DATA STRUCTURES AND ALGORITHMS

LECTURE 5

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

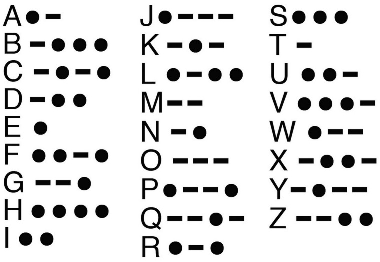
In Lecture 4...

- Containers
  - ADT Matrix
  - ADT Stack
  - ADT Queue
  - ADT PriorityQueue
  - ADT Map
  - ADT SortedMap

Today

- Containers
- Linked Lists

- Morse Code, is a code which assigns to every letter a sequence of dots and dashes.



https://medium.com/@timboucher/learning-morse-code-35e1f4d285f6

- Given a list of words, find the largest subset of the words, for which the Morse representation is the same.

- For example, if the words are *cat*, *ca*, *nna*, *abc* and *nnet*, their Morse code representation is:
  - *cat* -.-..–
  - *ca* -.-..-
  - *nna* -.-..-
  - *abc* .–...-.-.
  - *nnet* -.-..-
- What would be the characteristics of the container used for this problem?

- For example, if the words are *cat*, *ca*, *nna*, *abc* and *nnet*, their Morse code representation is:
  - *cat* -.-..–
  - *ca* -.-..-
  - *nna* -.-..-
  - *abc* .–...-.-.
  - *nnet* -.-..-
- What would be the characteristics of the container used for this problem?
  - We could solve the problem if we used the Morse representation of a word as a key and the corresponding word as a value
  - One key can have multiple values
  - Order of the elements is not important
- The container in which we store key - value pairs, and where a key can have multiple associated values, is called a **ADT MultiMap**.

ADT MultiMap

- Domain of ADT MultiMap:

$\mathcal{MM} = \{mm | mm$ is a Multimap with TKey, TValue pairs$\}$

ADT MultiMap - Interface I

- init (mm)
  - **descr:** creates a new empty multimap
  - **pre:** true
  - **post:** $mm \in \mathcal{MM}$, mm is an empty multimap

ADT MultiMap - Interface II

- destroy(mm)
  - **descr:** destroys a multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** the multimap was destroyed

ADT MultiMap - Interface III

- add(mm, k, v)
  - **descr:** add a new pair to the multimap
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$, $v - TValue$
  - **post:** $mm' \in \mathcal{MM}$, $mm' = mm \cup < k, v >$

ADT MultiMap - Interface IV

- remove(mm, k, v)
  - **descr:** removes a key value pair from the multimap
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$, $v - TValue$
  - **post:** $remove \leftarrow$
    $$\begin{cases} true, & \text{if} < k, v > \in mm, mm' \in \mathcal{MM}, mm' = mm- < k, v > \\ false, & \text{otherwise} \end{cases}$$

ADT MultiMap - Interface V

- search(mm, k, l)
  - **descr:** returns a list with all the values associated to a key
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$
  - **post:** $l \in \mathcal{L}$, l is the list of values associated to the key k. If k is not in the multimap, l is the empty list.

ADT MultiMap - Interface VI

- iterator(mm, it)
  - **descr:** returns an iterator over the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $it \in \mathcal{I}$, *it* is an iterator over *mm*, the current element from *it* is the first pair from *mm*, or, *it* is invalid if *mm* is empty

  - **Obs:** the iterator for a MultiMap is similar to the iterator for other containers, but the *getCurrent* operation returns a $<$key, value$>$ pair.

ADT MultiMap - Interface VII

- size(mm)
  - **descr:** returns the number of pairs from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $size \leftarrow$ the number of pairs from mm

- Other possible operations:
- keys(mm, s)
  - **descr:** returns the set of all keys from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $s \in \mathcal{S}$, s is the set of all keys from mm

- values(mm, b)
  - **descr:** returns the bag of all values from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $b \in \mathcal{B}$m b is a bag with all the values from mm

- pairs(mm, b)
  - **descr:** returns the bag of all pairs from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $b \in \mathcal{B}$, b is a bag with all the pairs from mm

- We can have a MultiMap where we can define an order (a relation) on the set of possible keys. However, if a key has multiple values, they can be in any order (we order the keys only, not the values) $\Rightarrow$ **ADT SortedMultiMap**
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted MultiMap, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSet and SortedBag.

- There are several data structures that can be used to implement an ADT MultiMap (or ADT SortedMultiMap), the dynamic array being one of them (others will be discussed later):

- Regardless of the data structure used, there are two options to represent a MultiMap (sorted or not):
  - Store individual $<$ key, value $>$ pairs. If a key has multiple values, there will be multiple pairs containing this key. (R1)
  - Store unique keys and for each key store a *list* of associated values. (R2)

- For the example with the Morse code, we would have:

-.-..-- cat	-.-..- ca	-.-..- nna	.--...-.-. abc	-.-..- nnet

- Key is written with red and the value with black.
- Every element is one key - value pair.

ADT MultiMap - R2

- For the example with the Morse code, we would have:

-.-..-- [cat]	-.-..- [ca, nna, nnet]	.--...-.-. [abc]

- Key is written with red and the value with black.
- Every element is one key together with all the values belonging to it. The *list of values* can be another dynamic array, or a linked list, or any other data structure.

- A *list* can be seen as a sequence of elements of the same type, $< l_1, l_2, ..., l_n >$, where there is an order of the elements, and each element has a *position* inside the list.
- In a list, the order of the elements is important (positions are important).
- The number of elements from a list is called the length of the list. A list without elements is called *empty*.

ADT List

- A List is a container which is either *empty* or
  - it has a unique *first* element
  - it has a unique *last* element
  - for every element (except for the last) there is a unique *successor* element
  - for every element (except for the first) there is a unique *predecessor* element
- In a list, we can insert elements (using positions), remove elements (using positions), we can access the successor and predecessor of an element from a given position, we can access an element from a position.

ADT List - Positions

- Every element from a list has a unique position in the list:
  - positions are relative to the list (but important for the list)
  - the position of an element:
    - identifies the element from the list
    - determines the position of the successor and predecessor element (if they exist).

ADT List - Positions

- Position of an element can be seen in different ways:
  - as the *rank* of the element in the list (first, second, third, etc.)
    - similarly to an array, the position of an element is actually its index
  - as a *reference* to the memory location where the element is stored.
    - for example a pointer to the memory location
- For a general treatment, we will consider in the following the *position* of an element in an abstract manner, and we will consider that positions are of type *TPosition*

ADT - List - Positions

- A position p will be considered *valid* if it denotes the position of an actual element from the list:
  - if p is a pointer to a memory location, p is valid if it is the address of an element from a list (not NIL or some other address that is not the address of any element)
  - if p is the rank of the element from the list, p is valid if it is between 1 and the number of elements.
- For an invalid position we will use the following notation: $\perp$

- Domain of the ADT List:

  $\mathcal{L} = \{l |$l is a list with elements of type TElem, each having a unique position in l of type TPosition$\}$

ADT List II

- init(l)
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:** $l \in \mathcal{L}$, l is an empty list

- first(l)
  - **descr:** returns the TPosition of the first element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $first \leftarrow p \in TPosition$

$$p = \begin{cases} \text{the position of the first element from l} & \text{if l} \neq \emptyset \\ \perp & otherwise \end{cases}$$

ADT List IV

- last(l)
  - **descr:** returns the TPosition of the last element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $last \leftarrow p \in TPosition$

$$p = \begin{cases} \text{the position of the last element from l} & \text{if l} \neq \emptyset \\ \perp & otherwise \end{cases}$$

ADT List V

- valid(l, p)
  - **descr:** checks whether a TPosition is valid in a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$
  - **post:** $valid \leftarrow \begin{cases} true & \text{if p is a valid position in l} \\ false & otherwise \end{cases}$

ADT List VI

- next(l, p)
  - **descr:** goes to the next TPosition from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
  - **post:**

$$next \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the next element after p} & \text{if p is not the last position} \\ \perp & \textit{otherwise} \end{cases}$$

  - **throws:** exception if p is not valid

ADT List VII

- previous(l, p)
  - **descr:** goes to the previous TPosition from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition, valid(l, p)$
  - **post:**

$$previous \leftarrow q \in TPosition$$

$$q = \begin{cases} \text{the position of the element before p} & \text{if p is not the first position} \\ \perp & \textit{otherwise} \end{cases}$$

  - **throws:** exception if p is not valid

ADT List VIII

- getElement(l, p)
  - **descr:** returns the element from a given TPosition
  - **pre:** $l \in \mathcal{L}, p \in TPosition$, $valid(l, p)$
  - **post:** $getElement \leftarrow e$, $e \in TElem$, e $=$ the element from position p from l
  - **throws:** exception if p is not valid

- position(l, e)
  - **descr:** returns the TPosition of an element
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$position \leftarrow p \in TPosition$$

$$\text{p} = \begin{cases} \text{the first position of element e from l} & \text{if } e \in l \\ \perp & otherwise \end{cases}$$

- setElement(l, p, e)
  - **descr:** replaces an element from a TPosition with another
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem$, $valid(l, p)$
  - **post:** $l' \in \mathcal{L}$, the element from position p from l' is e, $setElement \leftarrow el$, $el \in TElem$, el is the element from position p from l (returns the previous value from the position)
  - **throws:** exception if p is not valid

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

- addToEnd(l, e)
  - **descr:**adds a new element to the end of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT List XIII

- addBeforePosition(l, p, e)
    - **descr:** inserts a new element before a given position
    - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem,\ valid(l, p)$
    - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l before the position p
    - **throws:** exception if p is not valid

ADT List XIV

- addAfterPosition(l, p, e)
  - **descr:** inserts a new element after a given position
  - **pre:** $l \in \mathcal{L}, p \in TPosition, e \in TElem,\ valid(l, p)$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l after the position p
  - **throws:** exception if p is not valid

- remove(l, p)
  - **descr:** removes an element from a given position from a list
  - **pre:** $l \in \mathcal{L}, p \in TPosition$, $valid(l, p)$
  - **post:** $remove \leftarrow e$, $e \in TElem$, e is the element from position p from l, $l' \in \mathcal{L}$, l' $=$ l - e.
  - **throws:** exception if p is not valid

ADT List XVI

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

- search(l, e)
  - **descr:** searches for an element in the list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$search \leftarrow \begin{cases} true & \text{if } e \in l \\ false & otherwise \end{cases}$$

- isEmpty(l)
  - **descr:** checks if a list is empty
  - **pre:** $l \in \mathcal{L}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true & \text{if } l = \emptyset \\ false & otherwise \end{cases}$$

ADT List XIX

- size(l)
  - **descr:** returns the number of elements from a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $size \leftarrow$ the number of elements from l

- destroy(l)
  - **descr:** destroys a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** l was destroyed

ADT List XXI

- iterator(l, it)
  - **descr:** returns an iterator for a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $it \in \mathcal{I}$, *it* is an iterator over *l*, the current element from *it* is the first element from *l*, or, if *l* is empty, *it* is invalid

TPosition - Integer

- In Python and Java, TPosition is represented by an index.
- We can add and remove using index and we can access elements using their index (but we have iterator as well for the List).
- For example (Python):
  insert (int index, E object)
  index (E object)
  - Returns an integer value, position of the element (or exception if *object* is not in the list)
- For example (Java):

  void add(int index, E element)
  E get(int index)
  E remove(int index)
  - Returns the removed element

ADT IndexedList

- If we consider that TPosition is an Integer value (similar to Python and Java), we can have an *IndexedList*
- In case of an *IndexedList* the operations that work with a position take as parameter integer numbers representing these positions
- There are less operations in the interface of the *IndexedList*
  - Operations *first*, *last*, *next*, *previous*, *valid* do not exist

ADT IndexedList I

- init(l)
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:** $l \in \mathcal{L}$, l is an empty list

ADT IndexedList II

- getElement(l, i)
    - **descr:** returns the element from a given position
    - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}$, i is a valid position
    - **post:** $getElement \leftarrow e$, $e \in TElem$, e $=$ the element from position i from l
    - **throws:** exception if i is not valid

- position(l, e)
  - **descr:** returns the position of an element
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$position \leftarrow i \in \mathcal{N}$$

$$\text{i} = \begin{cases} \text{the first position of element e from l} & \text{if } e \in l \\ -1 & otherwise \end{cases}$$

ADT IndexedList IV

- setElement(l, i, e)
  - **descr:** replaces an element from a position with another
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElem$, i is a valid position
  - **post:** $l' \in \mathcal{L}$, the element from position i from l' is e, $setElement \leftarrow el$, $el \in TElem$, el is the element from position i from l (returns the previous value from the position)
  - **throws:** exception if i is not valid

ADT IndexedList V

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT IndexedList VI

- addToEnd(l, e)
  - **descr:**adds a new element to the end of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT IndexedList VII

- addToPosition(l, i, e)
  - **descr:** inserts a new element at a given position (it is the same as *addBeforePosition*)
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}, e \in TElem$, i is a valid position (size + 1 is valid for adding an element)
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l at the position i
  - **throws:** exception if i is not valid

ADT IndexedList VIII

- remove(l, i)
  - **descr:** removes an element from a given position from a list
  - **pre:** $l \in \mathcal{L}, i \in \mathcal{N}$, i is a valid position
  - **post:** $remove \leftarrow e$, $e \in TElem$, e is the element from position i from l, $l' \in \mathcal{L}$, l' = l - e.
  - **throws:** exception if i is not valid

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

ADT IndexedList X

- search(l, e)
  - **descr:** searches for an element in the list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$search \leftarrow \begin{cases} true & \text{if } e \in l \\ false & otherwise \end{cases}$$

ADT IndexedList XI

- isEmpty(l)
  - **descr:** checks if a list is empty
  - **pre:** $l \in \mathcal{L}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true & \text{if } l = \emptyset \\ false & otherwise \end{cases}$$

ADT IndexedList XII

- size(l)
  - **descr:** returns the number of elements from a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $size \leftarrow$ the number of elements from l

ADT IndexedList XIII

- destroy(l)
  - **descr:** destroys a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** l was destroyed

- iterator(l, it)
  - **descr:** returns an iterator for a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $it \in \mathcal{I}$, *it* is an iterator over *l*, the current element from *it* is the first element from *l*, or, if *l* is empty, *it* is invalid

TPosition - Iterator

- In STL (C++), TPosition is represented by an iterator.
- For example - vector:

  iterator insert(iterator position, const value_type& val)
  - Returns an iterator which points to the newly inserted element

  iterator erase (iterator position);
  - Returns an iterator which points to the element after the removed one
- For example - list:

  iterator insert(iterator position, const value_type& val)
  iterator erase (iterator position);

ADT IteratedList

- If we consider that TPosition is an Iterator (similar to C++) we can have an *IteratedList*.
- In case of an *IteratedList* the operations that take as parameter a position use an Iterator (and the position is the current element from the Iterator)
- Operations *valid*, *next*, *previous* no longer exist in the interface of the List (they are operations for the Iterator).

ADT IteratedList I

- init(l)
  - **descr:** creates a new, empty list
  - **pre:** true
  - **post:** $l \in \mathcal{L}$, l is an empty list

- first(l)
  - **descr:** returns an Iterator set to the first element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $first \leftarrow it \in Iterator$

$$it = \begin{cases} \text{an iterator set to the first element} & \text{if l} \neq \emptyset \\ \text{an invalid iterator} & otherwise \end{cases}$$

ADT IteratedList III

- last(l)
  - **descr:** returns an Iterator set to the last element
  - **pre:** $l \in \mathcal{L}$
  - **post:** $last \leftarrow it \in Iterator$

$$\text{it} = \begin{cases} \text{an iterator set to the last element} & \text{if } \mathrm{l} \neq \emptyset \\ \text{an invalid iterator} & otherwise \end{cases}$$

ADT IteratedList IV

- getElement(l, it)
  - **descr:** returns the element from the position denoted by an Iterator
  - **pre:** $l \in \mathcal{L}, it \in Iterator, valid(it)$
  - **post:** $getElement \leftarrow e$, $e \in TElem$, e $=$ the element from l from the current position
  - **throws:** exception if *it* is not valid

ADT IteratedList V

- position(l, e)
  - **descr:** returns an iterator set to the first position of an element
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$position \leftarrow it \in Iterator$$

$$\text{it} = \begin{cases} \text{an iterator set to the first position of element e from l} & \text{if } e \in l \\ \text{an invalid iterator} & otherwise \end{cases}$$

ADT IteratedList VI

- setElement(l, it, e)
  - **descr:** replaces the element from the position denoted by an Iterator with another element
  - **pre:** $l \in \mathcal{L}, it \in Iterator, e \in TElem$, valid(it)
  - **post:** $l' \in \mathcal{L}$, the element from the position denoted by *it* from *l'* is e, $setElement \leftarrow el$, $el \in TElem$, *el* is the element from the current position from *it* from *l* (returns the previous value from the position)
  - **throws:** exception if *it* is not valid

- addToBeginning(l, e)
  - **descr:** adds a new element to the beginning of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the beginning of l

ADT IteratedList VIII

- addToEnd(l, e)
  - **descr:** inserts a new element at the end of a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added at the end of l

ADT IteratedList IX

- addToPosition(l, it, e)
  - **descr:** inserts a new element at a given position specified by the iterator (it is the same as *addAfterPosition*)
  - **pre:** $l \in \mathcal{L}, it \in Iterator, e \in TElem$, $valid(it)$
  - **post:** $l' \in \mathcal{L}$, l' is the result after the element e was added in l at the position specified by *it*
  - **throws:** exception if *it* is not valid

- remove(l, it)
  - **descr:** removes an element from a given position specfied by the iterator from a list
  - **pre:** $l \in \mathcal{L}, it \in Iterator, valid(it)$
  - **post:** $remove \leftarrow e$, $e \in TElem$, e is the element from the position from l denoted by it, $l' \in \mathcal{L}$, l' $=$ l - e.
  - **throws:** exception if *it* is not valid

ADT IteratedList XI

- remove(l, e)
  - **descr:** removes the first occurrence of a given element from a list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$remove \leftarrow \begin{cases} true & \text{if } e \in l \text{ and it was removed} \\ false & otherwise \end{cases}$$

ADT IteratedList XII

- search(l, e)
  - **descr:** searches for an element in the list
  - **pre:** $l \in \mathcal{L}, e \in TElem$
  - **post:**

$$search \leftarrow \begin{cases} true & \text{if } e \in l \\ false & otherwise \end{cases}$$

ADT IteratedList XIII

- isEmpty(l)
  - **descr:** checks if a list is empty
  - **pre:** $l \in \mathcal{L}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true & \text{if } l = \emptyset \\ false & otherwise \end{cases}$$

ADT IteratedList XIV

- size(l)
  - **descr:** returns the number of elements from a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** $size \leftarrow$ the number of elements from l

ADT IteratedList XV

- destroy(l)
  - **descr:** destroys a list
  - **pre:** $l \in \mathcal{L}$
  - **post:** l was destroyed

ADT SortedList

- We can define the ADT *SortedList*, in which the elements are memorized in an order given by a relation.

- You have below the list of operations for ADT *List*

  - init(l)
  - first(l)
  - last(l)
  - valid(l, p)
  - next(l, p)
  - previous(l, p)
  - getElement(l, p)
  - position(l, e)
  - setElement(l, p, e)
  - addToBeginning(l, e)
  - addToEnd(l, e)
  - addToPosition(l, p, e)
  - remove(l, p)
  - remove(l, e)
  - search(l, e)
  - isEmpty(l)
  - size(l)
  - destroy(l)
  - iterator(l, it)

- Which operations do no longer exist for a *SortedList*? What operations should be added? Should we change the parameters of some operations?

- The interface of the ADT *SortedList* is very similar to that of the ADT *List* with some exceptions:
  - The *init* function takes as parameter a relation that is going to be used to order the elements
  - We no longer have several *add* operations (*addToBeginning*, *addToEnd*, *addToPostion*), we have one single *add* operation, which takes as parameter only the element to be added (and adds it to the position where it should go based on the relation)
  - We no longer have a *setElement* operation (might violate ordering)

- We can consider *TPosition* in two different ways for a *SortedList* as well $\Rightarrow$ *SortedIndexedList* and *SortedIteratedList*

Dynamic Array - review

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.
- This gives us the main advantage of the array:
  - constant time access to any element from any position
  - constant time for operations (add, remove) at the end of the array

Dynamic Array - review

- The main idea of the (dynamic) array is that all the elements from the array are in one single consecutive memory location.
- This gives us the main advantage of the array:
  - constant time access to any element from any position
  - constant time for operations (add, remove) at the end of the array
- This gives us the main disadvantage of the array as well:
  - $\Theta(n)$ complexity for operations (add, remove) at the beginning of the array

Linked Lists

- A *linked list* is a linear data structure, where the order of the elements is determined not by indexes, but by a pointer which is placed in each element.

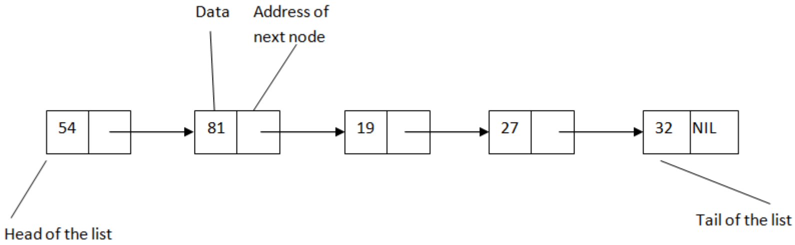
- A linked list is a structure that consists of *nodes* (sometimes called *links*) and each node contains, besides the data (that we store in the linked list), a pointer to the address of the next node (and possibly a pointer to the address of the previous node).

- The nodes of a linked list are not necessarily adjacent in the memory, this is why we need to keep the address of the successor in each node.

- Elements from a linked list are accessed based on the pointers stored in the nodes.
- We can directly access only the first element (and maybe the last one) of the list.

Linked Lists

- Example of a linked list with 5 nodes:



- The linked list from the previous slide is actually a *singly linked list - SLL*.
- In a SLL each node from the list contains the data and the address of the next node.
- The first node of the list is called *head* of the list and the last node is called *tail* of the list.
- The tail of the list contains the special value *NIL* as the address of the next node (which does not exist).
- If the head of the SLL is *NIL*, the list is considered empty.

Singly Linked Lists - Representation

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

SLLNode:
  info: TElem *//the actual information*
  next: $\uparrow$ SLLNode *//address of the next node*

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

<u>SLLNode:</u>
  info: TElem *//the actual information*
  next: $\uparrow$ SLLNode *//address of the next node*

<u>SLL:</u>
  head: $\uparrow$ SLLNode *//address of the first node*

- Usually, for a SLL, we only memorize the address of the head. However, there might be situations when we memorize the address of the tail as well (if the application requires it).

SLL - Operations

- Possible operations for a singly linked list:
  - search for an element with a given value
  - add an element (to the beginning, to the end, to a given position, after a given value)
  - delete an element (from the beginning, from the end, from a given position, with a given value)
  - get an element from a position

- These are *possible* operations; usually we need only part of them, depending on the container that we implement using a SLL.

SLL - Search

function search (sll, elem) **is:**
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL

SLL - Search

function search (sll, elem) **is:**
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
  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

- Complexity:

SLL - Search

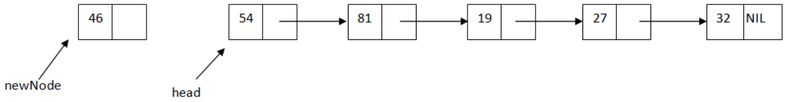
function search (sll, elem) **is:**
//pre: sll is a SLL - singly linked list; elem is a TElem
//post: returns the node which contains elem as info, or NIL
current $\leftarrow$ sll.head
while current $\neq$ NIL **and** [current].info $\neq$ elem **execute**
current $\leftarrow$ [current].next
end-while
search $\leftarrow$ current
end-function

- Complexity: $O(n)$ - we can find the element in the first node, or we may need to verify every node.

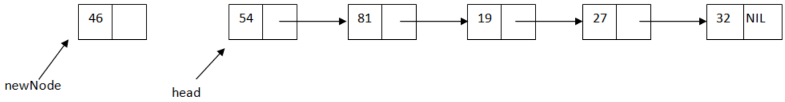
SLL - Walking through a linked list

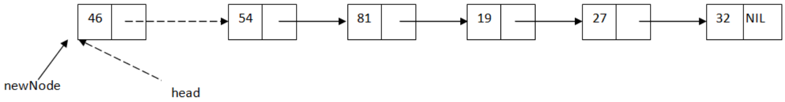
- In the *search* function we have seen how we can walk through the elements of a linked list:
  - we need an auxiliary node (called *current*), which starts at the head of the list
  - at each step, the value of the *current* node becomes the address of the successor node (through the $current \leftarrow [current].next$ instruction)
  - we stop when the current node becomes *NIL*

SLL - Insert at the beginning



SLL - Insert at the beginning





subalgorithm insertFirst (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: the element 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

- Complexity:

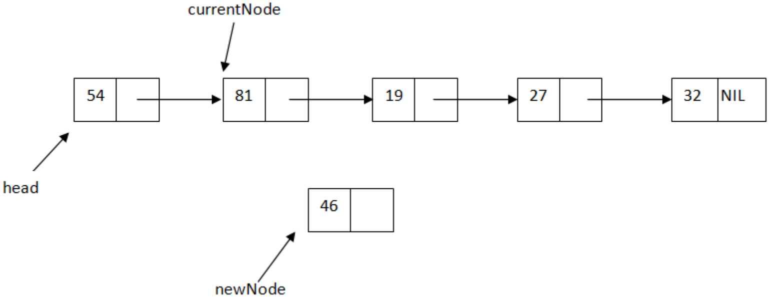
SLL - Insert at the beginning

subalgorithm insertFirst (sll, elem) **is:**
//pre: sll is a SLL; elem is a TElem
//post: the element 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

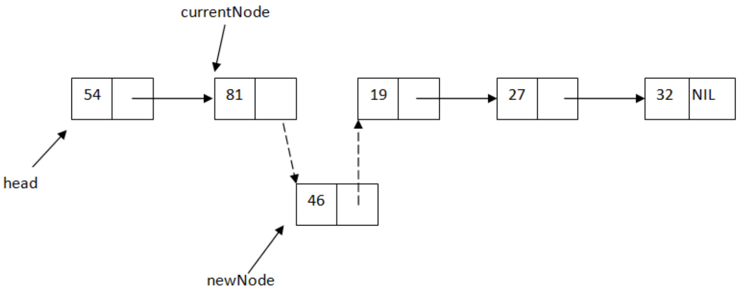
- Complexity: $\Theta(1)$

SLL - Insert after a node

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



SLL - Insert after a node



SLL - Insert after a node

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

- Complexity:

SLL - Insert after a node

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

- Complexity: $\Theta(1)$

Insert before a node

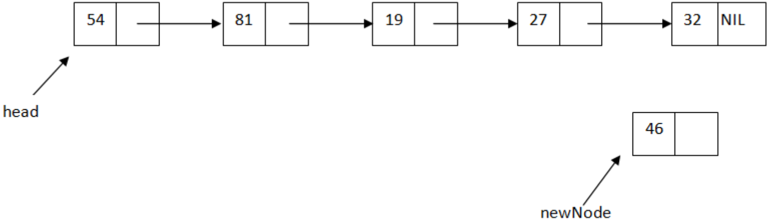
- Think about the following case: if you have a node, how can you insert an element in front of the node?

SLL - Insert at a position

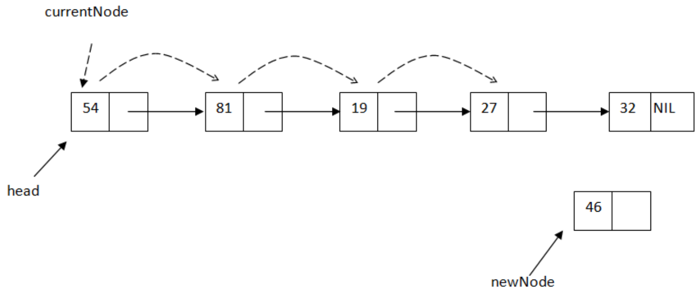
- We usually do not have the node after which we want to insert an element: we either know the position to which we want to insert, or know the element (not the node) after which we want to insert an element.

- Suppose we want to insert a new element at integer 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.

- We want to insert element 46 at position 5.

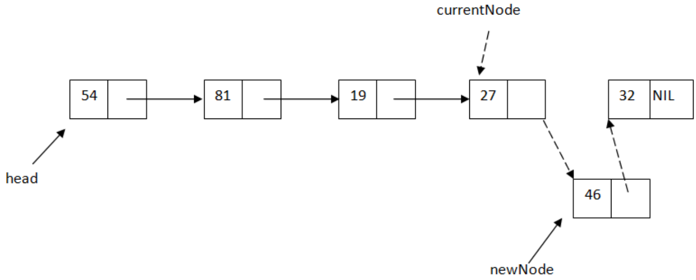


- We need the 4^{th} node (to insert element 46 after it), but we have direct access only to the first one, so we have to take an auxiliary node (*currentNode*) to get to the position.



SLL - Insert at a position

- Now we insert after node *currentNode*



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*
        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...

```
    if currentNode ≠ NIL then
        newNode ← allocate() //allocate a new SLLNode
        [newNode].info ← elem
        [newNode].next ← [currentNode].next
        [currentNode].next ← newNode
    else
        @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity:

```
    if currentNode ≠ NIL then
      newNode ← allocate() //allocate a new SLLNode
      [newNode].info ← elem
      [newNode].next ← [currentNode].next
      [currentNode].next ← newNode
    else
      @error, invalid position
    end-if
  end-if
end-subalgorithm
```

- Complexity: $O(n)$