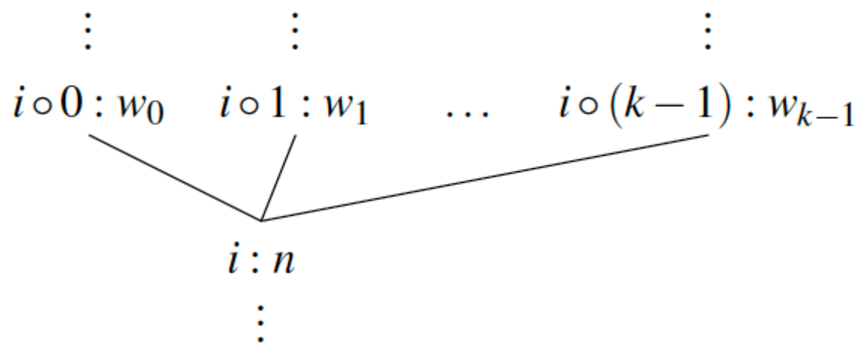
remember implementation of derivation trees

Implementing derivation trees in C0



*C*0

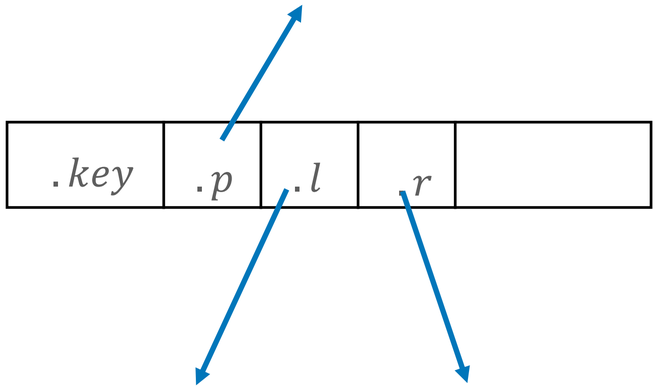
derivation tree




```
typedef DTE* DTEp;
typedef struct {
  uint label;
  DTEp father;
  DTEp fson;
  DTEp bro
} DTE;
```

from I2OS slide set 6

Implementing binary trees in C0

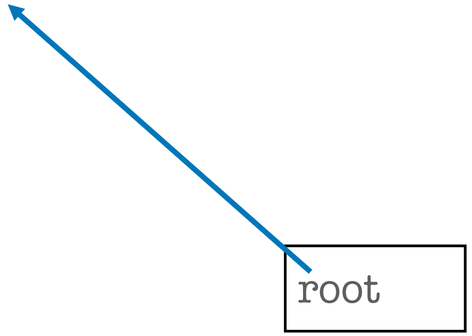



```
 typedef TE* TEp
typedef struct {
 int key;
 TEp p; /*parent*/
 TEp l; /*lson*/
 TEp r; /*rson */
  ....
} TE
```

tree element

```
TEp root
```

root pointer



time for operations

Stacks

IsEmpty: O(1) time, checks whether the stack is empty

Push: O(1) time, adds an element to the stack

Pop: O(1) time, removes an element to the stack

Stacks use the *last-in-first-out* principle.

time for operations

Stacks

IsEmpty: O(1) time, checks whether the stack is empty

Push: O(1) time, adds an element to the stack

Pop: O(1) time, removes an element to the stack

Stacks use the *last-in-first-out* principle.

Queues

IsEmpty: O(1), checks whether queue is empty

Enqueue: O(1), adds an element to the queue

Dequeue: O(1), removes an element from the queue

Queues use the *first-in-first-out* principle.

time for operations

Stacks

IsEmpty: $O(1)$ time, checks whether the stack is empty

Push: $O(1)$ time, adds an element to the stack

Pop: $O(1)$ time, removes an element to the stack

Stacks use the *last-in-first-out* principle.

Doubly linked lists

List-search: $O(n)$, finds an element with a given key

List-insert: $O(1)$, adds an element to the front of the list

List-delete: $O(1)$, removes an element given by a reference

Queues

IsEmpty: $O(1)$, checks whether queue is empty

Enqueue: $O(1)$, adds an element to the queue

Dequeue: $O(1)$, removes an element from the queue

Queues use the *first-in-first-out* principle.