LISTS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Agenda

1 Abstract data types and data structures

2 List ADT

3 List ADT as an array

4 List ADT as a linked list

5 Asymptotic efficiency of lists

6 Bibliography

Centro de Informática UFPE

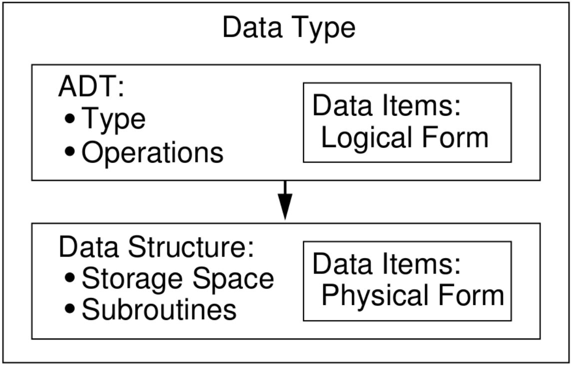
Universidade Federal de Pernambuco

Abstract data types and data structures

Important concepts

- Type: a collection of values
- Simple vs. composite type
- Data type: a type and operations to manipulate the type
- Abstract data type (ADT): a data type as a software component
- Data structure: implementation for an ADT
- Multiple implementations for the same ADT

Abstract data types and data structures[1]



[1] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Agenda

1 Abstract data types and data structures

2 List ADT

3 List ADT as an array

4 List ADT as a linked list

5 Asymptotic efficiency of lists

6 Bibliography

List ADT

Mathematical abstraction

- Sequences of homogenous elements
- A symbol $|$ to denote the current position of the cursor

Example

- Let $l = \langle 20, 23 \mid 12, 15 \rangle$
- After inserting 10 in l, we have $l' = \langle 20, 23 \mid 10, 12, 15 \rangle$
- After removing an element from l, we have $l'' = \langle 20, 23 \mid 12, 15 \rangle$

List ADT

Operations (let *E* be an arbitrary data type):

- void clear(List l);
- void insert(List l, E item);
- void append(List l, E item);
- E remove(List l);
- void moveToStart(List l);
- void moveToEnd(List l);
- void prev(List l);
- void next(List l);
- int length(List l);
- int currPos(List l);
- void moveToPos(List l, int pos);
- E getValue(List l);

Centro de Informática UFPE

Universidade Federal de Pernambuco

Declaring an ADT in C

Type and operations as prototypes in a .h file

- A function prototype to create an instance of the ADT
- Provide to the functions the values to be manipulated (`List* l`)

```c
#ifndef _LIST_H
#define _LIST_H

typedef struct list List;
List* create_list(int size); // Creates a new list
void clear(List* l);
void insert(List* l, E item);

// ...
#endif
```

Declaring an ADT in Java

Typically, an interface, in addition to operations as abstract methods

- No need of a function prototype to create an instance of the ADT
- No need to provide to the methods the values to be manipulated

```java
public interface List {

    public void clear();
    public void insert(E item);
    // ...

}
```

Operations implemented in the logical level

We might have operations implemented in the logical level

- Same implementation regardless of data structure

Algorithm: boolean find(List l, int k)

```
moveToStart(l);
while currPos(l) < length(l) do
    if k = getValue(l) then
        return true;
    next(l);
return false;
```

Operations implemented in the logical level

Concrete functions defined in the *.h file

```
#ifndef _LIST_H
#define _LIST_H
// ...

bool find(List* l, int k) {
    moveToStart(l);
    while(currPos(l) < length(l)) {
        if (k == getValue(l)) { return true; }
        next(l);
    }
    return false;
}
#endif
```

Operations implemented in the logical level

Concrete methods defined in an abstract class

```java
public abstract class List {
    // ...

    public boolean find(int k) {
        this.moveToStart();
        while (this.currPos() < this.length()) {
            if (k == this.getValue()) { return true; }
            this.next();
        }
        return false;
    }
}
```

Agenda

1 Abstract data types and data structures

2 List ADT

3 List ADT as an array

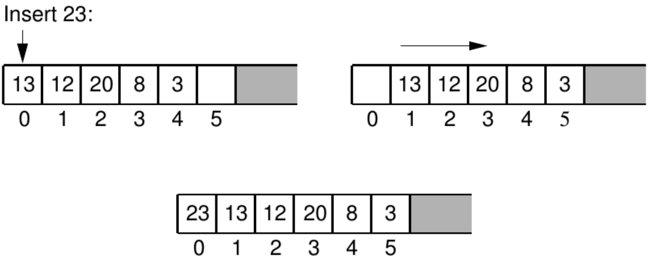
4 List ADT as a linked list

5 Asymptotic efficiency of lists

6 Bibliography

List ADT implemented as an array[2]

Insertion



Deletion

- Analogous implementation

[2] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

List ADT implemented as an array

Composite type:

```
int maxSize;                         // capacity
int listSize;                        // number of elements
int curr;                            // position of the cursor
E[] listArray;                       // array with values
```

Algorithm: List create_list(int size)

$l.maxSize \leftarrow size$;
$l.listSize \leftarrow l.curr \leftarrow 0$;
$l.listArray \leftarrow new\ E[size]$;
return l;

List ADT implemented as an array (C)

Data structure: defined in a .c file

```c
#include "list.h"
typedef struct list {
    int maxSize; int listSize; int curr; E* listArray;
} List;

List* create_list(int size) {
    List* l = (List*) malloc(sizeof(List));
    l->maxSize = size;
    l->listSize = l->curr = 0;
    l->listArray = (E*) malloc(size * sizeof(E));
    return l;
}
// ...
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

List ADT implemented as an array (Java)

Data structure: defined in a concrete class

```
public class ArrayList implements List { // List: an interface

    private int maxSize; private int listSize;
    private int curr; private E[] listArray;

    // The constructor has the role of ``create_list''
    public ArrayList(int size) {
        this.maxSize = size;
        this.listSize = this.curr = 0;
        this.listArray = new E[size];
    }
    //...
}
```

List ADT implemented as an array: <|> ⇝ <|7>

Algorithm: void insert(List l, E it)

if *l.listSize* $\geq$ *l.maxSize* **then** *error*;
i $\leftarrow$ *l.listSize*;
while $i >$ *l.curr* **do**
    l.listArray$[i] \leftarrow$ *l.listArray*$[i-1]$; // shift right
    i--;
l.listArray[*l.curr*] $\leftarrow$ *it*;
l.listSize++;

`insert(1,7)`, just after `create_list(5)` (`c` denotes `curr`)

index:	0	1	2	3	4
value:	-	-	-	-	-
	c				

Centro de Informática UFPE

Universidade Federal de Pernambuco

List ADT implemented as an array: <|> $\rightsquigarrow$ <|7>

Algorithm: void insert(List l, E it)

if $l.listSize \geq l.maxSize$ **then** $error$;
$i \leftarrow l.listSize$;
while $i > l.curr$ **do**
    $l.listArray[i] \leftarrow l.listArray[i-1]$; `// shift right`
    i--;
$l.listArray[l.curr] \leftarrow it$;
$l.listSize$++;

`insert(l,7)`, just after `create_list(5)` (`c` denotes `curr`)

index:	0	1	2	3	4
value:	-	-	-	-	-
	c				
	i				

Centro de Informática UFPE
Universidade Federal de Pernambuco

List ADT implemented as an array: <|> ⇝ <|7>

Algorithm: void insert(List l, E it)

if *l.listSize* $\geq$ *l.maxSize* **then** *error*;
$i \leftarrow l.listSize$;
while $i > l.curr$ **do**
    $l.listArray[i] \leftarrow l.listArray[i-1]$;   `// shift right`
    i--;
$l.listArray[l.curr] \leftarrow it$;
l.listSize++;

`insert(l,7)`, just after `create_list(5)` (`c` denotes `curr`)

index:	0	1	2	3	4
value:	7	-	-	-	-
	c				
	i				

Centro de Informática UFPE
Universidade Federal de Pernambuco

List ADT implemented as an array: <|7> $\rightsquigarrow$ <|5,7>

Algorithm: void insert(List l, E it)

if $l.listSize \geq l.maxSize$ **then** $error$;
$i \leftarrow l.listSize$;
while $i > l.curr$ **do**
  $l.listArray[i] \leftarrow l.listArray[i-1]$; `// shift right`
  i--;
$l.listArray[l.curr] \leftarrow it$;
$l.listSize$++;

`insert(l,5)`

index:	0	1	2	3	4
value:	7	-	-	-	-
	c				
		i			

Centro de Informática UFPE
Universidade Federal de Pernambuco

List ADT implemented as an array: $<|7> \rightsquigarrow <|5,7>$

Algorithm: void insert(List l, E it)

> **if** $l.listSize \geq l.maxSize$ **then** *error*;
> $i \leftarrow l.listSize$;
> **while** $i > l.curr$ **do**
> $l.listArray[i] \leftarrow l.listArray[i-1]$; `// shift right`
> i--;
> $l.listArray[l.curr] \leftarrow it$;
> $l.listSize$++;

`insert(l,5)`

index:	0	1	2	3	4
value:	7	7	-	-	-
	c				
		i			

Centro de Informática UFPE

Universidade Federal de Pernambuco

List ADT implemented as an array: <|7> $\rightsquigarrow$ <|5,7>

Algorithm: void insert(List l, E it)

if $l.listSize \geq l.maxSize$ **then** *error*;
$i \leftarrow l.listSize$;
while $i > l.curr$ **do**
    $l.listArray[i] \leftarrow l.listArray[i-1]$; `// shift right`
    i--;
$l.listArray[l.curr] \leftarrow it$;
$l.listSize$++;

`insert(l,5)`

index:	0	1	2	3	4
value:	7	7	-	-	-
	c				
	i				

Centro de Informática UFPE

Universidade Federal de Pernambuco

List ADT implemented as an array: <|7> ⇝ <|5,7>

Algorithm: void insert(List l, E it)

```
if l.listSize ≥ l.maxSize then error;
i ← l.listSize;
while i > l.curr do
    l.listArray[i] ← l.listArray[i − 1];        // shift right
    i--;
l.listArray[l.curr] ← it;
l.listSize++;
```

insert(l,5)

index:	0	1	2	3	4
value:	5	7	-	-	-
	c				
	i				

Centro de Informática UFPE

Universidade Federal de Pernambuco

List ADT implemented as an array

Algorithm: void moveToStart(List l)

$l.curr \leftarrow 0$;

Algorithm: void moveToEnd(List l)

$l.curr \leftarrow l.listSize$;

Algorithm: void prev(List l)

if $l.curr \neq 0$ **then** $l.curr$--;

Algorithm: void next(List l)

if $l.curr < l.listSize$ **then** $curr$++;

Centro de Informática UFPE

Universidade Federal de Pernambuco

List ADT implemented as an array: <2|5,7> $\rightsquigarrow$ <2|7>

Algorithm: E remove(List l)

```
if l.curr < 0 ∨ curr ≥ listSize then return NULL;
E it ← l.listArray[l.curr]; i ← l.curr;
while i < l.listSize − 1 do
    l.listArray[i] ← l.listArray[i + 1];        // shift left
    i++;
l.listSize--;
return it;
```

`remove(l)`, just after `insert(l,2) ; next(l)`

index:	0	1	2	3	4
value:	2	5	7	-	-
		c			

List ADT implemented as an array: <2|5,7> $\rightsquigarrow$ <2|7>

Algorithm: E remove(List l)

if $l.curr < 0 \vee curr \geq listSize$ **then return** $NULL$;
E $it \leftarrow l.listArray[l.curr]$; $i \leftarrow l.curr$;
while $i < l.listSize - 1$ **do**
    $l.listArray[i] \leftarrow l.listArray[i+1]$;   `// shift left`
    i++;
$l.listSize$--;
return it;

`remove(l)`, just after `insert(l,2) ; next(l)`

index:	0	1	2	3	4
value:	2	5	7	-	-
		c			
		i			

List ADT implemented as an array: $<2|5,7> \rightsquigarrow <2|7>$

Algorithm: E remove(List l)

---

if $l.curr < 0 \vee curr \geq listSize$ **then return** $NULL$;
E $it \leftarrow l.listArray[l.curr]$; $i \leftarrow l.curr$;
while $i < l.listSize - 1$ **do**
    $l.listArray[i] \leftarrow l.listArray[i+1]$;   `// shift left`
    i++;
$l.listSize$--;
return it;

`remove(l)`, just after `insert(l,2) ; next(l)`

index:	0	1	2	3	4
value:	2	7	7	-	-
		c			
		i			

Centro de Informática UFPE
Universidade Federal de Pernambuco

List ADT implemented as an array: <2|5,7> $\rightsquigarrow$ <2|7>

Algorithm: E remove(List l)

```
if l.curr < 0 ∨ curr ≥ listSize then return NULL;
E it ← l.listArray[l.curr]; i ← l.curr;
while i < l.listSize − 1 do
    l.listArray[i] ← l.listArray[i + 1];        // shift left
    i++;
l.listSize--;
return it;
```

`remove(l)`, just after `insert(l,2) ; next(l)`

index:	0	1	2	3	4
value:	2	7	7	-	-
		c			
			i		

Centro de Informática UFPE

Universidade Federal de Pernambuco

List ADT implemented as an array: <2|5,7> ⇝ <2|7>

Algorithm: E remove(List l)

if $l.curr < 0 \lor curr \geq listSize$ **then return** $NULL$;
E $it \leftarrow l.listArray[l.curr]$; $i \leftarrow l.curr$;
while $i < l.listSize - 1$ **do**
    $l.listArray[i] \leftarrow l.listArray[i+1]$; `// shift left`
    i++;
$l.listSize$--;
return it;

`remove(l)`, just after `insert(l,2) ; next(l)`

index:	0	1	2	3	4
value:	2	7	7	-	-
		c			
			i		

Centro de Informática UFPE
Universidade Federal de Pernambuco

Agenda

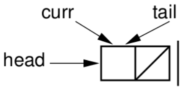
1. Abstract data types and data structures
2. List ADT
3. List ADT as an array
4. List ADT as a linked list
5. Asymptotic efficiency of lists
6. Bibliography

Centro de Informática UFPE

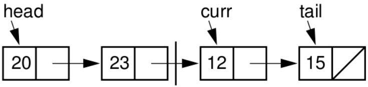
Universidade Federal de Pernambuco

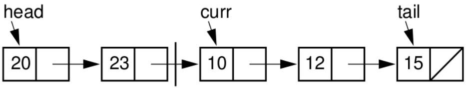
List ADT implemented as a linked list[3]

Empty list: header node *vs.* space cases (`insert` and `remove`)



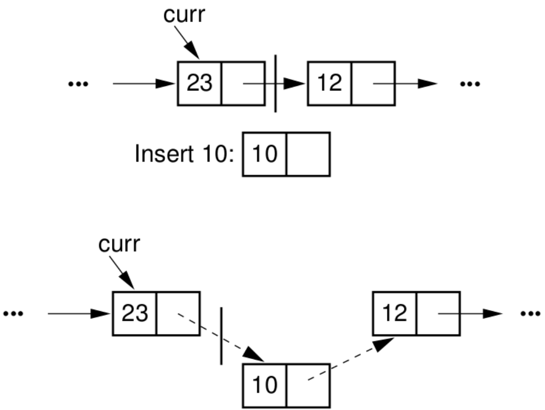
Attention to where `curr` is pointing to (not as follows)





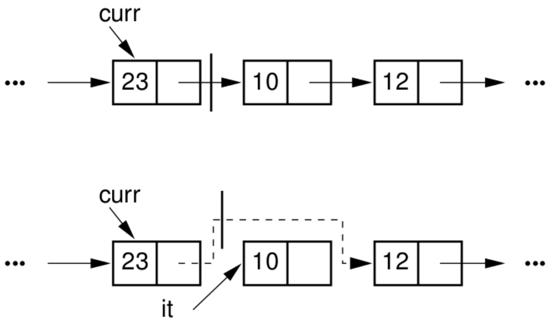
[3] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

List ADT implemented as a linked list: insertion[4]



[4] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

List ADT implemented as a linked list: deletion[5]



Important: memory deallocation

- With no garbage collection: user's duty
- With garbage collection: GC's duty

[5] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

List ADT implemented as a linked list

Composite type (Link):

```
E element;       // value stored in this link/node
Link next;       // reference to the next link/node
```

Algorithm: Link create_link(E it, Link nextval)

$n.element \leftarrow it$; $n.next \leftarrow nextval$; **return** n;

Algorithm: Link create_link(Link nextval)

$n.next \leftarrow nextval$; **return** n;

List ADT implemented as a linked list

Composite type (List):

```
Link head;
Link tail;
Link curr;
int cnt;                                   // list size
```

Algorithm: List create_list()

```
l.curr ← l.tail ← l.head ← create_link(NULL);    // header node
l.cnt ← 0;
return l;
```

Centro de Informática UFPE

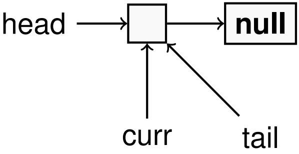
Universidade Federal de Pernambuco

List ADT implemented as a linked list: $<|> \rightsquigarrow <|7>$

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next)$;
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next$;
$l.cnt{+}{+}$;

`insert(l,7)`, just after `create_list()` (empty node: header)

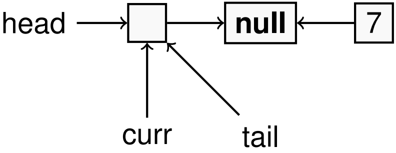


List ADT implemented as a linked list: <|> ⇝ <|7>

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next);$
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next;$
$l.cnt{+}{+};$

`insert(l,7)`, just after `create_list()` (empty node: header)

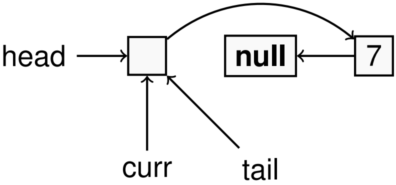


List ADT implemented as a linked list: <|> ⇝ <|7>

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next)$;
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next$;
$l.cnt{+}{+}$;

`insert(l,7)`, just after `create_list()` (empty node: header)



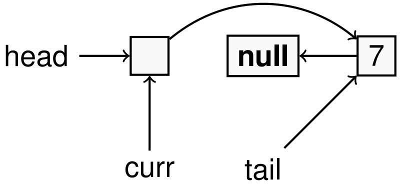
Centro de Informática UFPE
Universidade Federal de Pernambuco

List ADT implemented as a linked list: <|> $\rightsquigarrow$ <|7>

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next)$;
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next$;
$l.cnt$++;

`insert(l,7)`, just after `create_list()` (empty node: header)

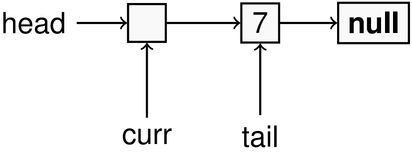


List ADT implemented as a linked list: $<|> \rightsquigarrow <|7>$

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next)$;
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next$;
$l.cnt$++;

Better presented as follows

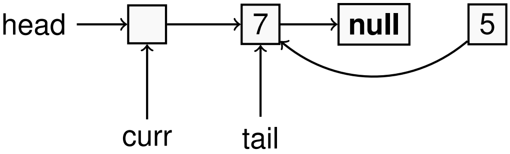


List ADT implemented as a linked list: <|7> $\rightsquigarrow$ <|5,7>

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next);$
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next;$
$l.cnt{+}{+};$

`insert(l,5)`

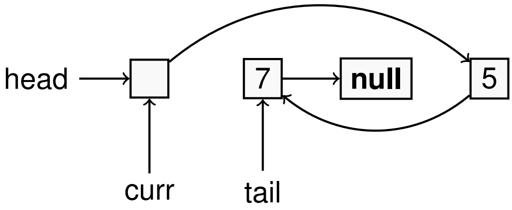


List ADT implemented as a linked list: <|7> $\rightsquigarrow$ <|5,7>

Algorithm: void insert(List l, E it)

1. $l.curr.next \leftarrow create\_link(it, l.curr.next)$;
2. **if** $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next$;
3. $l.cnt{+}{+}$;

`insert(l,5)`

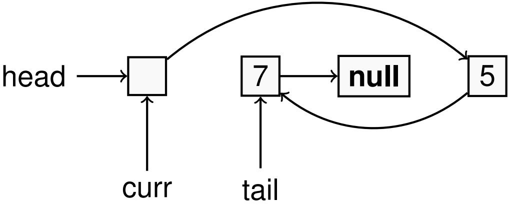


List ADT implemented as a linked list: <|7> $\rightsquigarrow$ <|5,7>

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next)$;
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next$;
$l.cnt{+}{+}$;

```
insert(l,5)
```

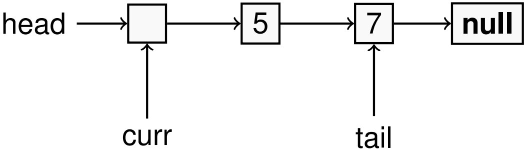


List ADT implemented as a linked list: <|7> ⇝ <|5,7>

Algorithm: void insert(List l, E it)

$l.curr.next \leftarrow create\_link(it, l.curr.next)$;
if $l.tail = l.curr$ **then** $l.tail \leftarrow l.curr.next$;
$l.cnt{+}{+}$;

Better presented as follows



List ADT implemented as a linked list

Algorithm: void moveToStart(List l)

$l.curr \leftarrow l.head$;

Algorithm: void prev(List l)

if $l.curr = l.head$ **then return**;
Link $temp \leftarrow l.head$;
while $temp.next \neq l.curr$ **do** $temp \leftarrow temp.next$;
$l.curr \leftarrow temp$;

Algorithm: void next(List l)

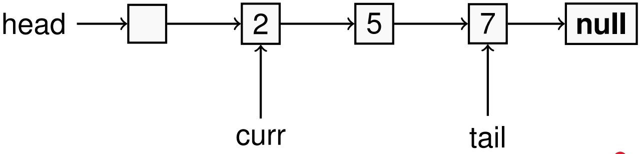
if $l.curr \neq l.tail$ **then** $l.curr \leftarrow l.curr.next$;

List ADT impl. as a linked list: $\langle 2|5,7\rangle \rightsquigarrow \langle 2|7\rangle$

Algorithm: E remove(List l)

if $l.curr.next = NULL$ **then return** $NULL$;
E $it \leftarrow l.curr.next.element$;
if $l.tail = l.curr.next$ **then** $l.tail \leftarrow l.curr$;
$l.curr.next \leftarrow l.curr.next.next$; $l.cnt$--;
return it;

`remove(l)`, just after `insert(l,2) ; next(l)`

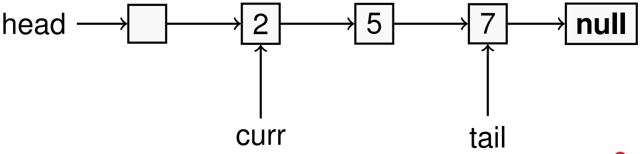


List ADT impl. as a linked list: <2|5,7> ⇝ <2|7>

Algorithm: E remove(List l)

if *l.curr.next* = *NULL* **then return** *NULL*;
E *it* ← *l.curr.next.element*;
if *l.tail* = *l.curr.next* **then** *l.tail* ← *l.curr*;
l.curr.next ← *l.curr.next.next*; *l.cnt*--;
return *it*;

`remove(l)`, just after `insert(l,2) ; next(l)`



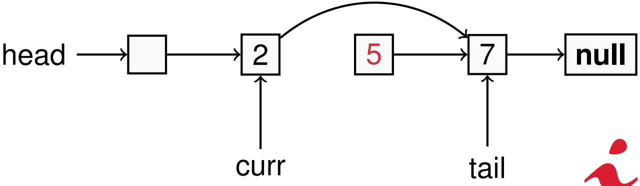
Centro de Informática UFPE
Universidade Federal de Pernambuco

List ADT impl. as a linked list: <2|5,7> ⇝ <2|7>

Algorithm: E remove(List l)

- **if** *l.curr.next* = *NULL* **then return** *NULL*;
- E *it* ← *l.curr.next.element*;
- **if** *l.tail* = *l.curr.next* **then** *l.tail* ← *l.curr*;
- *l.curr.next* ← *l.curr.next.next*; *l.cnt--*;
- **return** *it*;

`remove(l)`, just after `insert(l,2) ; next(l)`

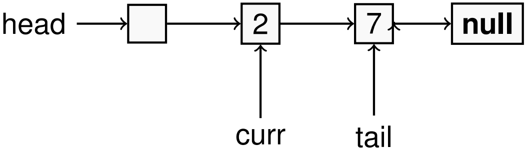


List ADT impl. as a linked list: <2|5,7> $\rightsquigarrow$ <2|7>

Algorithm: E remove(List l)

if $l.curr.next = NULL$ **then return** $NULL$;
E $it \leftarrow l.curr.next.element$;
if $l.tail = l.curr.next$ **then** $l.tail \leftarrow l.curr$;
$l.curr.next \leftarrow l.curr.next.next$; $l.cnt$--;
return it;

After memory deallocation (by the user or by the GC)



Agenda

1 Abstract data types and data structures

2 List ADT

3 List ADT as an array

4 List ADT as a linked list

5 Asymptotic efficiency of lists

6 Bibliography

Asymptotic efficiency of lists[6]

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Centro de Informática UFPE

Universidade Federal de Pernambuco

[6] Source: http://bigocheatsheet.com/

Other considerations

List ADT as an array

- $\pm$ Predefined maximum size (alternative: dynamic arrays)
  - Dynamic allocation of arrays does not imply dynamic arrays
  - Amortised analysis (append): cost of copy becomes not relevant
- $+$ No extra space with pointers (links)
- $-$ Space consumed by unused positions

List ADT as a linked list

- $+$ With no predefined maximum size
- $+$ No extra space for no longer accessible elements
- $-$ Space consumed by pointers (links)

Other considerations

List ADT as an array

- $\pm$ Predefined maximum size (alternative: dynamic arrays)
  - Dynamic allocation of arrays does not imply dynamic arrays
  - Amortised analysis (append): cost of copy becomes not relevant
- $+$ No extra space with pointers (links)
- $-$ Space consumed by unused positions

List ADT as a linked list

- $+$ With no predefined maximum size
- $+$ No extra space for no longer accessible elements
- $-$ Space consumed by pointers (links)

Variations

- Pool of free nodes (freelist)
- Doubly-linked lists
- Circular-linked lists

Agenda

1 Abstract data types and data structures

2 List ADT

3 List ADT as an array

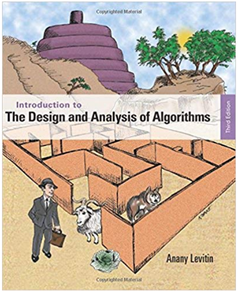
4 List ADT as a linked list

5 Asymptotic efficiency of lists

6 Bibliography

Bibliography

Chapter 1 (pp. 8–12)
Chapter 4 (pp. 93–117)
Clifford Shaffer.
Data Structures and Algorithm Analysis.
Dover, 2013.



Chapter 1 (pp. 25–28).
Anany Levitin.
Introduction to the Design and Analysis of Algorithms.
3rd edition. Pearson. 2011.

Centro de Informática UFPE

Universidade Federal de Pernambuco

LISTS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil