Algorithms and Data Structures

Implementing Lists

Ulf Leser

Content of this Lecture

- ADT List
- Using an Array
- Using a Linked List
- Using a Double-linked List
- Iterators

Lists

- Very often, we want to manage a list of „things“
  - A list of customer names that have an account on a web site
  - A list of windows that are visible on the current screen
  - A list of IDs of students enrolled in a course
- Lists are fundamental: There are objects and lists of objects
- Lists are ordered (1^{st}, 2^{nd}, ... element), but without any defined order (lexicographic , numerical, ...)
  - Unordered lists are typically called sets
  - There are also sorted lists – maintaining a defined order

Representing Lists

- We already discussed an ADT for a list without order

```
type list( T)
operators
  isEmpty:  list → bool;
  add:      list x T → list;
  delete:   list x T → list;
  contains: list x T → bool;
  length:   list → integer;
```

- In the following, we work with ordered lists
  - **`insert(L,t,p)`:** Add element t at pos p of L; if $p=|L|+1$, add t to L
  - **`delete(L,p)`:** Delete element at position p of list L
  - **`search(L,t)`:** Return first pos of t in L if $t \in L$; return 0 otherwise
  - **`elementAt(L,p)`:** Return element at position p of L
  - We require that the order of elements in the list is not changed by any of these operations (but the positions will)

Implementing Lists

- How can we implement this ADT?

```
type list( T)
import integer, bool;
operators
  isEmpty: list → bool;
  insert:  list x integer x T → list;
  delete:  list x int → list;
  search:  list x T → integer;
  elementAt: list x integer → T
  length:  list → integer;
```

- We shall discuss three options
  - Arrays
  - Linked-Lists
  - Double-Linked lists
- We assume values of constant size
  - E.g. real, no strings

Just a Start

- Of course, there are many more issues
  - If the list gets too large to fit into main memory
  - If the list contains complex objects and should be searchable by different attributes (first name, last name, age, ...)
  - If the list is stored on different computers, but should be accessible through a single interface
  - If multiple users can access and modify the list concurrently
  - If the list contains lists as elements (nested lists)
  - ...

Just a Start

- Of course, there are many more issues
  - If the list gets too large to fit into main memory
    - See databases, caching, operating systems
  - If the list contains complex objects and should be searchable by different attributes (first name, last name, age, ...)
    - See databases; multidimensional index structures
  - If the list is stored on different computers, but should be accessible through a single interface
    - See distributed algorithms, cloud-computing, peer-2-peer
  - If different users can access and modify the list concurrently
    - See databases; transactions; parallel/multi-threaded programming
  - If the list contains lists as elements (nested lists)
    - See trees and graphs
  - ...

Content of this Lecture

- ADT List
- Using an Array
- Using a Linked List
- Using a Double-linked List
- Iterators

Lists based on Arrays

- Probably the simplest method
  - Fix a maximal number of elements **max_length**
  - Access elements by their offset within the array
  - Array must be dense – no “holes”
  - We need to maintain the actual **size** of the list – which positions are valid?
  - We may insert only within this **size**
    - Or immediately right for **size**
  - We may delete only within **size**

```
class list {
  size: integer;
  a: array[1..max_length]

  func void init() {
    size := 0;
  }
  func bool isEmpty() {
    if (size=0)
      return true;
    else
      return false;
    end if;
  }
}
```

Insert, Delete, Search (Array of integer)

Problem!

```
func void insert (t real, p integer) {
  if size = max_length then
    return ERROR;
  end if;
  if p!=size+1 then
    if (size<p) or (p<1) then
      return ERROR;
    end if;
    for i := size downto p do
      A[i+1] := A[i];
    end for;
  end if;
  A[p] := t;
  size := size + 1;
}
```

- Complexity (worst-case)?
  - Insert: O(n)
  - Delete: O(n)
  - Search: O(n)
  - elementAt: O(1)

```
func void delete(p integer) {
  if (size<p) or (p<1) then
    return ERROR;
  end if;
  for i := p .. size-1 do
    A[i] := A[i+1];
  end for;
  size := size - 1;
}
```

```
func int search(t real) {
  for i := 1 .. size do
    if A[i]=t then
      return i;
    end if;
  end for;
  return 0;
}
```

```
func int elementAt(p int) {
  if p<1 or p>size then
    return ERROR;
  else
    return A[p];
  end if;
}
```

Properties

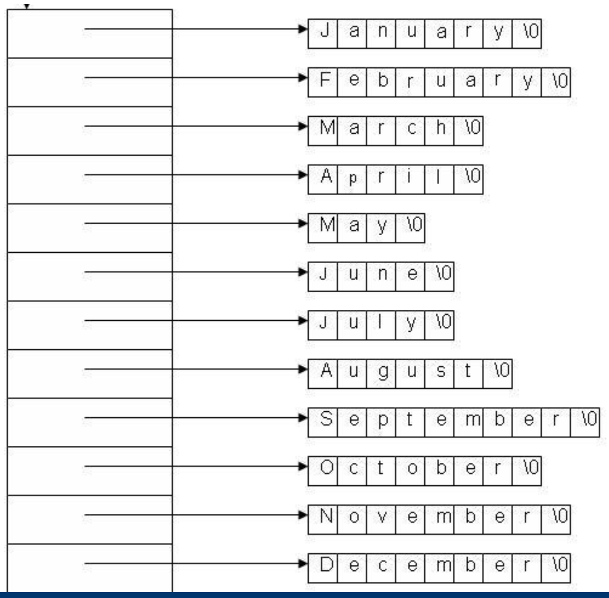
- We can access position p in constant time, but need to move O(n) elements to insert/delete an element
  - If all positions appear with the same probability, we expect n/2 operations on average (still O(n))
  - In stacks or queues, insert/delete positions do not have the same probabilities
  - Unbalanced: Inserting at the end of an array costs O(1), inserting at the start costs O(n) operations
- Disadvantages
  - If `max_length` too small, we run into errors
  - If `max_length` too large, we waste space
- Help: Dynamic arrays (with other disadvantages)
  - See later

Arrays of Strings

- We assumed that every element of the list requires constant space
  - Elements are stored one-after-the-other in main memory
  - Element at position p can be access directly by computing the address of the memory cell
- What happens for other data types, e.g. strings?

Arrays of Strings

- We assumed that every element of the list requires constant space
  - Elements are one-after-the-other in main memory
  - Element at position p can be access directly by computing the address of the memory cell
- What happens for other data types, e.g. strings?
  - Each string actually is a list itself
    - Implemented in whatever way (arrays, linked lists, ...)
  - Thus, we are building a list of lists
  - Array A holds pointer to strings
  - Pointers require constant space



Summary

	Array	Linked list	Double-linked l.
insert	O(n)		
delete	O(n)		
search	O(n)		
add anywhere to list	O(1)		
elementAt	O(1)		
Space	Static, upfront		

Content of this Lecture

- ADT List
- Using an Array
- Using a Linked List
- Using a Double-linked List
- Iterators

Linked Lists (here: of real values)

- The static space allocation is a severe problem of arrays
- Alternative: Linked lists
  - Every list element is a tuple (**value**, **next**)
  - **value** is the value of the element
  - **next** is a pointer to the next element in the list
- Disadvantage: O(n) additional space for all the pointers
  - Space complexity still O(n), but practically this makes a true difference
- Certain properties make slightly different operations attractive

```
class element {
  value: real;
  next: element;
}
```

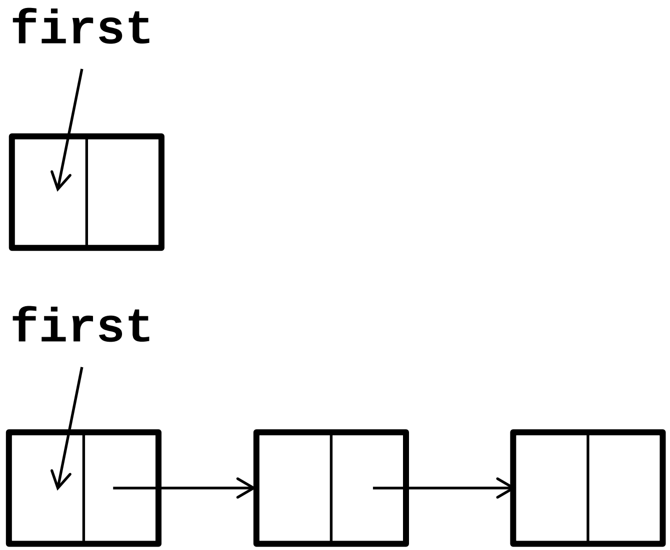
```
class list {
  first: element;

func void init() {
    first := null;
  }
func bool isEmpty() {
    if (first=null)
      return true;
    else
      return false;
    end if;
  }
}
```

Search

- Return the first element with value=t, or null if no such element exists
  - Note: Here we return the element, not the position of the element
  - Makes sense: Returned ptr necessary e.g. to change the value

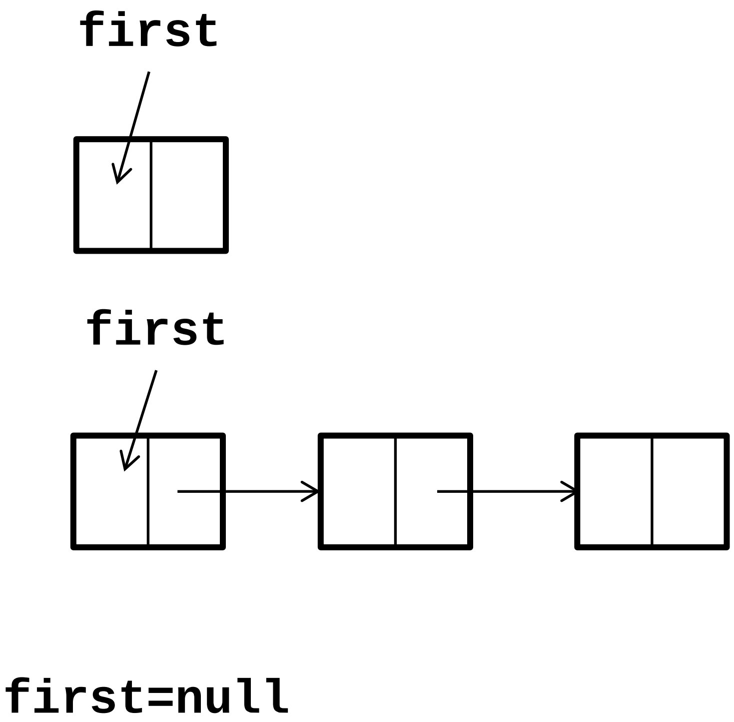
```
func element search(t real) {
  e := first;
  if e.value = t then
    return e;
  end if;
  while (e.next != null) do
    e := e.next;
    if (e.value  = t) then
      return e;
    end if;
  end while;
  return null;
}
```



Search

- Return the first element with value=t, or null if no such element exists

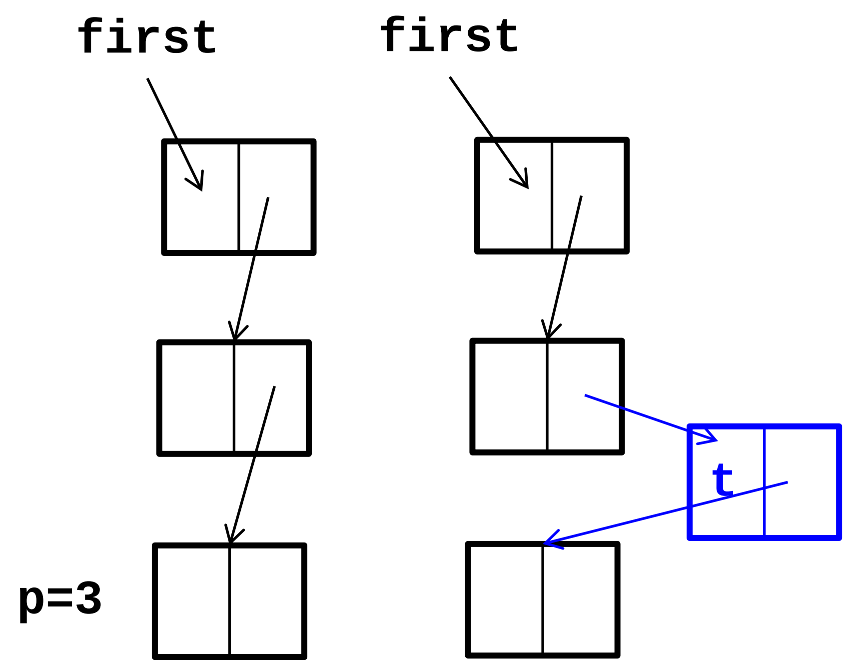
```
func element search(t real) {
  if first=null then
    return null;
  end if;
  e := first;
  if e.value = t then
    return e;
  end if;
  while (e.next != null) do
    e := e.next;
    if (e.value  = t) then
      return e;
    end if;
  end while;
  return null;}
```



Insert

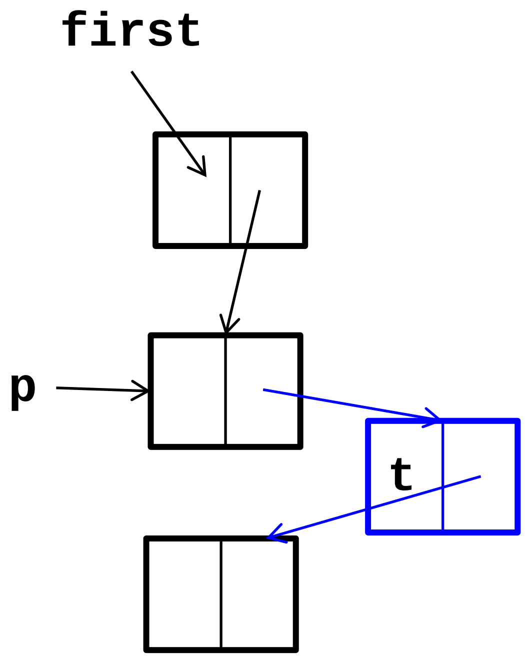
- insert(t, p) as insert(t, p-1) – insert after pth position

```
func void insert (t real, p integer) {
  new := new element (t, null);
  e := first;
  if e=null then
    if p≠1 then
      return ERROR;
    else
      first := new;
      return;
    end if;
  end if;
  for i := 1 .. p-2 do
    if (e.next=null) then
      return ERROR;
    else
      e := e.next;
    end if;
  end for;
  new.next := e.next;
  e.next := new;
}
```



InsertAfter

- In linked lists, a slightly different operation also makes sense: We insert after element t, not at position p
  - E.g., we search an element p and want to insert a new element right after p
- No difference in complexity for arrays, but large difference for linked lists

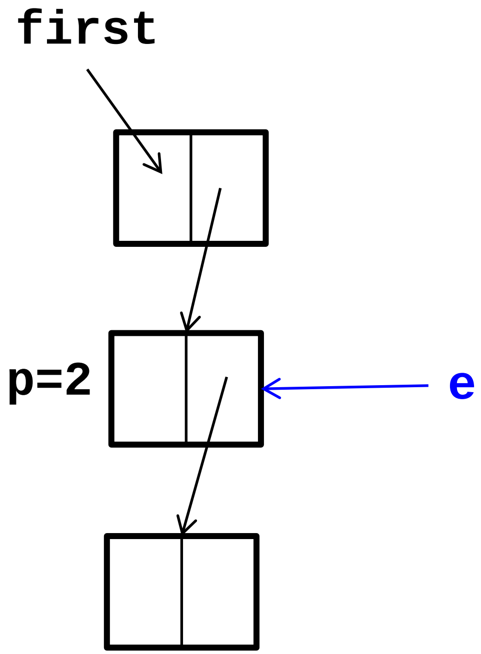



```
func void insertAfter (t real, p element) {
  new := new element (t, null);
  new.next := p.next;
  p.next := new;
}
```

Delete

- Delete the p'th element of the list

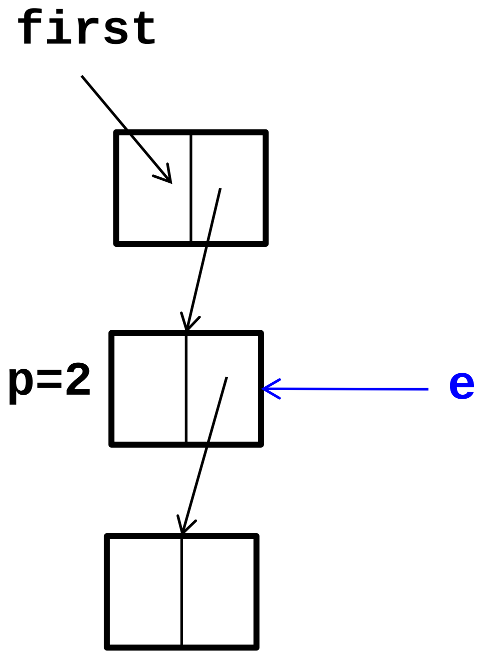
```
func void delete(t real, p integer) {
  e := first;
  if (e=null) or (p<1) then
    return ERROR;
  end if;
  for i := 1 .. p-1 do
    if (e.next=null) then
      return ERROR;
    else
      e := e.next;
    end if;
  end for;
  ? PROBLEM ?
}
```



Delete – Bug-free?

- Delete the p’th element of the list

```
func void delete(t real, p integer) {
  e := first;
  if (e=null) or (p<1) then
    return ERROR;
  end if;
  for i := 1 .. p-1 do
    last := e;
    if (e.next=null) then
      return ERROR;
    else
      e := e.next;
    end if;
  end for;
  last.next := e.next;
}
```

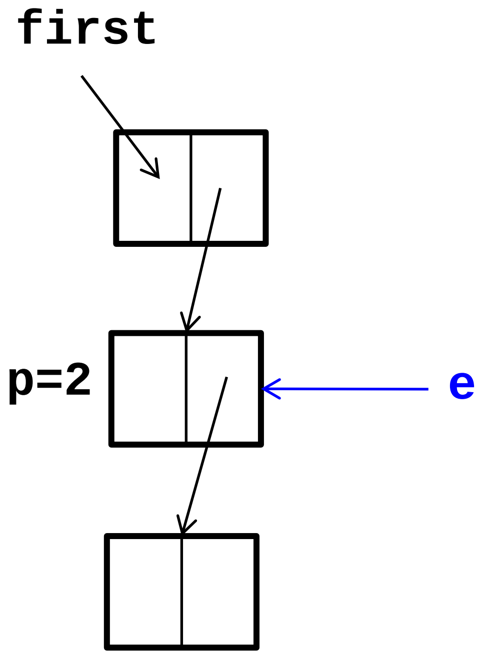


- What if p=1?

Delete – Bug-free

- Delete the p'th element of the list

```
func void delete(t real, p integer) {
  e := first;
  if (e=null) or (p<1) then
    return ERROR;
  end if;
  if p=1 then
    first := e.next;
    return;
  end if;
  for i := 1 .. p-1 do
    last := e;
    if (e.next=null) then
      return ERROR;
    else
      e := e.next;
    end if;
  end for;
  last.next := e.next;
}
```

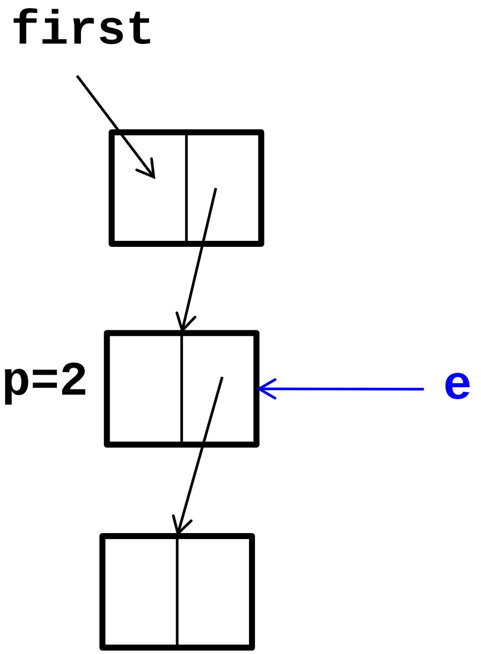


Delete – faster

- Delete the p'th element of the list

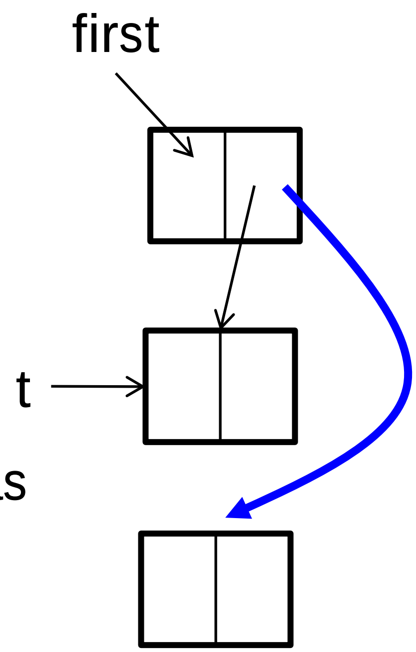
```
func void delete(t real, p integer) {
  e := first;
  if (e=null) or (p<1) or (p>size) then
    return ERROR;
  end if;
  if p=1 then
    first := e.next;
    return;
  end if;
  for i := 1 .. p-1 do
    last := e;
    e := e.next;
  end for;
  last.next := e.next;
}
```

Stop: We neither defined or maintain a list size



DeleteThis

- In linked lists, a slightly different operation sometimes makes more sense: Delete element t, not at position p
  - Again: We search an element t and then want to delete exactly t
- Big problem
  - If we have t, we cannot directly access the predecessor s of t (the s with **`s.next=t`**)
  - We need to go through the entire list to find t (again)
  - Thus, **`deleteThis`** has the same complexity as **`delete`**
  - Remedy not so easy: If we found t, we (clients) don't want to keep predecessor of t



Two More Issues

- Show me the list

```
func String print() {
  if (first=null) then
    return "";
  end if;
  tmp := "";
  while (e≠null) do
    tmp := tmp+e.value;
    e := e.next;
  end for;
  return tmp;
}
```

- What happens to deleted elements t?
  - In most languages, the space occupied by t remains blocked
  - These languages offer an explicit "dispose" which you should use
  - Java: "Dangling" space is freed automatically by garbage collector
    - After some (rather unpredictable) time

Summary

	Array	Linked list	Double-linked l.
Insert	O(n)	O(n)	
InsertAfter	O(n)	O(1)	
Delete	O(n)	O(n)	
DeleteThis	O(n)	O(n)	
Search	O(n)	O(n)	
Add to list	O(1)	O(1)	
elementAt	O(1)	O(n)	
Space	Static	n+1 add. pointers	

How?

Double-Linked List

- Two modifications
  - Every element holds pointers to next and to previous element
  - List holds pointer to first and to last element
- Advantages
  - **`deleteThis`** can be implemented in O(1)
  - Concatenation of lists can be implemented in O(1)
  - Addition/removal of last element can be implemented in O(1)
- Disadvantages
  - Requires more space
    - Beware of the space necessary for a pointer on a 64bit machine
  - Slightly more complicated operations

Summary

Both first have to search – critical operation

	Array	Linked list	Double-linked l.
Insert	O(n)	O(n)	O(n)
InsertAfter	O(n)	O(1)	O(1)
Delete	O(n)	O(n)	O(n)
DeleteThis	O(n)	O(n)	O(1)
Search	O(n)	O(n)	O(n)
Add to start of list	O(n)	O(1)	O(1)
Add to end of list	O(1)	O(n)	O(1)
elementAt	O(1)	O(n)	O(n)
concatenate	O(n)	O(n)	O(1)
Space	Static	n+1 add. pointers	2(n+1)+2 add. point.

Very important advantage

Outlook

- Can we do any better in search?

- Yes – if we sort the list on the searchable value
- Yes – if we know which elements are searched most often

Content of this Lecture

- ADT List
- Using an Array
- Using a Linked List
- Using a Double-linked List
- Iterators

Example

- Assume we have a list of customers with home addresses
- We want to know how many customers we have per city
  - This is a “group-by” in database terms

Meier	Berlin
Müller	Hamburg
Meyer	Dresden
Michel	Hamburg
Schmid	Berlin
Schmitt	Hamburg
Schmidt	Wanne-Eikel
Schmied	Hamburg

⇨

Berlin	2
Hamburg	4
Dresden	1
Wanne-Eikel	1

Using a List

- Assume we have a data type **`groups`** which maintains a list of **`city`** and offers an operation **`increment(city)`**

```
class group {
  count: integer;
  city: string;
}

class groups
import group
…
increment: …

class customer{
  name: string;
  city: string;
}
```

```
func void group_by( customers list;
                    g groups) {
  if customers.isEmpty() then
    return;
  end if;
  c : customer;
  for i:= 1 .. customers.size do
    c := customers.elementAt( i);
    g.increment( c.city);
  end for;
}
```

Complexity?

- We run once through costumers: O(n)
- Complexity of **`elementAt`** depends on list implementation
- For linked lists, this gives $O(n^2)$ in total
  - Only O(n) for arrays, but these had other problems
- Not satisfactory: We are doing unnecessary work
  - We only need to follow pointers – but driven by the client
  - One useful access pattern: Access all elements one after the other
  - But our data type “list” has no state, i.e., no “current” position
  - Without in-list state, the state (variable i) must be managed outside the list, and the list must be put to the right state again for every operation (**`elementAt`**)
  - Remedy: Stateful list ADT

Stateful Lists

```
type slist( T)
import
  integer, bool;
operators
  isEmpty:     slist → bool;
  setState:    slist x integer → slist;
  insertHere:  slist x T → slist;
  deleteHere:  slist x T → slist;
  getNext:     slist → T;
  search:      slist x T → integer;
  size:        slist → integer;
```

- Impl: List holds an internal pointer **`p_current`**
  - This is the state
- **`p_current`** can be set to position p using **`setState()`**
- **`insertHere`** inserts after **`p_current`**, **`deleteHere`** deletes **`p_current`**
- **`getNext()`** returns element at position **`p_current`** and increments **`p_current`** by 1

Using Stateful Lists

```
func void group_by( customers stateful_list;
                    g groups) {
  if customers.isEmpty() then
    return;
  end if;
  c : customer;
  customers.setState(1);
  for i:= 1 .. customers.size-1 do
    c := customers.getNext();
    groups.increment( c.city);
  end for;
  print groups;
}
```

- Advantage: **`getNext()`** can be implemented in O(1)
  - Using linked lists or arrays

Iterators

- **`slist`** only allows to manage one state per list
- What if multiple threads want to read the list concurrently?
  - Every thread needs its own pointer
  - These pointers cannot be managed easily in the (one and only) list itself
- Iterators
  - An iterator is an object created by a list which holds list state
    - One **`p_current`** per iterator
  - Multiple iterators can operate independently on the same list
  - Implementation of iterator depends on implementation of list, but can be kept secret from the client
  - Iterators know about list states (more exposure), but clients don’t

Using an Iterator

```
func void group_by( customers stateful_list
                    g groups) {
  if customers.isEmpty() then
    return;
  end if;
  c : customer;
  it := customers.getIterator();
  while it.hasNext() do
    c := it.getNext();
    groups.increment( c.city);
  end while;
  print groups;
}
```

```
class iterator_for_linked_list (T) {
  p_current: T;

  func iterator init( l list) {
    p_current := l.getFirst();
  }

  func bool hasNext() {
    return (p_current ≠ null);
  }

  func T getNext() {
    if p_current = null then
      return ERROR;
    end if;
    tmp := p_current;
    p_current := p_current.next;
    return tmp;
  }
}
```

Take Home Message

- Finding robust ADTs that can remain stable for many applications is an art
  - See the complexity of standardization processes, e.g. Java community process
  - Growing trend to standardize ADTs / APIs
- Different implementations of an ADT yield different complexities of operations
- Therefore, one needs to look “behind” the ADT if efficient implementations for specific operations are required

Exemplary Questions

- Give pseudo-code for an efficient implementation to delete all elements with a given value v in a (a) linked list, (b) double-linked list
- What is the complexity of searching in an array (a) value at given position p; (b) value at the end of the list; (c) all positions with a given value
- A skip list is a linked list where every element also holds a pointer to the 1^{st}, 2^{nd}, 4^{th}, 8^{th}, ... $\log(n)^{th}$ successor element. (a) Analyze the space complexity of a skip list. What is the complexity of (b) accessing the i^{th} element and of (c) accessing the first element with value v?