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
  - And lists of lists of objects – which are lists of objects (of type list)
- Lists are ordered (1^{st}, 2^{nd}, … element), but without any defined order (lexicographic , numerical, …)
  - Lists have a 1^{st} element, but without any specific property
  - Operations must allow maintenance of this order
    - For instance to keep a sort-order: Sorted lists
  - I.e., the list doesn't change the order by itself

List Operations

- Typical operations on (ordered) lists
  - **`insert(L,t,p)`**: Add element t at pos p of L
    - If p=|L|+1, append t to L
    - If p<1 or p>|L|+1, return error
  - **`delete(L,p)`**: Delete element at position p of list L
    - With p>0 and p<|L|+1; otherwise error
  - **`search(L,t)`**: Return first pos of t in L if t$\in$L; return 0 otherwise
    - "First pos" – values might appear more than once
  - **`elementAt(L,p)`**: Return element at position p of L
    - With p>0 and p<|L|+1; otherwise error
- The order of current elements in the list is not changed by any of these operations (but the positions are)

Question

- How can we implement this ADT?

```
type list( T)
import
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;
```

  - Array
  - Linked list
  - Double-linked list
  - Skip list
  - Binary trees
  - Red-black trees, AVL trees
  - …

Implementing Lists

- How can we implement this ADT?

```
type list( T)
import
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 discuss three options
  - Arrays
  - Linked-Lists
  - Double-Linked lists
- We assume values of constant size
  - 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 of **`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 reals)

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;
}
```

```
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;
}
```

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

Properties

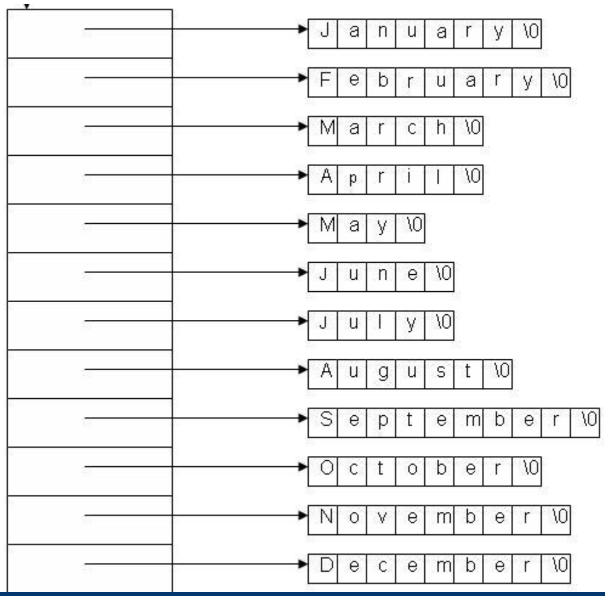
- We can access position p in constant time, but need to move O(n) elements to insert/delete an element
  – If all positions occur with the same probability, we expect n/2 operations on average (still O(n))
    - In stacks or queues, insert/delete positions do not have the same probabilities (leading to different complexities)
  – 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
  – 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 at p	O(n)		
delete at p	O(n)		
search	O(n)		
add	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
  - Special pointer to first element: **first**
- Disadvantage: O(n) additional space for all the pointers
  - Space complexity still O(n), but practically there is a factor of ~2
- 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;
  }
}
```

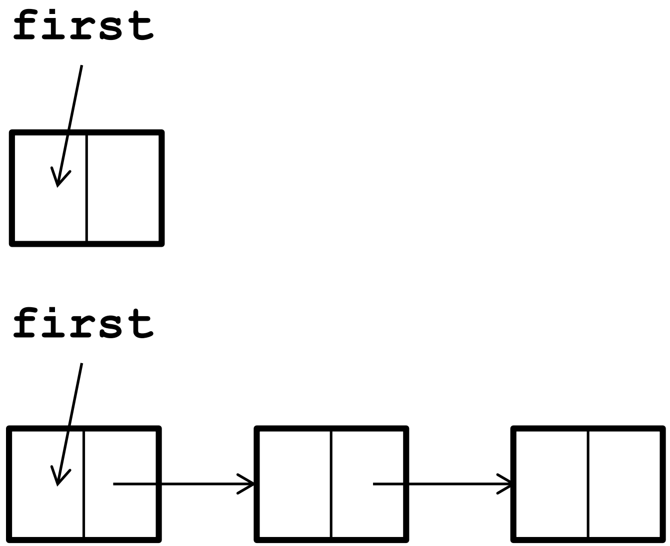
Caveat

- In an ideal world, we would implement exactly the same operations (i.e., the ADT) as with arrays
- But: We will see that this may lead to inefficient algorithms
- We will, however, find very similar operations allowing for efficient implementations with linked lists
- Not unusual – ADTs determine implementations, but implementations also favor ADTs
  - Designing an ADT is not advisable without considering its "implement'ability"

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 value in O(1)

```
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;
}
```

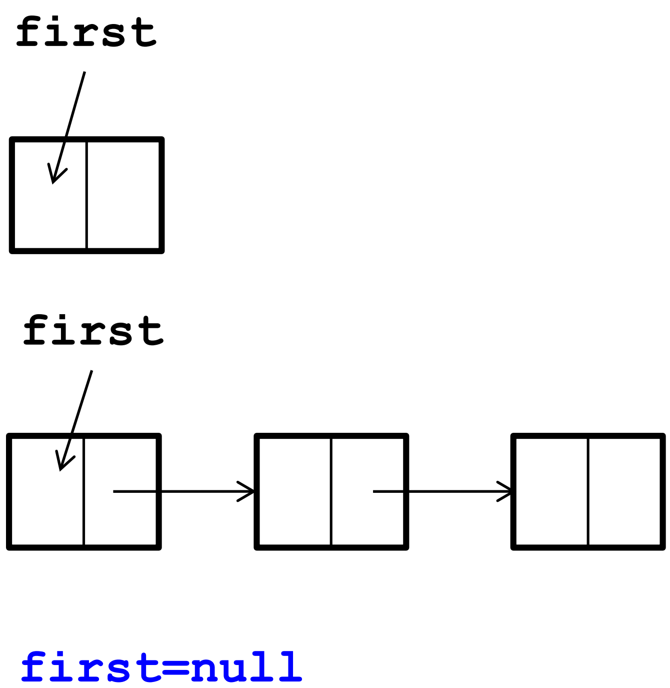


But …

Search

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

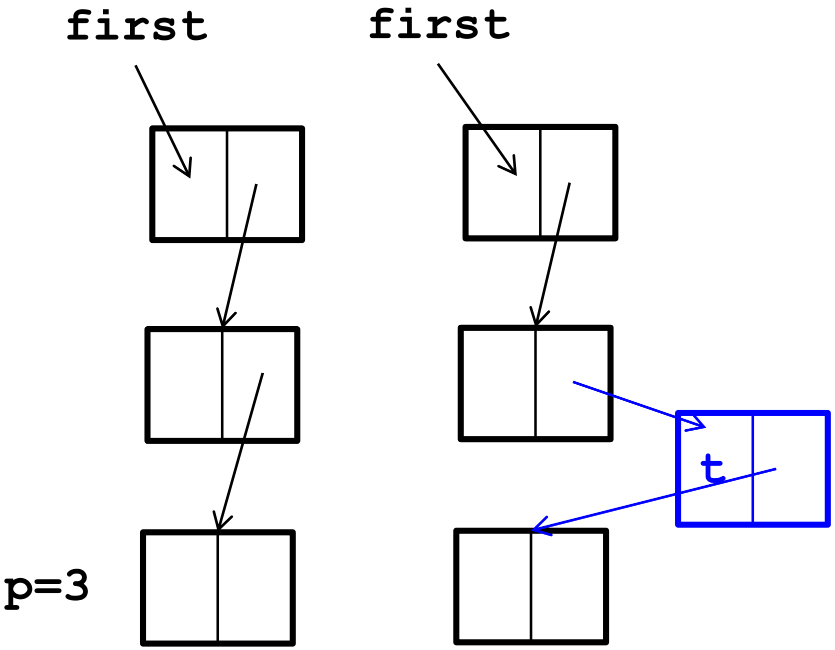
```
func element search(t real) {
  if first=null then
    return null;
  end if;
  e := first;
  if e.value = t then
    return e;
  end if;
  while (e.next != null) do
    e := e.next;
    if (e.value  = t) then
      return e;
    end if;
  end while;
  return null;
}
```



Insert

- insert(t, p) – insert after p-1'th position

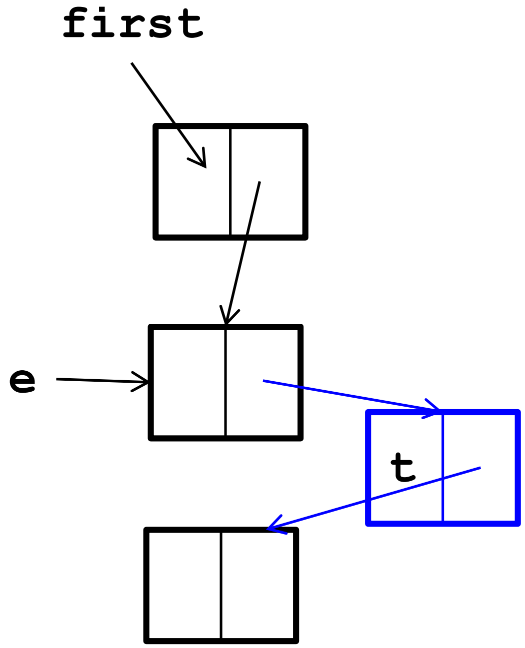
```
func void insert (t real, p integer) {
  new := new element (t, null);
  e := first;
  if e=null then
    if p≠1 then
      return ERROR;
    else
      first := new;
      return;
    end if;
  end if;
  for i := 1 .. p-1 do
    if (e.next=null) then
      return ERROR;
    else
      e := e.next;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



InsertAfter

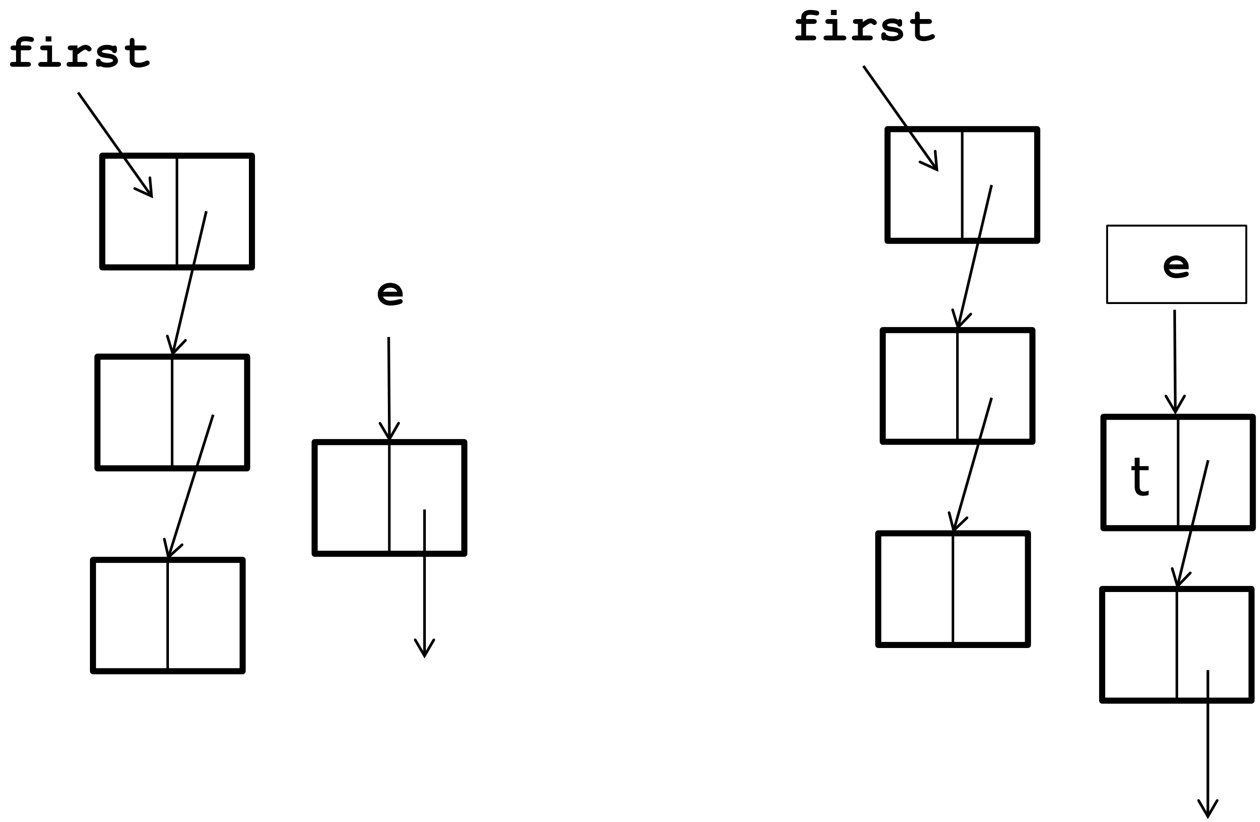
- In linked lists, a slightly different operation also makes sense: We insert after element e, not at position p
  - E.g., we search an element e and want to insert a new element right after e
- No difference in complexity for arrays, but large difference for linked lists

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



Caution

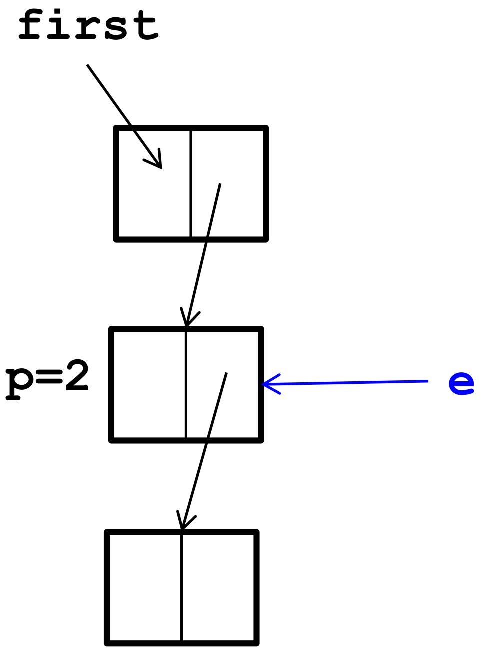
- We did not check if e actually is an element of L; if not, we actually didn't change the list at all



Delete

- Delete the p'th element of the list

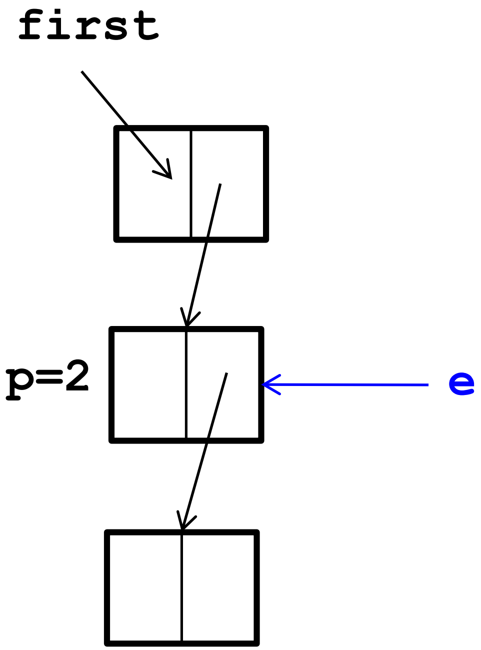
```
func void delete(t real, p integer) {
  e := first;
  if (e=null) or (p<1) then
    return ERROR;
  end if;
  for i := 1 .. p-1 do
    if (e.next=null) then
      return ERROR;
    else
      e := e.next;
    end if;
  end for;
  ? PROBLEM ?
}
```



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;
}
```

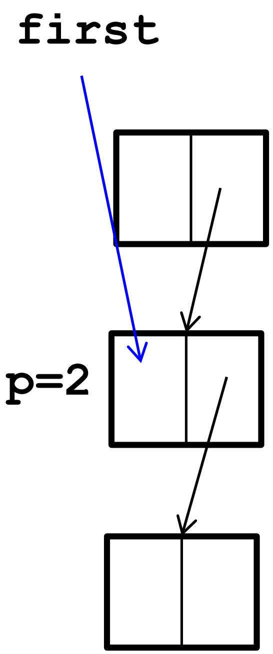


- What if p=1?

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;
}
```

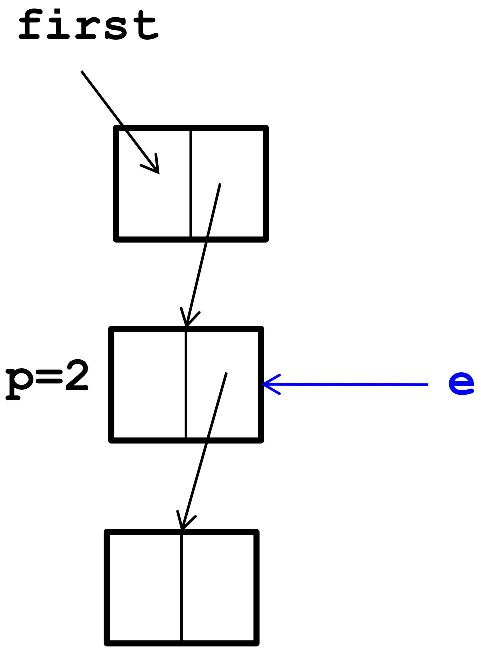


Delete – faster?

- Delete the p'th element of the list

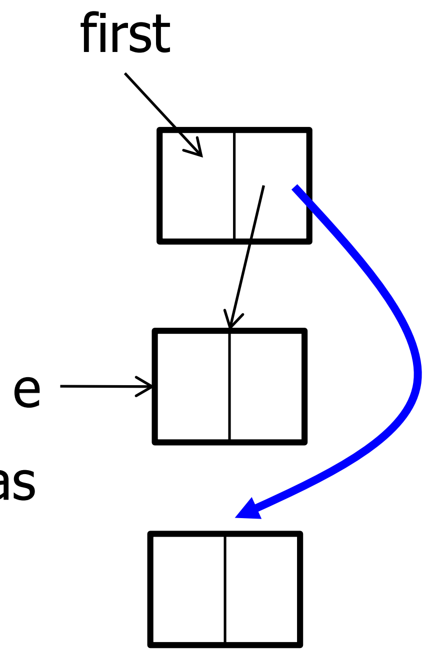
```
func void delete(t real, p integer) {
  e := first;
  if (e=null) or (p<1) or (p>size) then
    return ERROR;
  end if;
  if p=1 then
    first := e.next;
    return;
  end if;
  for i := 1 .. p-1 do
    last := e;
    e := e.next;
  end for;
  last.next := e.next;
}
```

Stop: We neither defined nor maintain a list size



DeleteThis

- In linked lists, a slightly different operation sometimes makes more sense: Delete element e, not at position p
  - Again: We search an element e and then want to delete exactly e
- Big problem
  - If we have e, we cannot directly access the predecessor s of e (the s with **`s.next=e`**)
  - 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 a client found e, it doesn't want to (or can) keep predecessor of e



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 e?
  - In most languages, the space occupied by e 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 at p	O(n)	O(n)	
InsertAfter e	O(n)	O(1)	
Delete at p	O(n)	O(n)	
DeleteThis e	O(n)	O(n)	
Search	O(n)	O(n)	
Add	O(1)	O(1)	
elementAt	O(1)	O(n)	
Space	Static	n+1 add. pointers	

How?

Linked lists as Queues and Stacks

- With O(1) insertion at list head and O(1) elementAt (1), linked lists are perfect means for implementing a stack
- But not for queues: Accessing the last element is O(n)

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)
    - In a linked list, we have to find the last element of the first list: O(n)
    - Compromise: Linked list with additional pointer to last element
  - Addition/removal of last element can be implemented in O(1)
    - Important for queues
- Disadvantages
  - Requires more space
  - Slightly more complicated operations

Summary

Both first have to search – critical operation

	Array	Linked list	Double-linked l.
Insert at p	O(n)	O(n)	O(n)
InsertAfter e	O(n)	O(1)	O(1)
Delete at p	O(n)	O(n)	O(n)
DeleteThis e	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	2n+2 add. point.

Very important advantage

Outlook

- Can we do any better in search?
  - Many lists are searched much more often than modified
  - Queues / stacks are never "searched" and very often modified

- 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`**)
  - Solution: Stateful lists

Stateful Lists

```
type slist( T)
import
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
- Iterating over list is O(n) also for linked lists

Iterators

- **`slist`** only manages one state per list
- What if multiple clients want to read the list concurrently?
  - Every client 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;
  }
}
```

New problems

- Iterators store information about internals of a list
  - Pointer to a “current” element
- Iterators are used when multiple clients read a list
- But what if multiple clients manipulate a list?
  - Other client might delete element that is “current” in some iterator
  - Error
- We need a synchronized list
  - Considerable overhead
  - Makes list operations slower – do you need this?
  - Watch out for concrete implementation of the lists you use

Take Home Message

- Arrays are efficient for accessing elements by position
- LinkedLists are efficient data structures for stacks
- DoubleLinkedLists are efficient data structures for queues
- None of the three allow searching in less than O(n)

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?