DATA STRUCTURES AND ALGORITHMS
LECTURE 5

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 4...

- Binary Heap

- Singly Linked List

  - Insert at the beginning
  - Insert after a node

Today

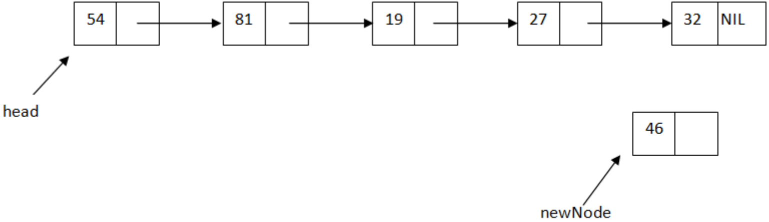
1. Linked Lists
   - Singly Linked Lists
   - Doubly Linked Lists

SLL - Insert at a position

- We usually do not have the node after which we want to insert an element: we either know the position to which we want to insert, or know the element (not the node) after which we want to insert an element.
- Suppose we want to insert a new element at position p (after insertion the new element will be at position p). Since we only have access to the *head* of the list we first need to find the position *after* which we insert the element.

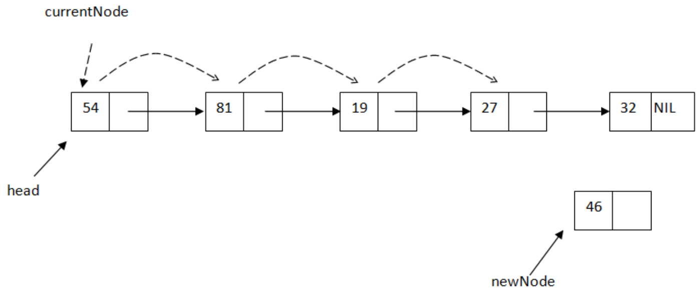
SLL - Insert at a position

- We want to insert element 46 at position 5.



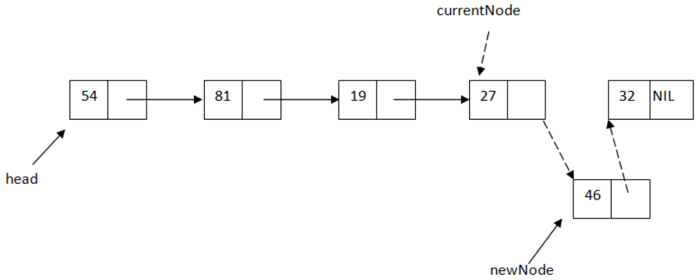
SLL - Insert at a position

- We need the 4^{th} node (to insert element 46 after it), but we have direct access only to the first one, so have to take an auxilary node (*currentNode*) to get to the position.



SLL - Insert at a position

- Now we insert after node *currentNode* (like in the previous case, when we already had the node).



SLL - Insert at a position

subalgorithm insertPosition(sll, pos, elem) **is:**

SLL - Insert at a position

subalgorithm insertPosition(sll, pos, elem) **is:**
//pre: sll is a SLL; pos is an integer number; elem is a TElem
//post: a node with TElem will be inserted at position pos
    if pos $<$ 1 **then**
        @error, invalid position
    else if pos $=$ 1 **then** *//we want to insert at the beginning*
        insertFirst(sll, elem)
    else
        currentNode $\leftarrow$ sll.head
        currentPos $\leftarrow$ 1
        while currentPos $<$ pos - 1 **and** currentNode $\neq$ NIL **execute**
            currentNode $\leftarrow$ [currentNode].next
            currentPos $\leftarrow$ currentPos $+$ 1
        end-while
//continued on the next slide...

```
    if currentNode ≠ NIL then
        insertAfter(sll, currentNode, elem)
    else
        @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity:

```
    if currentNode ≠ NIL then
        insertAfter(sll, currentNode, elem)
    else
        @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity: $O(n)$

Get element from a given position

- Since we only have access to the head of the list, if we want to get an element from a position p we have to go through the list, node-by-node until we get to the p^{th} node.

Get element from a given position

subalgorithm getElement(sll, pos, elem) **is:**

Get element from a given position

subalgorithm getElement(sll, pos, elem) **is:**
//pre: sll is a SLL; pos is an integer number
//post: elem is a TElem, the one from position pos from sll
    if pos $<$ 1 **then**
        @error, invalid position
    else
        currentNode $\leftarrow$ sll.head
        currentPos $\leftarrow$ 1
        while currentPos $<$ pos **and** currentNode $\neq$ NIL **execute**
            currentNode $\leftarrow$ [currentNode].next
            currentPos $\leftarrow$ currentPos $+$ 1
        end-while
        if currentNode $\neq$ NIL **then**
            elem $\leftarrow$ [currentNode].info
        else
            @error, invalid position
        end-if
    end-if
end-subalgorithm

Get element from a given position

- Complexity:

Get element from a given position

- Complexity: $O(n)$

SLL - Delete from the beginning

- Deleting a node from the beginning simply means setting the head of the list to the next element

function deleteFirst(sll) **is:**

SLL - Delete from the beginning

- Deleting a node from the beginning simply means setting the head of the list to the next element

```
function deleteFirst(sll) is:
//pre: sll is a SLL
//post: the first node from sll is deleted and returned
   deletedNode ← NIL
   if sll.head ≠ NIL then
      deletedNode ← sll.head
      sll.head ← [sll.head].next
   end-if
   deleteFirst ← deletedNode
end-function
```

- Complexity:

SLL - Delete from the beginning

- Deleting a node from the beginning simply means setting the head of the list to the next element

function deleteFirst(sll) **is:**
//pre: sll is a SLL
//post: the first node from sll is deleted and returned
  deletedNode $\leftarrow$ NIL
  if sll.head $\neq$ NIL **then**
    deletedNode $\leftarrow$ sll.head
    sll.head $\leftarrow$ [sll.head].next
  end-if
  deleteFirst $\leftarrow$ deletedNode
end-function

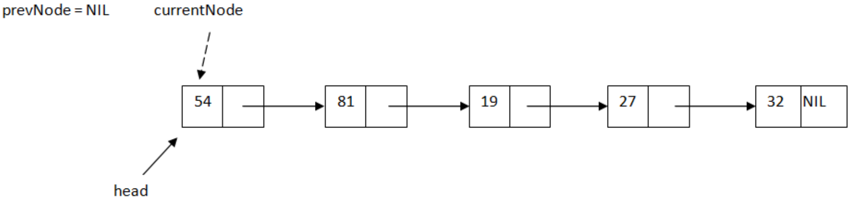
- Complexity: $\Theta(1)$

SLL - Delete a given element

- When we want to delete a node from the middle of the list (either a node with a given element, or a node from a position), we need to find the node *before* the one we want to delete.
- The simplest way to do this, is to walk through the list using two pointers: *currentNode* and *prevNode* (the node before *currentNode*). We will stop when *currentNode* points to the node we want to delete.

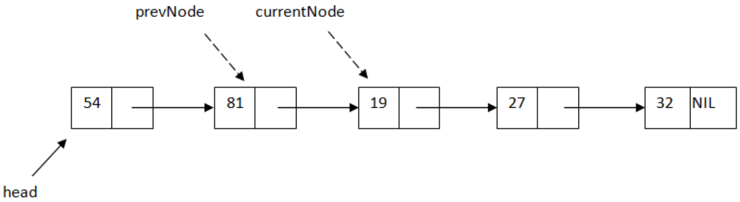
SLL - Delete a given element

- Suppose we want to delete the node with information 19.



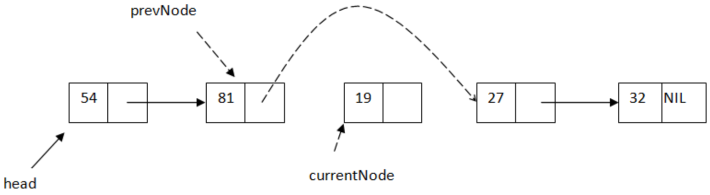
SLL - Delete a given element

- Move with the two pointers until *currentNode* is the node we want to delete.



SLL - Delete a given element

- Delete *currentNode* by *jumping over it*



SLL - Delete a given element

function deleteElement(sll, elem) **is:**

SLL - Delete a given element

```
function deleteElement(sll, elem) is:
//pre: sll is a SLL, elem is a TElem
//post: the node with elem is removed from sll and returned
   currentNode ← sll.head
   prevNode ← NIL
   while currentNode ≠ NIL and [currentNode].info ≠ elem execute
      prevNode ← currentNode
      currentNode ← [currentNode].next
   end-while
   if prevNode = NIL then //we delete the head
      currentNode ← deleteFirst(sll)
   else if currentNode ≠ NIL then
      [prevNode].next ← [currentNode].next
      [currentNode].next ← NIL
   end-if
   deleteElement ← currentNode
end-function
```

SLL - Delete a given element

- Complexity of *deleteElement* function:

SLL - Delete a given element

- Complexity of *deleteElement* function: $O(n)$

SLL - Other operations

- Insert element at the end of the list - walk through the list until we find the last node, add a new node after it
- Delete element from the end of the list - walk through the list (with two nodes) until we find the last node, and delete it.
- Get length of the list - walk through the list and count how many nodes it has

SLL - Iterator

- How can we define an iterator for a SLL?
- Remember, an iterator needs a reference to a *current element* from the data structure it iterates over. How can we denote a *current element* for a SLL?

SLL - Iterator

- In case of a SLL, the current element from the iterator is actually a node of the list.

<u>SLLIterator:</u>
list: SLL
currentElement: $\uparrow$ SLLNode

SLL - Iterator - init operation

- What should the *init* operation do?

SLL - Iterator - init operation

- What should the *init* operation do?

subalgorithm init(it, sll) **is:**
//pre: sll is a SLL
//post: it is a SLLIterator over sll
  it.sll ← sll
  it.currentElement ← sll.head
end-subalgorithm

- Complexity:

SLL - Iterator - init operation

- What should the *init* operation do?

subalgorithm init(it, sll) **is:**
//pre: sll is a SLL
//post: it is a SLLIterator over sll
  it.sll ← sll
  it.currentElement ← sll.head
end-subalgorithm

- Complexity: $\Theta(1)$

SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

subalgorithm getCurrent(it, e) **is:**
//pre: it is a SLLIterator, it is valid
//post: e is TElem, e is the current element from it
  e ← [it.currentElement].info
end-subalgorithm

- Complexity:

SLL - Iterator - getCurrent operation

- What should the *getCurrent* operation do?

subalgorithm getCurrent(it, e) **is:**
//pre: it is a SLLIterator, it is valid
//post: e is TElem, e is the current element from it
  e ← [it.currentElement].info
end-subalgorithm

- Complexity: $\Theta(1)$

SLL - Iterator - next operation

- What should the *next* operation do?

SLL - Iterator - next operation

- What should the *next* operation do?

subalgorithm next(it) **is:**
//pre: it is a SLLIterator, it is valid
//post: it' is a SLLIterator, the current element from it' refers to the next element
  it.currentElement ← [it.currentElement].next
end-subalgorithm

- Complexity:

SLL - Iterator - next operation

- What should the *next* operation do?

subalgorithm next(it) **is:**
//pre: it is a SLLIterator, it is valid
//post: it' is a SLLIterator, the current element from it' refers to the next element
  it.currentElement ← [it.currentElement].next
end-subalgorithm

- Complexity: $\Theta(1)$

SLL - Iterator - valid operation

- What should the *valid* operation do?

SLL - Iterator - valid operation

- What should the *valid* operation do?

function valid(it) **is:**
//pre: it is a SLLIterator
//post: true if it is valid, false otherwise
  if it.currentElement $\neq$ NIL **then**
    valid $\leftarrow$ True
  else
    valid $\leftarrow$ False
  end-if
end-subalgorithm

- Complexity:

SLL - Iterator - valid operation

- What should the *valid* operation do?

function valid(it) **is:**
//pre: it is a SLLIterator
//post: true if it is valid, false otherwise
  if it.currentElement $\neq$ NIL **then**
    valid $\leftarrow$ True
  else
    valid $\leftarrow$ False
  end-if
end-subalgorithm

- Complexity: $\Theta(1)$

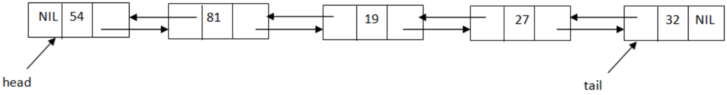
Iterating through all the elements of a sll

- Similar to the DynamicArray, if we want to go through all the elements of a singly linked list, we have two options:
  - Use an iterator
  - Use a for loop and the *getElement* subalgorithm
- What is the complexity of the two approaches?

Doubly Linked Lists - DLL

- A doubly linked list is similar to a singly linked list, but the nodes have references to the address of the previous node as well (besides the *next* link, we have a *prev* link as well).
- If we have a node from a DLL, we can go to the next node or to the previous one: we can walk through the elements of the list in both directions.
- The *prev* link of the first element is set to *NIL* (just like the *next* link of the last element).

Example of a Doubly Linked List



- Example of a doubly linked list with 5 nodes.

Doubly Linked List - Representation

- For the representation of a DLL we need two structures: one struture for the node and one for the list itself.

<u>DLLNode:</u>
    info: TElem
    next: $\uparrow$ DLLNode
    prev: $\uparrow$ DLLNode

Doubly Linked List - Representation

- For the representation of a DLL we need two structures: one struture for the node and one for the list itself.

DLLNode:
  info: TElem
  next: ↑ DLLNode
  prev: ↑ DLLNode

DLL:
  head: ↑ DLLNode
  tail: ↑ DLLNode

DLL - Operations

- We can have the same operations on a DLL that we had on a SLL:
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, etc.)
  - delete an element (from the beginning, from the end, from a given positions, etc.)
  - get an element from a position
- Some of the operations have the exact same implementation as for SLL (e.g. search, get element), other have similar implementations. In general, we need to modify more links and have to pay attention to the *tail* node.

DLL - Insert at the end

- Inserting a new element at the end of a DLL is simple, because we have the *tail* of the list, we no longer have to walk through all the elements.

```
subalgorithm insertLast(dll, elem) is:
//pre: dll is a DLL, elem is TElem
//post: elem is added to the end of dll
   newNode ← allocate() //allocate a new DLLNode
   [newNode].info ← elem
   [newNode].next ← NIL
   [newNode].prev ← dll.tail
   if dll.head = NIL then //the list is empty
      dll.head ← newNode
      dll.tail ← newNode
   else
      [dll.tail].next ← newNode
      dll.tail ← newNode
   end-if
end-subalgorithm
```

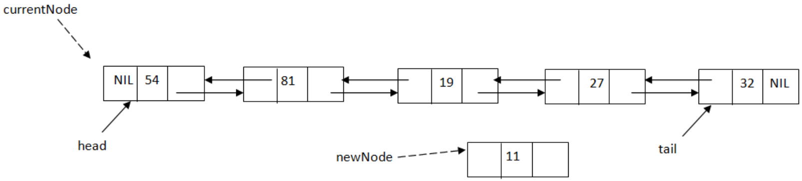
- Complexity: $\Theta(1)$

DLL - Insert on position

- The basic principle of inserting a new element at a given position is the same as in case of SLL.
- The main difference is that we need to set more links (we have the *prev* links as well) and we have to check whether we modify the tail of the list.
- In case of SLL we *had to* stop at the node after which we wanted to insert an element, in case of DLL we can stop before or after the node (but we have to decide in advance, because this decision influences the special cases we need to test).

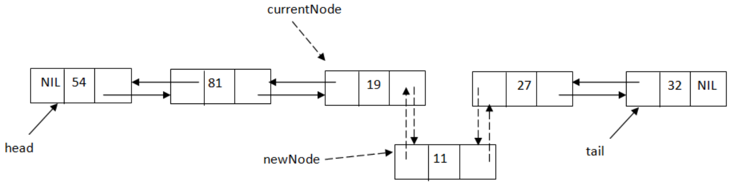
DLL - Insert on position

- Let's insert value 46 at the 4^{th} position in the following list:



DLL - Insert on position

- We move with the *currentNode* to position 3, and set the 4 links.



DLL - Insert at a position

subalgorithm insertPosition(dll, pos, elem) **is:**
//pre: dll is a DLL; pos is an integer number; elem is a TElem
//post: elem will be inserted on position pos in dll
    if pos $<$ 1 **then**
        @ error, invalid position
    else if pos $=$ 1 **then**
        insertFirst(dll, elem)
    else
        currentNode $\leftarrow$ dll.head
        currentPos $\leftarrow$ 1
        while currentNode $\neq$ NIL **and** currentPos $<$ pos - 1 **execute**
            currentNode $\leftarrow$ [currentNode].next
            currentPos $\leftarrow$ currentPos $+$ 1
        end-while
//continued on the next slide...

DLL - Insert at position

```
    if currentNode = NIL then
        @error, invalid position
    else if currentNode = dll.tail then
        insertLast(dll, elem)
    else
        newNode ← alocate()
        [newNode].info ← elem
        [newNode].next ← [currentNode].next
        [newNode].prev ← currentNode
        [[currentNode].next].prev ← newNode
        [currentNode].next ← newNode
    end-if
  end-if
end-subalgorithm
```

- Complexitate: $O(n)$

DLL - Insert at a position

- Observations regarding the *insertPosition* subalgorithm:
  - We did not implement the *insertFirst* subalgorithm, but we suppose it exists.
  - The order in which we set the links is important: reversing the setting of the last two links will lead to a problem with the list.
  - It is possible to use two *currentNodes*: after we found the node after which we insert a new element, we can do the following:

nodeAfter ← currentNode
nodeBefore ← [currentNode].next
//now we insert between nodeAfter and nodeBefore
[newNode].next ← nodeBefore
[newNode].prev ← nodeAfter
[nodeBefore].prev ← newNode
[nodeAfter].next ← newNode

DLL - Delete from the beginning

function deleteFirst(dll) **is:**
//pre: dll is a DLL
//post: the first node is removed and returned
    deletedNode ← NIL
    if dll.head ≠ NIL **then**
        deletedNode ← dll.head
        if dll.head = dll.tail **then**
            dll.head ← NIL
            dll.tail ← NIL
        else
            dll.head ← [dll.head].next
            [dll.head].prev ← NIL
        end-if
    @set links of deletedNode to NIL
    deleteFirst ← deletedNode
end-function

DLL - Delete from the beginning

- Complexity of *deleteFirst*: $\Theta(1)$

DLL - Delete a given element

- If we want to delete a node with a given element, we first have to find the node:
  - we can use the *search* function (discussed at SLL, but it is the same here as well)
  - we can walk through the elements of the list until we find the node with the element (this is implemented below)

DLL - Delete a given element

function deleteElement(dll, elem) **is:**
//pre: dll is a DLL, elem is a TElem
//post: the node with element elem will be removed and returned
    currentNode ← dll.head
    while currentNode ≠ NIL **and** [currentNode].info ≠ elem **execute**
        currentNode ← [currentNode].next
    end-while
    deletedNode ← currentNode
    if currentNode ≠ NIL **then**
        if currentNode = dll.head **then**
            deleteElement ← deleteFirst(dll)
        else if currentNode = dll.tail **then**
            deleteElement ← deleteLast(dll)
        else
//continued on the next slide...

DLL - Delete a given element

[[currentNode].next].prev ← [currentNode].prev
[[currentNode].prev].next ← [currentNode].next
@set links of deletedNode to NIL
end-if
end-if
deleteElement ← deletedNode
end-function

- Complexity: $O(n)$
- If we used the *search* algorithm to find the node to delete, the complexity would still be $O(n)$ - *deleteElement* would be $\Theta(1)$, but searching is $O(n)$

DLL - Iterator

- The iterator for a DLL is identical to the iterator for the SLL (but *currentNode* is *DLLNode* not *SLLNode*).
- In case of a DLL it is easy to define a bi-directional iterator:
  - Besides the operations for the unidirectional iterator, we need another operation: *previous*.
  - It would be useful to define two operations: *first* and *last* to set the iterator to the head/tail of the list.

Think about it

- How could we define a bi-directional iterator for a SLL? What would be the complexity of the *previous* operation?

- How could we define a bi-directional iterator for a SLL if we know that the *previous* operation will never be called twice consecutively (two consecutive calls for the *previous* operation will always be divided by at least one call to the *next* operation)? What would be the complexity of the operations?