SEQUENCES, ARRAYS, AND LINKED LISTS

(with remarks on recursion and values vs. references)

by

David Toman

University of Waterloo

List of Slides

List and Stack ADTs

- Ordered sequences of elements
- Operations:

	List	Stack
initialize	make-list	make-stack
test for emptiness	is-empty(l)	is-empty(s)
lookup 1st element	get(l,1)	top(s)
add 1st element e	insert(l,e,1)	push(s,e)
remove 1st element	delete(l,1)	pop(s)

$\Rightarrow$ a *list ADT* often allows

looking up, adding, and removing i-th element
adding elements “at the end”, etc.

$\Rightarrow$ Stack is just a special case of a list.

Linked List Data Structure

- an *inductive* data type

$$\texttt{list}(\tau) = [\,] + (\tau \times \texttt{ref to list}(\tau))$$

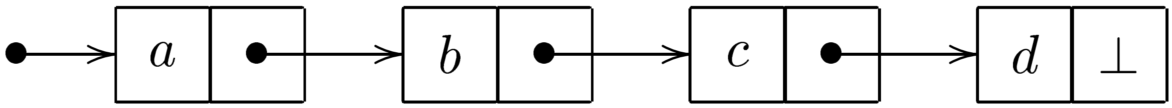
$\Rightarrow$ instances of lists often written as $[a, b, c, d]$

- a corresponding record type:

```
list = record
          value: element;
          next: ^list;
       end;
```

$\Rightarrow$ assuming $[\,]$ is represented by **`nil`**

- pictorially:



An Array Version

- an *inductive* data type

$$\mathtt{list}(\tau) = [\,] + (\tau \times \mathtt{list}(\tau))$$

$\Rightarrow$ that's just an array of τs!

a	b	c	d	$\perp$

how many elements does it contain?

special "end of list" marker
additional "count" field

Examples:

$\Rightarrow$ strings (in C)
$\Rightarrow$ hardware stacks
stored backwards with "extra space" for growth

List of What?

What can we store in lists/stacks?

- homogeneous lists: single type of elements

  ⇒ references to elements
  ⇒ in-lined elements
  common:

```
t = record
      <many fields here>;
      next:^t;
    end;
```

- heterogeneous lists

  ⇒ with variant records

  ⇒ with casts (wrong!)

Implementation of Operations

IDEA: induction on list structure

Example: get n-th element:

- using the list ADT operations
- from linked list:

```
function getn(l:list,n:integer):element;
  if (l=nil) then ``no n-th element'';
  else
    if (n=0) then getn:=l.value
             else getn:=getn(l.next,n-1);
```

$\Rightarrow O(n)$ (which is contained in $O(|l|)$)

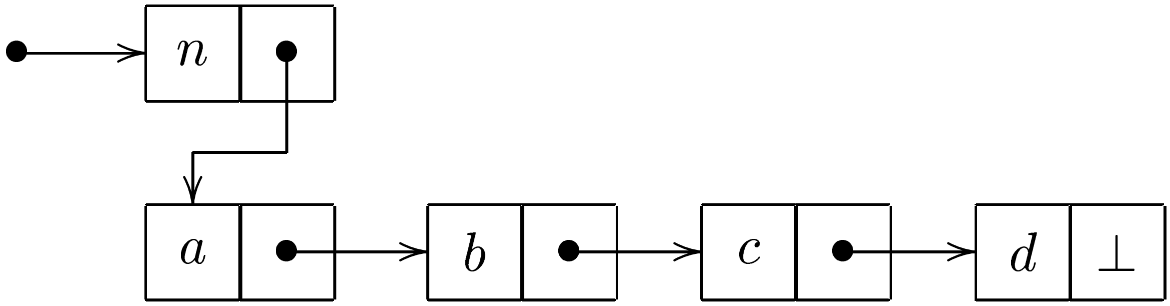
- from array:

```
function getn(l:list,n:integer):element;
  if (l.size>n) then getn := l.values[n];
                else ``no n-th element'';
```

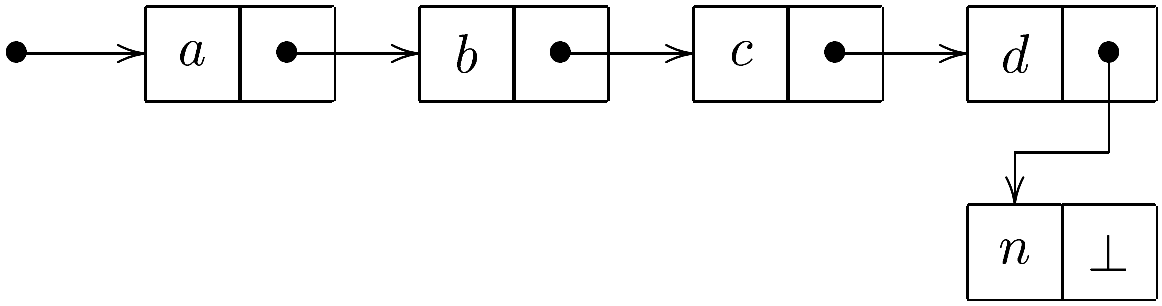
$\Rightarrow O(1)$

Adding Elements to a Linked List

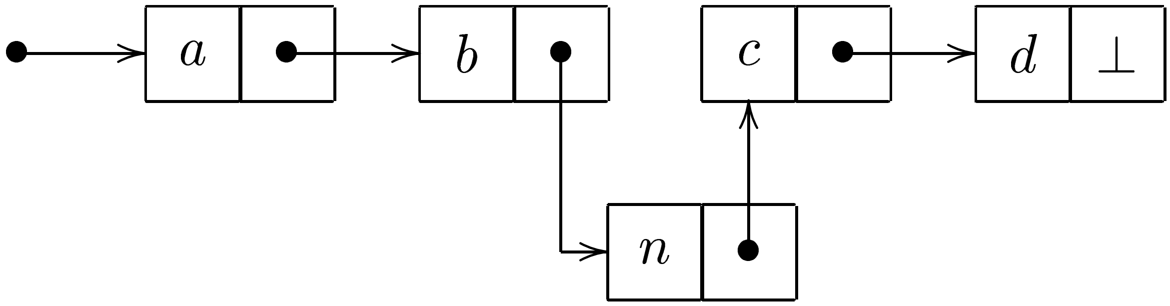
- to the beginning:



- to the end:



- in the middle of the list:



Linked List in an Array

IDEA: remember that the "memory" is just a big array
$\Rightarrow$ we can use an array explicitly

	0	1	2	3	4	5	6	$\cdots$
value:	a		b	d	c			$\cdots$
next:	2		4	-1	3			$\cdots$

$\Rightarrow$ we need to remember where it starts (0)
$\Rightarrow$ insertions/deletions:

  what about the empty spaces?

  $\Rightarrow$ memory management (ala **`new`**/**`dispose`**)

Queue ADT

An ADT that supports the following operations:

- new-queue : e

  $\Rightarrow$ creates a new empty queue

- enqueue(q, e)

  $\Rightarrow$ adds an element e on the end of q

- dequeue(q) : e

  $\Rightarrow$ removes the first element from q and returns it

Implementations

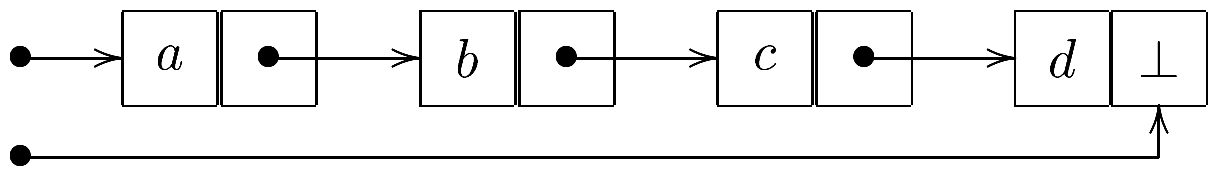
$\Rightarrow$ based on an array (circular buffer)

$\Rightarrow$ based on a list

here we have a problem: "enqueue" $\in \Theta(n)$

Variations on Linked Lists

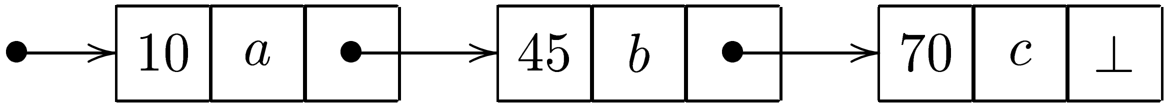
- **GOAL:** improve "appending to the end"
  $\Rightarrow$ trade a little bit of space for lots of time
- **SOLUTION:** additional $O(1)$ space for a reference to the end of the list:



$\Rightarrow$ enqueue drops from $\Theta(n)$ to $O(1)$ time

Sparse Arrays

- how to store array (or a matrix) contains mostly 0's?



$\Rightarrow$ return 0 for “skipped” elements.

- cost of accessing element i?

$\Rightarrow$ linear in the size of the array

we don't know how many elements are before us

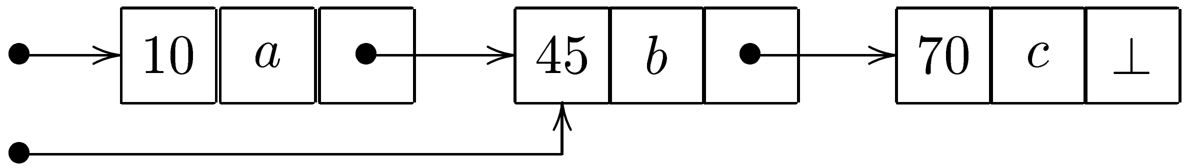
Variations on Linked Lists II

- whet about computing $\sum_{i=1}^{n} a_i$?

```
sum := 0;
for i:=1 to n do
    sum := sum+getn(a,i);
```

$\Rightarrow O(n^2)$ — BAD!

- add extra $O(1)$ space for a reference to the "last accessed" item:

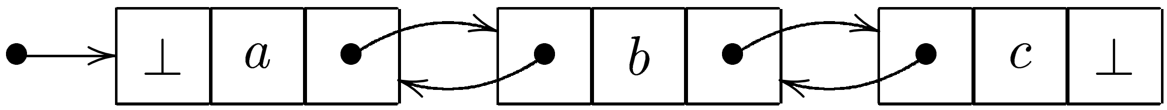


$\Rightarrow$ and modify **getn**(l, i) as follows
start from the last accessed element if it is $\leq i$
start from the the beginning otherwise

$\Rightarrow$ improves the time complexity of the loop to $O(n)$

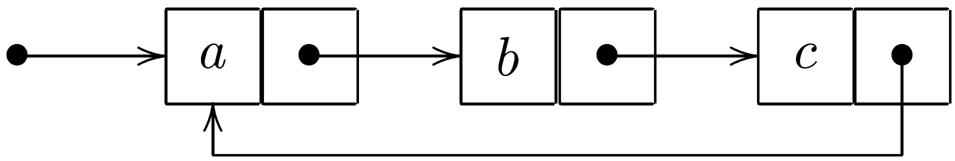
X-linked List

- doubly-linked list



$\Rightarrow$ with $O(1)$ space we can have queues with enqueue/dequeue *at both ends* is in $O(1)$ time

- circular list



- circular doubly-linked list
- etc.

Converting Recursion to Iteration

Standard pattern of algorithms for lists:

$\Rightarrow$ recursion on the structure of the list

$$F(l) = \begin{cases} l = [\,] & \mapsto \; B \\ l = [h|t] & \mapsto \; I(h, F(t)) \end{cases}$$

where B is what to do in the base case
I is what to do in the recursive case

Can we convert recursion into a **while** loop(s)?

$\Rightarrow$ use the stack ADT!
$\Rightarrow$ TRO optimization

Values vs. References

Consider the following code:

```
p:= newstack; push(p,1); push(p,2);

p1:=p; push(p1,3); push(p1,4);
p2:=p; push(p2,3); push(p2,4);
```

Question is **p1** the same as **p2**?

⇒ NO!

⇒ why? **getn(p1,i)**=**getn(p2,i)**!

⇒ but changing, e.g., the 2nd element in **p1** will make them different!

(partial) Solution: header node

⇒ makes sure everyone is “sharing” the same stack/list

Summary

- (X-)Linked Lists are probably **the most common** dynamic data structure in existence
  - $\Rightarrow$ while quite simple, you wouldn't believe how many errors you find in processing simplest lists
  - $\Rightarrow$ even though there is a "canonical" way to do it!
- Lists/Arrays are commonly used to build other ADTs
  - $\Rightarrow$ Stacks
  - $\Rightarrow$ Queues
  - $\Rightarrow$ Sparse Arrays
  - $\Rightarrow$ ...
- Often we can trade a bit of space for time
  - $\Rightarrow$ depends on (the most frequent) "access pattern"(s)