Algorithms and Data Structures

Fundamental Data Structures: Lists

Ulf Leser
Content of this Lecture

- ADT List
- Using Arrays
- Linked Lists
- Double-linked Lists
- 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 the students enrolled in a course
  - ...

• Lists are fundamental: We have things and lists of things

• Difference between a list and a set?
  - Lists: Have order, may contain duplicates

• Note: A list has “an” order; this doesn’t imply that we require that the elements of a list have a particular order
  - Like lexicographic / numerical / …
Representing Lists

• We already discussed an ADT for a list

<table>
<thead>
<tr>
<th>type list( T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>operators</td>
</tr>
<tr>
<td>isEmpty: list → bool;</td>
</tr>
<tr>
<td>add: list x T → list;</td>
</tr>
<tr>
<td>delete: list x T → list;</td>
</tr>
<tr>
<td>contains: list x T → bool;</td>
</tr>
<tr>
<td>length: list → integer;</td>
</tr>
</tbody>
</table>

• Today, we require slightly different operations
  - Positions are counted starting from 1
  - insert(t,p,L): Add element t at pos p of L; if p=|L|+1, add t to L
  - delete(p,L): Delete element at position p of list L
  - search(t,L): Return first pos of t in L if t∈L; return 0 otherwise

• We also require that the order of the existing elements in the list is not changes by any of these operations
Representing Lists

• How can we implement this ADT?

```plaintext
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;
    length:  list -> integer;
```

• We shall discuss three options
  - Arrays
  - Linked-Lists
  - Double-Linked lists

• We assume to store values of constant size (e.g. real)
Just a Snapshot

- 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 different users can access and modify the list at the same time
  - If the list contains lists as elements (nested lists)
  - …
Just a Snapshot

- 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 at the same time
    - 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 Arrays
- Linked Lists
- Double-linked Lists
- Iterators
Lists through Arrays

• Probably the simplest method
  - Fix a maximal number of elements max_length
  - Access elements by their offset within the array

```plaintext
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;
}
```
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-1 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;
}

• Complexity (worst-case)?
  - Insert: O(n)
  - Delete: O(n)
  - Search: O(n)
Properties

• Source of complexity: We can jump to position $p$, but we need to move $O(n)$ elements to insert/delete an item
  - If all positions appear with the same probability, we expect $n/2$ operations on average for each insert/delete (still $O(n)$)

• Unbalanced: Adding to the end of an array costs $O(1)$, adding to the start costs $n$ 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)
  - Think of Java vectors …
Arrays of Strings

• We assumed that every element of the list requires constant space
  - Thus, elements of the array lie one-after-the-other in main memory
  - Element at position p can be accessed 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
  - Thus, elements of the array lie one-after-the-other in main memory
  - Element at position p can be accessed 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
  - Thus, we are building a list of lists
  - Array A holds pointer to strings
  - Strings can be implemented in various ways (arrays, linked lists, …)
  - A pointer requires constant space
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Linked list</th>
<th>Double-linked l.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>O(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>O(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add to list</td>
<td>O(1) (or error)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space</td>
<td>Not dynamic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Content of this Lecture

• ADT List
• Using Arrays
• Linked Lists
• Double-linked Lists
• Iterators
Linked Lists

• Static allocation of space is a problem of arrays

• Alternative: Linked lists
  – An element of a list becomes a tuple \((\text{value}, \text{next})\)
  – \text{value} is the value of the element
  – \text{next} is a pointer to the next element in the list

• Disadvantage: \(O(n)\) additional space or all the pointers

```c
class list {
    first: element;

    func void init() {
        first := null;
    }

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

```c
class element {
    value: real;
    next: element;
}
```
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 in linked lists: Access to properties in O(1)

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

\(\text{first} = \text{null}\)
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;
    v := first;
    if v.value = t then
        return v;
    end if;
    while (v.next != null) do
        v := v.next;
        if (v.value = t) then
            return v;
        end if;
    end while;
    return null;
} first
```

Ulf Leser: Alg&DS, Summer semester 2011
Insert

- Insert value \( t \) after the \( p \)'th element of the list

```c
func void insert (t real, p integer) {
    v := first;
    if v=null then
        if p≠1 then
            return ERROR;
        else
            first := new element(t, null);
            return;
        end if;
    end if;
    new := new element (t, null);
    for i := 1 .. p-1 do
        if (v.next=null) then
            return ERROR;
        else
            v := v.next;
        end if;
    end for;
    new.next := v.next;
    v.next := new;
}
```
In linked lists, a slightly different operation also makes sense: We insert after element \( p \), not at position \( p \):
- E.g., we search an element \( p \) and want to insert a new element right after \( p \)

No difference in arrays, but large difference for linked lists

```c
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

```plaintext
func void delete(t real, p integer) {
  v := first;
  if (v=null) or (p=0) then
    return ERROR;
  end if;
  for i := 1 .. p-1 do
    if (v.next=null) then
      return ERROR;
    else
      v := v.next;
    end if;
  end for;
  ? PROBLEM ?
}
```
Delete – Bug-free?

• Delete the p’th element of the list

```java
func void delete(t real, p integer) {
  v := first;
  if (v=null) or (p=0) then
    return ERROR;
  end if;
  for i := 1 .. p-1 do
    last := v;
    if (v.next=null) then
      return ERROR;
    else
      v := v.next;
    end if;
  end for;
  last.next := v.next;
}
```

• What if p=1?
Delete – Bug-free

- Delete the p’th element of the list

```c
func void delete(t real, p integer) {
    v := first;
    if (v=null) or (p=0) then
        return ERROR;
    end if;
    if p=1 then
        first := v.next;
        return;
    end if;
    for i := 1..p-1 do
        last := v;
        if (v.next=null) then
            return ERROR;
        else
            v := v.next;
        end if;
    end for;
    last.next := v.next;
}
```
DeleteThis

- In linked lists, a slightly different operation makes more sense: We delete element $p$, not that at position $p$
  - Again: We search an element $p$ and then want to delete exactly $p$
- Big problem
  - If we have $p$, we cannot directly access the predecessor $q$ of $p$ (the $q$ with $q.next=p$)
  - We need to go through the entire list to find $p$
  - deleteThis has the same complexity as delete, the additional information ($p$) does not help
Two More Issues

- Show me the list

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

- What happens to deleted elements p?
  - In most languages, the space occupied by p 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

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Linked list</th>
<th>Double-linked l.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>InsertAfter</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>DeleteThis</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>Add to list</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Space</td>
<td>Not dynamic</td>
<td>n+1 add pointers</td>
<td></td>
</tr>
</tbody>
</table>

How?
Double-Linked List

- Simple idea: Every element holds a pointer to the next and to the previous element
- List holds pointer to first and 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
    - More pointers to update
### Summary

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Linked list</th>
<th>Double-linked l.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>InsertAfter</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>DeleteThis</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Search</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Add to start of list</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Add to end of list</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Space</td>
<td>Not dynamic</td>
<td>n+1 add. pointers</td>
<td>2(n+1) add. point.</td>
</tr>
</tbody>
</table>

- Can we do any better?
  - Yes – if we may sort the list of the searchable value
  - Yes – if we know which elements are searched most often
Content of this Lecture

- ADT List
- Using Arrays
- Linked Lists
- Double-linked Lists
- Iterators
Example

- Assume we have a list of customers with their home addresses
- We want to know how many customers we have per city

<table>
<thead>
<tr>
<th>Name</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meier</td>
<td>Berlin</td>
</tr>
<tr>
<td>Müller</td>
<td>Hamburg</td>
</tr>
<tr>
<td>Meyer</td>
<td>Dresden</td>
</tr>
<tr>
<td>Michel</td>
<td>Hamburg</td>
</tr>
<tr>
<td>Schmid</td>
<td>Berlin</td>
</tr>
<tr>
<td>Schmitt</td>
<td>Hamburg</td>
</tr>
<tr>
<td>Schmidt</td>
<td>Wanne-Eikel</td>
</tr>
<tr>
<td>Schmied</td>
<td>Hamburg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>City</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>2</td>
</tr>
<tr>
<td>Hamburg</td>
<td>4</td>
</tr>
<tr>
<td>Dresden</td>
<td>1</td>
</tr>
<tr>
<td>Wanne-Eikel</td>
<td>1</td>
</tr>
</tbody>
</table>
Using a List

- Assume we have a list of cities (groups) with elements of class counts, and this list has an operation increment(city)

```
class counts {
    count: integer;
    city: String;
}

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

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

Original “search” searches a value, not the i’th value
Our Lists Lack Functionality

class list {
    func void init() ...
    func bool isEmpty() ...
    func valueAt( i integer) { ... }
    ...
}


Using a List 2

- Assume we have a list of cities (groups) with elements of class counts, and this list has an operation increment(city)

```java
class counts {
    count: integer;
    city: String;
}

class customer{
    name: String
    city: String;
}

func void group_by( customers list) {
    if customers.isEmpty() then
        return;
    end if;

groups: list( counts);
c : customer;
for i:= 1 .. customers.size do
    c := customers.valueAt( i);
    groups.increment( c.city);
end for;
print groups;
}
```
Complexity?

- We run once through customers: $O(n)$
- Complexity of `valueAt()` depends on list implementation
  - Arrays: $O(1)$
  - (Double-)Linked list: $O(n)$
- For linked lists, this gives $O(n^2)$ in total
- Not satisfactory: We are doing a lot of unnecessary work
  - We only need to follow pointers – but driven by the client
  - Our list has **no state**, i.e., no internal “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 (`valueAt`)
  - Remedy: **Stateful list ADT**
Stateful Lists

```plaintext
type list(T)
import
  integer, bool;
operators
  isEmpty: list → bool;
  insert: list x integer → list;
  delete: list x T → list;
...
```

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

```plaintext
type stateful_list(T)
import
  integer, bool;
operators
  isEmpty: list → bool;
  setState: list x integer → list;
  insertHere: list → list;
  deleteHere: list → list;
  getNext: list → T;
  search: list x T → integer;
  size: list → integer;
```
Using Stateful Lists

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

- `customers.setState(0)` sets `p_current` before element 1
- `getNext()` can be implemented in \(O(1)\) using arrays or linked lists
Iterators

- **stateful_lists** allow 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_counter` 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**
Using an Iterator

```plaintext
func void group_by( customers stateful_list) {
    if customers.isEmpty() then
        return;
    end if;
    groups: list( counts);
    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

- ADTs quickly build a complex mesh of dependencies
- 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