DATA STRUCTURES

Linked Lists

Lect. PhD. Diana-Lucia Miholca

2022 - 2023

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In Lecture 2...

- Dynamic Array
- Amortized complexity analysis
- Iterator

Today

- Linked Lists
  - Singly linked lists
  - Doubly linked lists
  - Iterator for linked lists

Dynamic Array - review

The elements of a dynamic array occupy a contiguous memory block.

Advantages:

- accessing any element in $\Theta(1)$
- adding at the end in $\Theta(1)$ amortized and removing from the end in $\Theta(1)$ (amortized)

Disadvantages:

- adding and removing from the beginning of the array in $\Theta(n)$

Linked Lists

A **linked list** is a linear data structure consisting of nodes, the order of the elements being determined not by indexes, but by pointers stored in each node.

Each node contains, besides the data, a pointer to the next node (and possibly a pointer to the previous node).

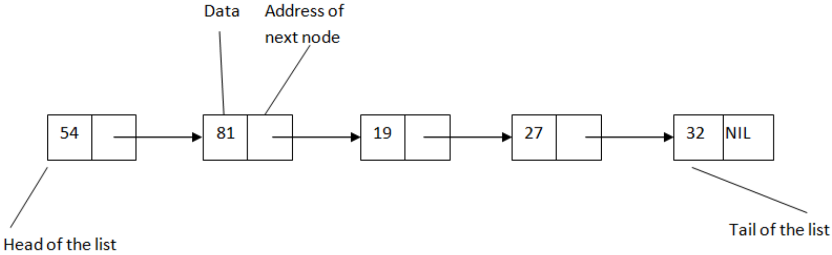
Linked Lists

The nodes of a linked list are not necessarily adjacent in memory. This is why we need to keep a pointer to the next node in each node.

We can directly access only the first node (and maybe the last one) of the list. For the other nodes and, therefore, elements, the access is sequential.

Linked Lists

Example of a linked list with 5 nodes:



This is actually a **singly** linked list (SLL).

Singly Linked Lists - SLL

Each SLL node contains the data and the pointer to the next node.

The first node of the list is called the **head** of the list and the last node is called the **tail** of the list.

The tail of the list contains the special value *NIL* as the pointer to the next node (which does not exist).

If the head of the linked list is *NIL*, then the list is empty.

Singly Linked Lists - Representation

- To represent a SLL we need two structures: one for the node and one for the list itself.

SLL's node representation:

SLLNode:
  info: TElem *//the actual information*
  next: ↑ SLLNode *//pointer to the next node*

SLL representation:

SLL:
  head: ↑ SLLNode *//address of the first node*

Usually, for a SLL, we only memorize the pointer to the head.

SLL - Operations

Possible operations for a singly linked list:

- searching for an element with a given value
- adding an element (at the beginning, to the end, at a given position)
- deleting an element (from the beginning, from the end, from a given position, with a given value)
- getting the element from a given position

These are *possible* operations; usually we need only part of them (depending on the container that we implement).

SLL - Search

Searching a given element into a SLL:

function search (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: returns a pointer to a node which contains elem as info,
//or NIL if sll does not contain the given element
    current $\leftarrow$ sll.head
    while current $\neq$ NIL **and** [current].info $\neq$ elem **execute**
        current $\leftarrow$ [current].next
    end-while
    search $\leftarrow$ current
end-function

What is the time complexity?

SLL - Search

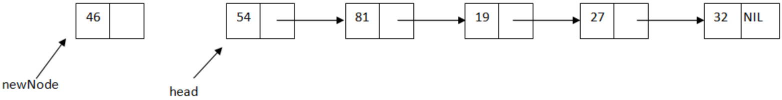
Searching a given element into a SLL:

function search (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: returns a pointer to a node which contains elem as info,
//or NIL if sll does not contain the given element
    current $\leftarrow$ sll.head
    while current $\neq$ NIL **and** [current].info $\neq$ elem **execute**
        current $\leftarrow$ [current].next
    end-while
    search $\leftarrow$ current
end-function

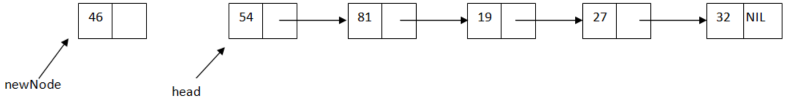
What is the time complexity?

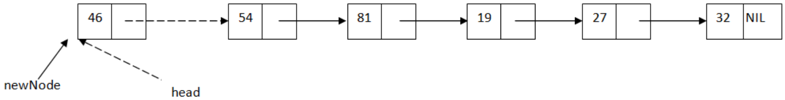
✓ $O(n)$

SLL - Insert at the beginning



SLL - Insert at the beginning





SLL - Insert at the beginning

Inserting at the beginning of a SLL:

subalgorithm insertFirst (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: elem will be inserted at the beginning of sll
    newNode ← allocate() *//allocate a new SLLNode*
    [newNode].info ← elem
    [newNode].next ← sll.head
    sll.head ← newNode
end-subalgorithm

What is the time complexity?

SLL - Insert at the beginning

Inserting at the beginning of a SLL:

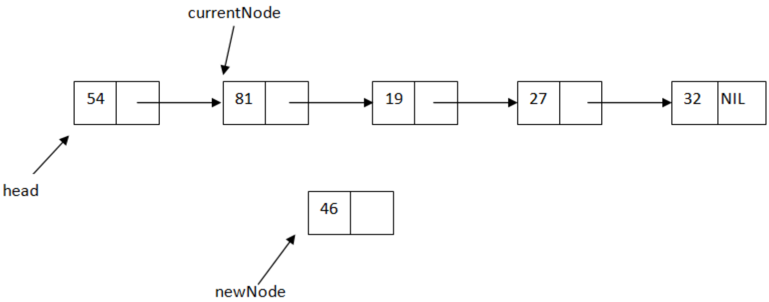
subalgorithm insertFirst (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: elem will be inserted at the beginning of sll
    newNode $\leftarrow$ allocate() *//allocate a new SLLNode*
    [newNode].info $\leftarrow$ elem
    [newNode].next $\leftarrow$ sll.head
    sll.head $\leftarrow$ newNode
end-subalgorithm

What is the time complexity?

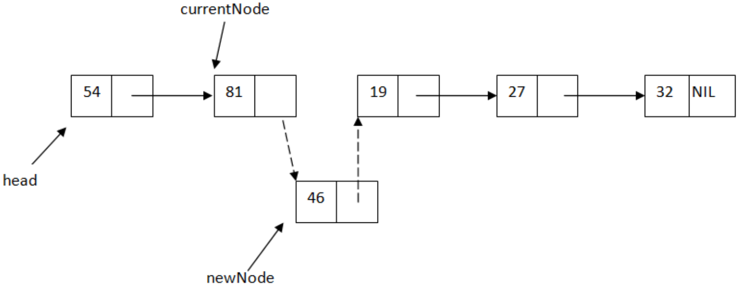
✓ $\Theta(1)$

SLL - Insert after a node

Suppose that we have the pointer to a node from a SLL and we want to insert a new element after that node.



SLL - Insert after a node



SLL - Insert after a node

Inserting after a node in a SLL:

subalgorithm insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
    newNode ← allocate() *//allocate a new SLLNode*
    [newNode].info ← elem
    [newNode].next ←[currentNode].next
    [currentNode].next ← newNode
end-subalgorithm

What is the time complexity?

SLL - Insert after a node

Inserting after a node in a SLL:

subalgorithm insertAfter(sll, currentNode, elem) **is:**
//pre: sll is a SLL; currentNode is an SLLNode from sll;
//elem is a TElem
//post: a node with elem will be inserted after node currentNode
    newNode ← allocate() *//allocate a new SLLNode*
    [newNode].info ← elem
    [newNode].next ←[currentNode].next
    [currentNode].next ← newNode
end-subalgorithm

What is the time complexity?

✓ $\Theta(1)$

Insert before a node

How can you insert an element in front of a given node in a SLL?

SLL - Insert at a position

How can we insert an element to a given integer position in a SLL?

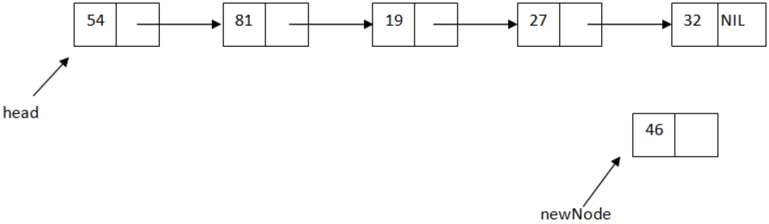
SLL - Insert at a position

How can we insert an element to a given integer position in a SLL?

We first need to find the node at the position *after* which we need to insert the element.

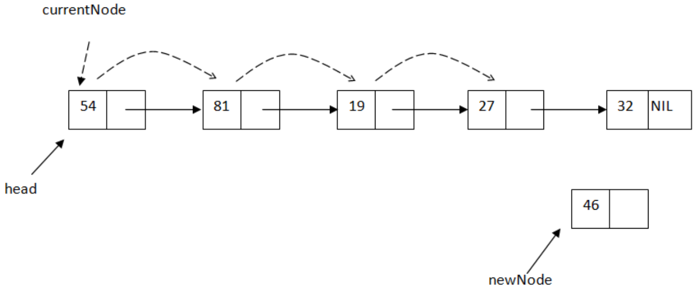
SLL - Insert at a position

Assume that we want to insert 46 at position 5.



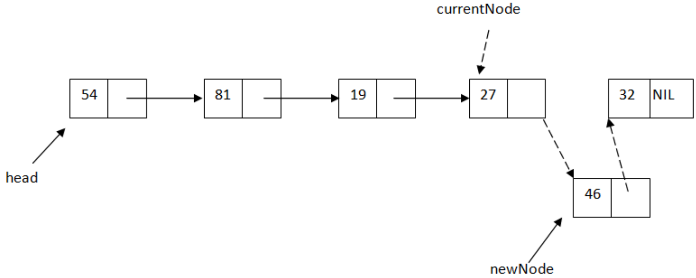
SLL - Insert at a position

We need the 4^{th} node (to insert the new node after it), but we have no direct access to it. So we have to traverse the list to it using and auxiliary pointer (*currentNode*).



SLL - Insert at a position

Now, we insert the new node after the node pointed by *currentNode*



SLL - Insert at a position

Inserting at a given position into a SLL:

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 in sll
    if pos $<$ 1 **then**
        @error, invalid position
    else if pos = 1 **then** *//we want to insert at the beginning*
        newNode $\leftarrow$ allocate() *//allocate a new SLLNode*
        [newNode].info $\leftarrow$ elem
        [newNode].next $\leftarrow$sll.head
        sll.head $\leftarrow$ newNode
    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...

Inserting at a given position into a SLL:

if currentNode $\neq$ NIL **then**
newNode $\leftarrow$ allocate() *//allocate a new SLLNode*
[newNode].info $\leftarrow$ elem
[newNode].next $\leftarrow$ [currentNode].next
[currentNode].next $\leftarrow$ newNode
else
@error, invalid position
end-if
end-if
end-subalgorithm

What is the time complexity?

Inserting at a given position into a SLL:

if currentNode $\neq$ NIL **then**
newNode $\leftarrow$ allocate() *//allocate a new SLLNode*
[newNode].info $\leftarrow$ elem
[newNode].next $\leftarrow$ [currentNode].next
[currentNode].next $\leftarrow$ newNode
else
@error, invalid position
end-if
end-if
end-subalgorithm

What is the time complexity?

✓ $O(n)$

Get element from a given position

How can we access an element at a position p from a SLL?

Get element from a given position

How can we access an element at a position *p* from a SLL?

Since we only have direct access to the head of the list, we have to go through the list, node-by-node, until we get to the p^{th} node.

The process is similar to the first part of the *insertPosition* subalgorithm.

SLL - Obtain the element at a given position

Obtaining the element at a given position in a SLL:

function getElement(sll, pos) **is:**
//pre: sll is a SLL; pos is an integer number
//post: getElement is a TElem representing the element at position pos in 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**
            getElement $\leftarrow$ [currentNode].info
        else:
            @error, invalid position
        end-if
    end-if
end-function

SLL - Delete a given element

? How can we delete an element from a SLL given either by its value or its position?

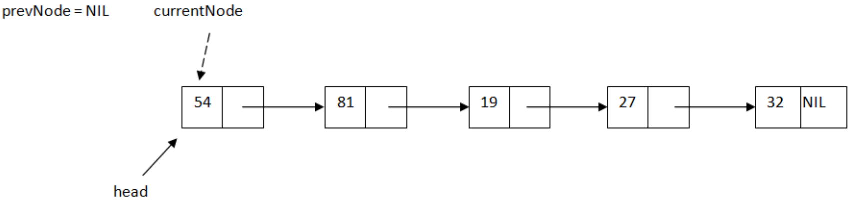
SLL - Delete a given element

How can we delete an element from a SLL given either by its value or its position?

We need to find the node *before* the one we want to delete. We can do this by walking through the list using two pointers, *currentNode* and *prevNode* (pointing to the previous node), until *currentNode* points to the node to be deleted.

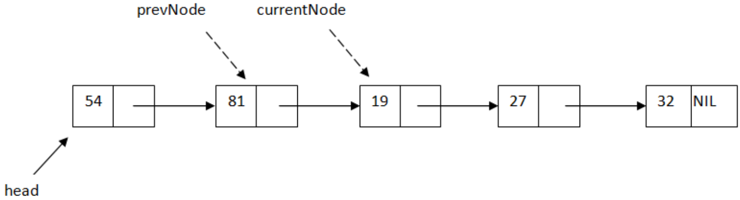
SLL - Delete a given element

Suppose we want to delete the node with information 19.



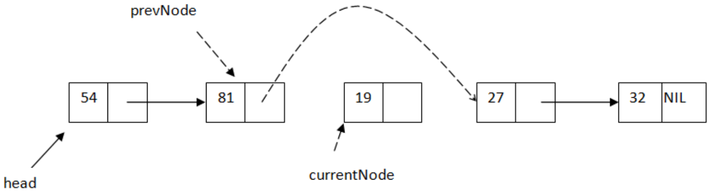
SLL - Delete a given element

Traverse the list with the two pointers until *currentNode* points to the node we want to delete.



SLL - Delete a given element

Delete the node pointed by *currentNode* by *jumping over it*.



SLL - Delete a given element

Deleting a given element from a SLL:

```
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 currentNode ≠ NIL AND prevNode = NIL then //we delete the head
    sll.head ← [sll.head].next
  else if currentNode ≠ NIL then
    [prevNode].next ← [currentNode].next
  end-if
  deleteElement ← currentNode
end-function
```

SLL - Delete a given element

What is the time complexity of *deleteElement*?

SLL - Delete a given element

What is the time complexity of *deleteElement*?

✓ $O(n)$

How can we define an iterator for a SLL? What would be the type of the iterator’s cursor?

SLL - Iterator

How can we define an iterator for a SLL? What would be the type of the iterator's cursor?

The iterator's cursor is a pointer to the current node in the list.

SLL - Iterator

SLL Iterator's representation:

SLLIterator:

list: SLL

currentElement: ↑ SLLNode

SLL - Iterator - init operation

The constructor of an iterator over a SLL:

subalgorithm init(i, sll) **is:**
//pre: sll is a SLL
//post: i is a SLLIterator over sll
  i.sll $\leftarrow$ sll
  i.currentElement $\leftarrow$ sll.head
end-subalgorithm

What is the time complexity?

SLL - Iterator - init operation

The constructor of an iterator over a SLL:

subalgorithm init(i, sll) **is:**
//pre: sll is a SLL
//post: i is a SLLIterator over sll
    i.sll $\leftarrow$ sll
    i.currentElement $\leftarrow$ sll.head
end-subalgorithm

What is the time complexity?

✓ $\Theta(1)$

SLL - Iterator - getCurrent operation

Returning the current element referred by a SLL Iterator:

function getCurrent(i) **is:**
//pre: i is a SLLIterator, i is valid
//post: getCurrent $\leftarrow$ e, e is TElem, the current element referred by i
//throws: exception if i is not valid
    if i.currentElement = NIL **then**
        @throw an exception
    end-if
    getCurrent $\leftarrow$ [i.currentElement].info
end-function

What is the time complexity?

SLL - Iterator - getCurrent operation

Returning the current element referred by a SLL Iterator:

function getCurrent(i) **is:**
//pre: i is a SLLIterator, i is valid
//post: getCurrent $\leftarrow$ e, e is TElem, the current element referred by i
//throws: exception if i is not valid
    if i.currentElement = NIL **then**
        @throw an exception
    end-if
    getCurrent $\leftarrow$ [i.currentElement].info
end-function

What is the time complexity?

✓ $\Theta(1)$

The next operation for a SLL Iterator:

subalgorithm next(i) **is:**
//pre: i is a SLLIterator, i is valid
//post: i' is a SLLIterator, the cursor of i' refers to the next element
//throws: exception if i is not valid
    if i.currentElement = NIL **then**
        @throw an exception
    end-if
    i.currentElement ← [i.currentElement].next
end-subalgorithm

What is the time complexity?

SLL - Iterator - next operation

The next operation for a SLL Iterator:

subalgorithm next(i) **is:**
//pre: i is a SLLIterator, i is valid
//post: i' is a SLLIterator, the cursor of i' refers to the next element
//throws: exception if i is not valid
    if i.currentElement = NIL **then**
        @throw an exception
    end-if
    i.currentElement $\leftarrow$ [i.currentElement].next
end-subalgorithm

What is the time complexity?

- $\Theta(1)$

SLL - Iterator - valid operation

The *valid* function of a SLL Iterator:

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

What is the time complexity?

SLL - Iterator - valid operation

The *valid* function of a SLL Iterator:

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

What is the time complexity?

✓ $\Theta(1)$

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?

Doubly Linked Lists - DLL

A **doubly linked list** is a linked list in which each node not only contains a pointer to the next node, but also a pointer to the previous one.

The first node has *NIL* as the pointer to the previous node (just like the last element has NIL as the pointer to the next node).

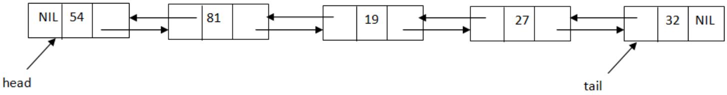
Singly vs. doubly linked list

There are trade-offs when choosing between SLL and DLL:

- SLL uses less memory, but we don’s have direct access to the previous elements.
- DLL can be easily traversed backwards, but it uses more memory.

Example of a Doubly Linked List

Example of a **doubly linked list** with 5 nodes:



Doubly Linked List - Representation

- To represent a DLL we need two structures: on for the node and one for the list itself.

A DLL's node representation:

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

Representation of a DLL:

DLL:
    head: ↑ DLLNode
    tail: ↑ DLLNode

DLL - Creating an empty list

- An empty list is one which has no nodes $\Rightarrow$ the address of the first node (and the address of the last node) is NIL

When we add or remove or search, we know that the list is empty if its head is NIL.

Creating an empty DLL:

subalgorithm init(dll) **is**:
//pre: true
//post: dll is a DLL
  dll.head $\leftarrow$ NIL
  dll.tail $\leftarrow$ NIL
end-subalgorithm

What is the time complexity?

DLL - Creating an empty list

- An empty list is one which has no nodes $\Rightarrow$ the address of the first node (and the address of the last node) is NIL

When we add or remove or search, we know that the list is empty if its head is NIL.

Creating an empty DLL:

subalgorithm init(dll) **is**:
//pre: true
//post: dll is a DLL
    dll.head $\leftarrow$ NIL
    dll.tail $\leftarrow$ NIL
end-subalgorithm

What is the time complexity?

$\Theta(1)$

DLL - Operations

Possible operations for a doubly linked list:

- searching for a given element
- adding an element (at the beginning, to the end, at a given position)
- deleting an element (from the beginning, from the end, from a given position, with a given value)
- getting the element from a given position

Some of the operations have the exact same implementation as for a SLL (e.g. *search*, *getElement*), but in general we need to modify more links and to pay attention to the *tail*.

DLL - Adding at the end

Adding a new element to the end of a DLL is simple. Because we have the *tail* of the list, we do not have to walk through all the elements (like we have to do in case of a SLL).

Adding an element at the end of a DLL:

```
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
    else
        [dll.tail].next ← newNode
    end-if
    dll.tail ← newNode
end-subalgorithm
```

What is the time complexity?

Adding an element at the end of a DLL:

```
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
    else
        [dll.tail].next ← newNode
    end-if
    dll.tail ← newNode
end-subalgorithm
```

What is the time complexity?

✓ $\Theta(1)$

DLL - Insert at a given position

How can we insert an element to a given integer position into a DLL?

DLL - Insert at a given position

How can we insert an element to a given integer position into a DLL?

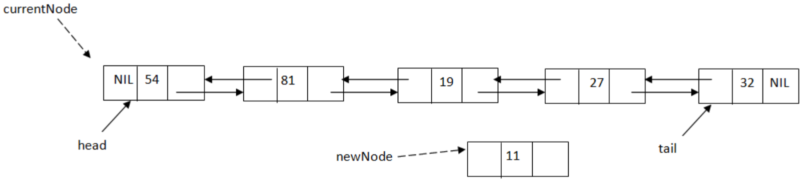
The principle is the same as in the case of a SLL excepting that:

- We need to also (re)set links to previous node
- We may need to reset the tail of the list

We can stop either before of after the position we want to insert the element at

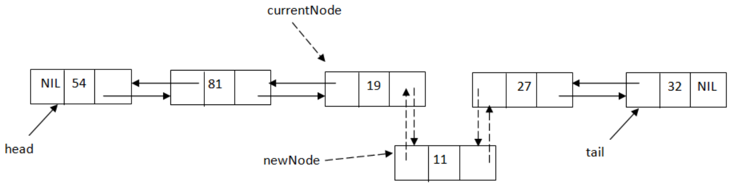
DLL - Insert on position

Let's insert value 11 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

Inserting at a given position into a DLL:

```
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 ← dll.head
        currentPos ← 1
        while currentNode ≠ NIL and currentPos < pos - 1 execute
            currentNode ← [currentNode].next
            currentPos ← currentPos + 1
        end-while
//continued on the next slide...
```

DLL - Insert at position

Inserting at a given position into a DLL:

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

What is the time complexity?

DLL - Insert at position

Inserting at a given position into a DLL:

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

What is the time complexity?

✓ $O(n)$

DLL - Delete a given element

If we want to delete a given element, we first have to find the node containing it.

Special cases:

- the element is not in list (includes the case of an empty list)
- remove head (which may be tail as well)
- remove tail

Deleting 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 //remove the first node
            if currentNode = dll.tail then //which is the last one as well
                dll.head ← NIL
                dll.tail ← NIL
            else //list has more than 1 element, remove first
                dll.head ← [dll.head].next
                [dll.head].prev ← NIL
            end-if
//continued on the next slide...
```

DLL - Delete a given element

Deleting a given element:

```
        else if currentNode = dll.tail then
            dll.tail ← [dll.tail].prev
            [dll.tail].next ← NIL
        else
            [[currentNode].next].prev ← [currentNode].prev
            [[currentNode].prev].next ← [currentNode].next
            @set links of deletedNode to NIL to separate it from the
nodes of the list
        end-if
    end-if
    deleteElement ← deletedNode
end-function
```

What is the time complexity?

DLL - Delete a given element

Deleting a given element:

```
        else if currentNode = dll.tail then
            dll.tail ← [dll.tail].prev
            [dll.tail].next ← NIL
        else
            [[currentNode].next].prev ← [currentNode].prev
            [[currentNode].prev].next ← [currentNode].next
            @set links of deletedNode to NIL to separate it from the
nodes of the list
        end-if
    end-if
    deleteElement ← deletedNode
end-function
```

What is the time complexity?

✓ $O(n)$

Iterating through all the elements of a linked list

Similar to the DynamicArray, if we want to go through all the elements of a (singly or doubly) linked list we have two options:

- Use an iterator
- Use a *for* loop and the *getElement* function

What is the complexity of the two approaches?

Traversing a SLL with Iterator

Printing all the elements of an SLL using the iterator:

subalgorithm printSLLWithIterator(sll) **is:**
    //pre: sll is a SLL
    init(i, sll)
    while valid(i) **execute**
        //get the current element referred by the iterator
        elem ← getCurrent(i)
        print elem
        //go to the next element
        next(i)
    end-while
end-subalgorithm

What is the complexity of *printSLLWithIterator*?

Traversing a SLL using *getElement*

Printing all the elements of a ALL using *getElement*:

subalgorithm printSLLUsingGetCurrent(sll) **is:**
//pre: sll is a SLL
    for i ← 1, size(sll) **execute**
        elem ← getElement(sll, i)
        print elem
    end-for
end-subalgorithm

What is the complexity of *printSLLUsingGetCurrent*?

Dynamic Array vs. Linked Lists

Advantages of Linked Lists:

- No memory used for non-existing elements
- Constant time complexity for the operations at the beginning
- Elements are never moved / shifted
- Overflow can never occur unless the memory is actually full.

Disadvantages of Linked Lists:

- No direct / random access to an element at a given position
- Extra space is used up by the addresses stored in the nodes

Containers represented using linked lists

Linked lists are used for representing the following containers:

- ADT List
    - LinkedList in Java, forward_list (SLL) and list (DLL) in C++ STL
- ADT Stack
    - Stack in C++
- ADT Queue
    - Queue in C++
- ADT Bag
- ADT Set

Linked lists - Applications

Applications of linked lists:

Compilers

- For symbol table management

Web browsers

- To keep track of the visited pages

Music player

- A music player can use a linked list to allow switching to the next/previous song

Text editors

- Undo and Redo mechanism in Excel, Notepad, WordPad, etc.

Multi-player games

- To keep the track of turns

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you