DATA STRUCTURES AND ALGORITHMS

LECTURE 6

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 5...

- Singly Linked List
- Doubly Linked List

Today

1. Linked Lists
   - Sorted Lists
   - Linked Lists on Arrays

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	DA	SLL	DLL
search			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
delete last position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
delete last position	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
delete position			

Dynamic Array vs. Linked Lists

- Dynamic Arrays and Linked Lists support the same general operations, but they can have different time complexities

Algorithm	**DA**	**SLL**	**DLL**
search	$O(n)$	$O(n)$	$O(n)$
get element from position	$\Theta(1)$	$O(n)$*	$O(n)$*
insert first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
insert last position	$\Theta(1)$	$O(n)$**	$\Theta(1)$
insert position	$O(n)$	$O(n)$	$O(n)$
delete first position	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
delete last position	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
delete position	$O(n)$	$O(n)$	$O(n)$

Dynamic Array vs. Linked Lists

- Observations regarding the previous table:
  - * - getting the element from a position i for a linked list has complexity $\Theta(i)$ - we need exactly i steps to get to the i^{th} node, but since $i \leq n$ we usually use $O(n)$.
  - ** - can be done in $\Theta(1)$ if we keep the address of the tail node as well.

Dynamic Array vs. Linked Lists

- Advantages of Linked Lists
  - No memory used for non-existing elements.
  - Constant time operations at the beginning of the list.
  - Elements are never *moved* (important if copying an element takes a lot of time).
- Disadvantages of Linked Lists
  - We have no direct access to an element from a given position (however, iterating through all elements of the list using an iterator has $\Theta(n)$ time complexity).
  - Extra space is used up by the addresses stored in the nodes.
  - Nodes are not stored at consecutive memory locations (no benefit from modern CPU caching methods).

Think about it

- Find the n^{th} node from the end of a SLL (optional requirement: passing through the list only once).

Think about it

- Find the n^{th} node from the end of a SLL (optional requirement: passing through the list only once).
- Given the first node of a SLL, determine whether the list ends with a node that has NIL as *next* or whether it ends with a cycle (the *last* node contains the address of a previous node as *next*).
- If the list from the previous problems contains a cycle, find the length of the cycle.
- Find if a SLL has an even or an odd number of elements, without counting the number of nodes in any way.
- Reverse a SLL non-recursively in linear time using $\Theta(1)$ extra storage.

Sorted Lists

- A *sorted list* (or ordered list) is a list in which the elements from the nodes are in a specific order, given by a *relation*.
- This *relation* can be $<$, $\leq$, $>$ or $\geq$, but we can also work with an abstract relation.
- Using an abstract relation will give us more flexibility: we can easily change the relation (without changing the code written for the sorted list) and we can have, in the same application, lists with elements ordered by different relations.

The relation

- You can imagine the *relation* as a function with two parameters (two *TComp* elems):

$$relation(c_1, c_2) = \begin{cases} -1, & c_1 \text{ should be in front of } c_2 \\ 0, & c_1 \text{ and } c_2 \text{ are equal for this relation} \\ 1, & c_2 \text{ should be in front of } c_1 \end{cases}$$

Sorted List - representation

- When we have a sorted list (or any sorted structure or container) we will keep the relation used for ordering the elements as part of the structure. We will have a field that represents this relation.

- In the following we will talk about a *sorted singly linked list* (representation and code for a *sorted doubly linked list* is really similar).

Sorted List - representation

- We need two structures: *Node - SSLLNode* and *Sorted Singly Linked List - SSLL*

SSLLNode:
info: TComp
next: ↑ SSLLNode

SSLL:
head: ↑ SSLLNode
rel: ↑ Relation

SSLL - Initialization

- The relation is passed as a parameter to the *init* function, the function which initializes a new SSLL.
- In this way, we can create multiple SSLLs with different relations.

subalgorithm init (ssll, rel) **is:**
//pre: rel is a relation
//post: ssll is an empty SSLL
  ssll.head $\leftarrow$ NIL
  ssll.rel $\leftarrow$ rel
end-subalgorithm

- Complexity: $\Theta(1)$

SSLL - Operations

- The main difference between the operations of a SLL and a SSLL is related to the insert operation:
  - For a SLL we can insert at the beginning, at the end, at a position, after/before a given element (so we can have multiple insert operations).
  - For a SSLL we have only one insert operation: we no longer can decide where an element should be placed, this is determined by the relation.
- We can still have multiple delete operations.
- We can have search and get element from position operations as well.

SSLL - insert

- Since we have a singly-linked list we need to find the node *after* which we insert the new element (otherwise we cannot set the links correctly).
- The node we want to insert after is the first node whose successor is *greater than* the element we want to insert (where *greater than* is represented by a value 1 returned by the relation).
- We have two special cases:
  - an empty SSLL list
  - when we insert before the first node

SSLL - insert

```
subalgorithm insert (ssll, elem) is:
//pre: ssll is a SSLL; elem is a TComp
//post: the element elem was inserted into ssll to where it belongs
    newNode ← allocate()
    [newNode].info ← elem
    [newNode].next ← NIL
    if ssll.head = NIL then
    //the list is empty
        ssll.head ← newNode
    else if ssll.rel(elem, [ssll.head].info) = -1 then
    //elem is "less than" the info from the head
        [newNode].next ← ssll.head
        ssll.head ← newNode
    else
//continued on the next slide...
```

SSLL - insert

```
        cn ← ssll.head //cn - current node
        while [cn].next ≠ NIL and ssll.rel(elem, [[cn].next].info) > 0 execute
            cn ← [cn].next
        end-while
        //now insert after cn
        [newNode].next ← [cn].next
        [cn].next ← newNode
    end-if
end-subalgorithm
```

- Complexity: $O(n)$

SSLL - Other operations

- The search operation is identical to the search operation for a SLL (except that we can stop looking for the element when we get to the first element that is "greater than" the one we are looking for).
- The delete operations are identical to the same operations for a SLL.
- The return an element from a position operation is identical to the same operation for a SLL.
- The iterator for a SSLL is identical to the iterator to a SLL (discussed in Lecture 3).

SSLL in action

- We define a function that receives as parameter two integer numbers and compares them:

```
function compareGreater(e1, e2) is:
//pre: e1, e2 integer numbers
//post: compareGreater returns -1 if e1 < e2; 0 if they are equal;
//and 1 if e1 > e2
    if e1 < e2 then
        compareGreater ← -1
    else if e1 = e2 then
        compareGreater ← 0
    else
        compareGreater ← 1
    end-if
end-function
```

SSLL in action

- We define another function that compares two integer numbers based on the sum of their digits

```
function compareGreaterSum(e1, e2) is:
//pre: e1, e2 integer numbers
//post: compareGreaterSum returns -1 if the sum of digits of e1 is less than
//that of e2; 0 if the sums are equal; 1 if sum for e1 is greater
    sumE1 ← sumOfDigits(e1)
    sumE2 ← sumOfDigits(e2)
    if sumE1 < sumE2 then
        compareGreaterSum ← -1
    else if sumE1 = sumE2 then
        compareGreaterSum ← 0
    else
        compareGreaterSum ← 1
    end-if
end-function
```

SSLL in action

- Suppose that the *sumOfDigits* function - used on the previous slide - is already implemented
- We define a subalgorithm that prints the elements of a SSLL using an iterator:

```
subalgorithm printWithIterator(ssll) is:
//pre: ssll is a SSLL; post: the content of ssll was printed
    iterator(ssll, it) //create an iterator for ssll
    while valid(it) execute
        getCurrent(it, elem)
        write elem
        next(it)
    end-while
end-subalgorithm
```

SSLL in action

- Now that we have defined everything we need, let's write a short main program, where we create a new SSLL and insert some elements into it and print its content.

```
subalgorithm main() is:
    init(ssll, compareGreater) //use compareGreater as relation
    insert(ssll, 55)
    insert(ssll, 10)
    insert(ssll, 59)
    insert(ssll, 37)
    insert(ssll, 61)
    insert(ssll, 29)
    printWithIterator(ssll)
end-subalgorithm
```

SSLL in action

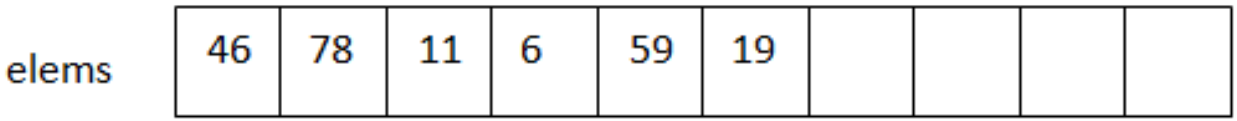
- Executing the *main* function from the previous slide, will print the following: 10, 29, 37, 55, 59, 61.
- Changing only the relation in the *main* function, passing the name of the function *compareGreaterSum*, instead of *compareGreater* as a relation, the order in which the elements are stored, and the output of the function changes to: 10, 61, 37, 55, 29, 59
- Moreover, if I need to, I can have a list with the relation *compareGreater* and another one with the relation *compareGreaterSum*. This is the flexibility that we get by using abstract relations for the implementation of a sorted list.

Linked Lists on Arrays

- What if we need a linked list, but we are working in a programming language that does not offer pointers (or references)?
- We can still implement linked data structures, without the explicit use of pointers or memory addresses, simulating them using arrays and array indexes.

Linked Lists on Arrays

- Usually, when we work with arrays, we store the elements in the array starting from the leftmost position and place them one after the other (no empty spaces in the middle of the list are allowed).
- The order of the elements is given by the order in which they are placed in the array.



- Order of the elements: 46, 78, 11, 6, 59, 19

Linked Lists on Arrays

- We can define a linked data structure on an array, if we consider that the order of the elements is not given by their relative positions in the array, but by an integer number associated with each element, which shows the index of the next element in the array.

elems	46	78	11	6	59	19				
next	5	6	1	-1	2	4				

head = 3

- Order of the elements: 11, 46, 59, 78, 19, 6

Linked Lists on Arrays

- Now, if we want to delete the number 46 (which is actually the second element of the list), we do not have to move every other element to the left of the array, we just need to modify the links:

elems		78	11	6	59	19			
next		6	5	-1	2	4			

head = 3

- Order of the elements: 11, 59, 78, 19, 6

Linked Lists on Arrays

- If we want to insert a new element, for example 44, at the 3^{rd} position in the list, we can put the element anywhere in the array, the important part is setting the links correctly:

elems		78	11	6	59	19		44		
next		6	5	-1	8	4		2		

head = 3

- Order of the elements: 11, 59, 44, 78, 19, 6

Linked Lists on Arrays

- When a new element needs to be inserted, it can be put to any empty position in the array. However, finding an empty position has $O(n)$ complexity, which will make the complexity of any insert operation (anywhere in the list) $O(n)$. In order to avoid this, we will keep a linked list of the empty positions as well.

elems		78	11	6	59	19		44		
next	7	6	5	-1	8	4	9	2	10	-1

head = 3

firstEmpty = 1

Linked Lists on Arrays

- In a more formal way, we can simulate a singly linked list on an array with the following:
  - an array in which we will store the elements.
  - an array in which we will store the links (indexes to the next elements).
  - the capacity of the arrays (the two arrays have the same capacity, so we need only one value).
  - an index to tell where the *head* of the list is.
  - an index to tell where the first empty position in the array is.

SLL on Array - Representation

- The representation of a singly linked list on an array is the following:

SLLA:
elems: TElem[]
next: Integer[]
cap: Integer
head: Integer
firstEmpty: Integer

SLLA - Operations

- We can implement for a SLLA any operation that we can implement for a SLL:
  - insert at the beginning, end, at a position, before/after a given value
  - delete from the beginning, end, from a position, a given element
  - search for an element
  - get an element from a position

SLLA - Init

subalgorithm init(slla) **is:**
//pre: true; post: slla is an empty SLLA
  slla.cap ← INIT_CAPACITY
  slla.elems ← @an array with slla.cap positions
  slla.next ← @an array with slla.cap positions
  slla.head ← -1
  for i ← 1, slla.cap-1 **execute**
    slla.next[i] ← i + 1
  end-for
  slla.next[slla.cap] ← -1
  slla.firstEmpty ← 1
end-subalgorithm

- Complexity: $\Theta(n)$

SLLA - Search

```
function search (slla, elem) is:
//pre: slla is a SLLA, elem is a TElem
//post: return True is elem is in slla, False otherwise
    current ← slla.head
    while current ≠ -1 and slla.elems[current] ≠ elem execute
        current ← slla.next[current]
    end-while
    if current ≠ -1 then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Complexity: $O(n)$

SLLA - Search

- From the *search* function we can see how we can go through the elements of a SLLA (and how similar this traversal is to the one done for a SLL):
  - We need a *current* element used for traversal, which is initialized to the index of the *head* of the list.
  - We stop the traversal when the value of *current* becomes -1
  - We go to the next element with the instruction: *current* $\leftarrow$ *slla.next[current]*.

SLLA - InsertFirst

```
subalgoritm insertFirst(slla, elem) is:
//pre: slla is an SLLA, elem is a TElem
//post: the element elem is added at the beginning of slla
    if slla.firstEmpty = -1 then
        newElems ← @an array with slla.cap * 2 positions
        newNext ← @an array with slla.cap * 2 positions
        for i ← 1, slla.cap execute
            newElems[i] ← slla.elems[i]
            newNext[i] ← slla.next[i]
        end-for
        for i ← slla.cap + 1, slla.cap*2 - 1 execute
            newNext[i] ← i + 1
        end-for
        newNext[slla.cap*2] ← -1
//continued on the next slide...
```

SLLA - InsertFirst

```
    //free slla.elems and slla.next if necessary
    slla.elems ← newElems
    slla.next ← newNext
    slla.firstEmpty ← slla.cap+1
    slla.cap ← slla.cap * 2
  end-if
  newPosition ← slla.firstEmpty
  slla.elems[newPosition] ← elem
  slla.firstEmpty ← slla.next[slla.firstEmpty]
  slla.next[newPosition] ← slla.head
  slla.head ← newPosition
end-subalgorithm
```

- Complexity: $\Theta(1)$ amortized

SLLA -InsertPosition

subalgorithm insertPosition(slla, elem, poz) **is:**
//pre: slla is an SLLA, elem is a TElem, poz is an integer number
//post: the element elem is inserted into slla at position pos
  if (pos $<$ 1) **then**
    @error, invalid position
  end-if
  if slla.firstEmpty $=$ -1 **then**
    @resize
  end-if
  if poz $=$ 1 **then**
    insertFirst(slla, elem)
  else
    pozCurrent $\leftarrow$ 1
    nodCurrent $\leftarrow$ slla.head
//continued on the next slide...

SLLA - InsertPosition

while nodCurrent $\neq$ -1 **and** pozCurrent $<$ poz - 1 **execute**
  pozCurrent $\leftarrow$ pozCurrent + 1
  nodCurrent $\leftarrow$ slla.next[nodCurrent]
end-while
if nodCurrent $\neq$ -1 **atunci**
  newElem $\leftarrow$ slla.firstEmpty
  slla.firstEmpty $\leftarrow$ slla.next[slla.firstEmpty]
  slla.elems[newElem] $\leftarrow$ elem
  slla.next[newElem] $\leftarrow$ slla.next[nodCurrent]
  slla.next[nodCurrent] $\leftarrow$ newElem
else
//continued on the next slide...

SLLA - InsertPosition

```
        @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity: $O(n)$

SLLA - InsertPosition

- Observations regarding the *insertPosition* subalgorithm
  - The *resize* operation is done in the exact same way as for the *insertFirst*.
  - Similar to the SLL, we iterate through the list until we find the element *after* which we insert (denoted in the code by *nodCurrent* - which is an index in the array).
  - We treat as a special case the situation when we insert at the first position (no node to insert after).