ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Fundamental data structures - Arrays and dynamical lists-

Algorithms and data structures

- lectures -

Outline

- Arrays
- Operations with arrays
- Dynamical linked lists

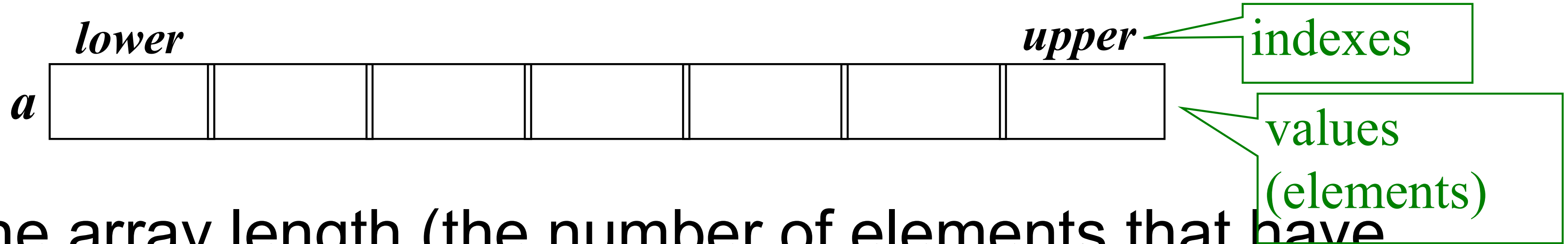
Arrays

- Consecutive set of memory locations
- Set of ordered pairs

  (index, value)
- For every index appearance, there is a corresponding value associated for that index

Arrays

- The indexes are unique and fixed for each value.
- The indexes are in range of ***lower*** limit to ***upper*** limit:



- The array length (the number of elements that have values) is a value that is given at the beginning when the array is formed

- Each array value can be accessed using its index, in $O(1)$ time.

IMPORTANT: The maximum index allowed in the array is always specified

Arrays

- Properties:
  - The arrays store several elements from the same type under the same name
  - The elements in the array can be accessed by an arbitrary order using the index
  - The array memory is predefined, there is no need for an additional memory space
  - The arrays are static data structures, their size is fixed and it can not be changed after their declaration

Arrays

- Functions (operations) that are performed for an array:
  - Traversal
  - Insert (add)
  - Delete (remove)
  - Search (find)
  - Sort
- Performance analysis for each of its operations

Traversal

- Traversing an array is the process of visiting each element once
- The traversal is necessary when:
  - The array elements are counted
  - The array elements are printed
  - A sum of array elements is computed
  - Etc ...

Insertion

- The insertion is a process of adding one or several new elements in the array
- The insertions of an element in an array can be done:
  - At the beginning of the array
  - At any index in the array
  - At the end of the array

Insertion

- Problem: for a given array *a*[*left*…*right*], to enter value *val* in *a*[ins]. If it is necessary, to shift the values in right to make space

(we suppose that *left* ≤ *ins* ≤ *right*.)

Insertion

- Animation:

1. copy a[ins…right–1] in a[ins+1…right].
2. copy val in a[ins].
3. end.

	left = 0	1	2	3	4	5	6 = *right*
a	The	fat	cat	sat	on	the	mouse

val fat   *ins* 1

Insertion analysis

- Analysis (the copy operations are counted (the assignments)):

  Let $n = right - left + 1$ be the array lenght.

  Step 2 performs only one copy operation.

  Step 1 performs from 0 to $n-1$ copy operations, in average $(n-1)/2$ copy operations.

  Average number of copy operations =

  $$(n-1)/2 + 1 = n/2 + 1/2$$

  The algorithms complexity is $\boldsymbol{O(n)}$.

Insertion analysis

- What is the best case and worst case complexity?
- Best case – insertion of an element at the end of the array
  - There is no need of performing step 1
  - Only step 2 is performed, just with 1 copy operation

  The algorithms complexity is O(1).
- Worst case – insertion of an element at the beginning of the array
  - Step 1 is performed from 0 to n-1 with n copy operations
  - Step 2 performs only 1 copy operation
  - In total n+1 copy operations

  The algorithms complexity is O(n).

Deletion

- Deletion of an element in an array is a process of removal of a given element and array reorganization
- The deletion can be performed on several ways:
  - Deletion from the beginning of the array
  - Deletion at the end of the array
  - Deletion at any index of the array

Deletion

- Problem: for a given array *a*[*left*…*right*], delete the value *val* in *a*[del]. If it is necessary, shift the values in left, to fill the empty space

(We suppose that $left \leq del \leq right$.)

Deletion

- Animation:

1. copy a[del+1…right] into a[del…right–1].
2. change right = right-1
3. end.

	left = 0	1	2	3	4	5 = *right*	
a	The	cat	sat	on	the	mouse	

del	1

Deletion analysis

- Analysis (the copy operations are counted (the assignments))

  Let n = *right* – *left* + 1 be the array length.

  Step 1 performs between 0 and n–1 copy operations.

  Average number of copy operations =

$$(n - 1)/2 = n/2 - 1/2$$

  The algorithms complexity is ***O*(*n*)**.

Deletion analysis

- What is the best case and worst case complexity?
- Best case – deletion of an element at the end of the array
  - There is no need of performing step 1
  - Only step 2 is performed, just one operation

  The algorithms complexity is O(1).
- Worst case – deletion of an element at the beginning
  - Step 1 is performed from 1 to n-1 with n-1 copy operations
  - Step 2 performs only 1 operation
  - In total n–1+1 operations

  The algorithms complexity is O(n).

Search

- Search of an element in an array is a process of finding a given element in an array with values.
- In this process it is determined whether the key element we are searching for is in the array or not.
- In general case the algorithms complexity is O(n).
  - A special case is when the array is sorted and the complexity is O(log n) (later in materials)

Sorting

- Array sorting is a process in which the elements are being ordered according to some user defined order. Example: numerical, alphabetical etc.
- The standard sorting process is performed in ascending order.

N-dimensional arrays

- A definition for a n-dimensional array:

  If we have *n+1* ordered values pairs, where the first *n* elements consist the index, and the last element is the value associated to that index

  $$(idx_1, idx_2, \ldots, idx_n, value)$$

- When **n=2** then we have a **matrix**

Matrices

$$\begin{array}{c} \begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -0.3 \\ 109 & -64 & 4 \\ 12 & 8 & 9 \\ 3.4 & 36 & 27 \end{bmatrix} \\ \text{column} \end{array}$$

$$\begin{array}{cc} & \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \\ \text{row} & \end{array}$$

Matrices

- Matrix dimensions (mxn)
  - m rows
  - n columns
- The number of elements is m*n
- If m=n the matrix is square
- Access to a matrix element with A[i][j]

Sparse matrices

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

- Matrices that have a lot of elements with values 0
- **Problem: They store a lot of memory!**

Sparse matrices

- Representation:

$$(i,\ j,\ value)$$

  - The first element is the matrix dimension and the number of the non-nul elements
  - Position and value of the non-nul elements

Sparse matrices

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \Rightarrow$$

```
        1,   2,    3
      -------------
A(0,    6,   6,    8)
 (1,    1,   1,   15)
 (2,    1,   4,   22)
 (3,    1,   6,  -15)
 (4,    2,   2,   11)
 (5,    2,   3,    3)
 (6,    3,   4,   -6)
 (7,    5,   1,   91)
 (8,    6,   3,   28)
```

Homework: Suggest appropriate representations of the lower/upper triangular matrices!

Need of dynamical data structures

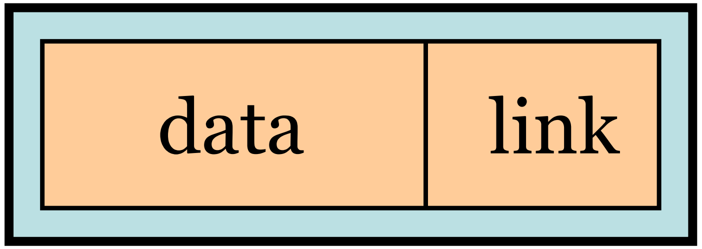
- The arrays disadvantage is that their size can not be changed in execution time.
  - The program will fail when we want to add the n+1-th element, if we have reserved just space for n elements
- What about the dynamical arrays?
  - They automatically grow in length when we try to insert a new element
  - During initialization with length 1 is started, then doubled to length 2, that is from n to 2n always when there is no space.
- Examples: vector in C++, ArrayList in Java

Need of dynamical data structures

- In general case we say that each of the n elements will be shifted in average 2 times, therefore, the complexity for working with dynamical arrays is also O(n)
  - It is the same as if we had provided enough memory in advance to place all the elements

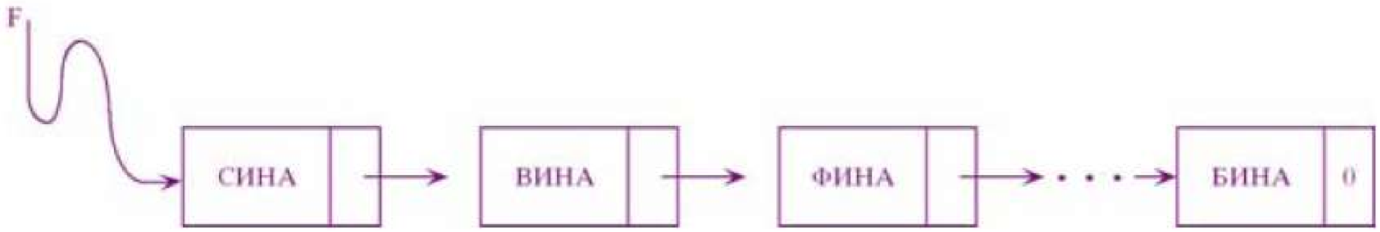
Single linked lists

- Single Linked List SLL
- The ordering of the elements is preserved, but there is no need for memory continuity
- Representation: a set of ordered elements, where each element is described by the value of the node (vertex) (data) and a pointer to the next node (link).

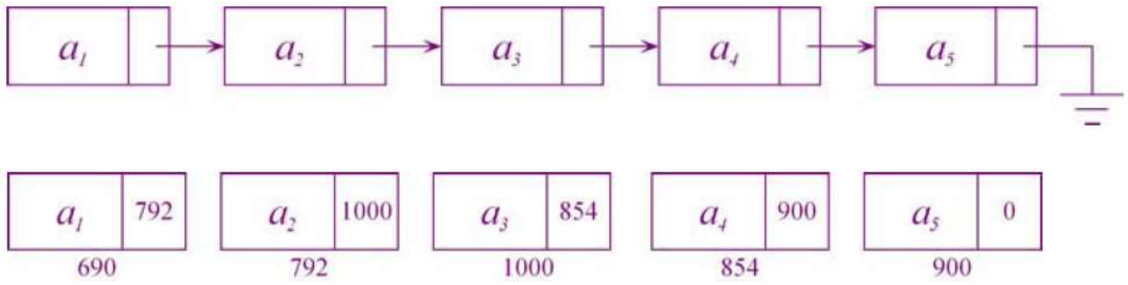


Single linked lists

	Податок	Линк
1	ДИНА	4
2		
3	СИНА	11
4	МИНА	14
5		
6		
7	ШИНА	8
8	БИНА	0
9	ФИНА	1
10		
11	ВИНА	9
	⋮	⋮

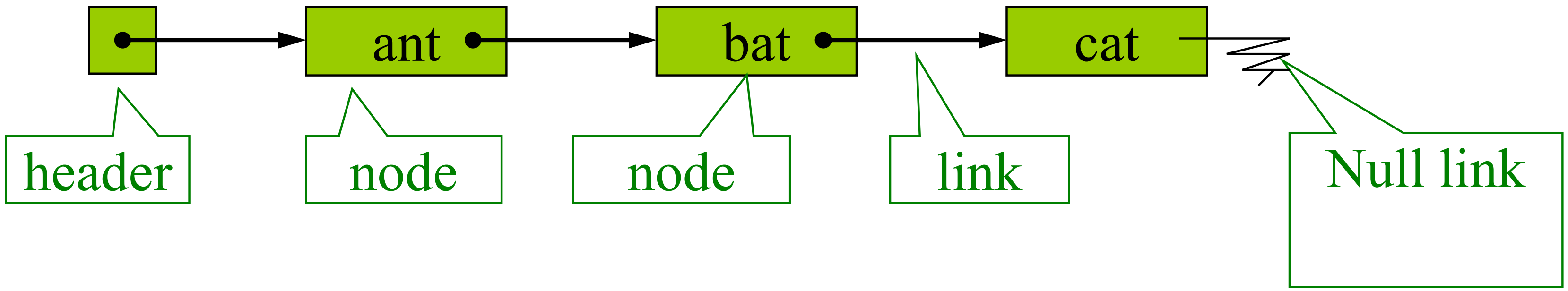


Other representation:



Single linked lists

- Every node (except the last) has a **successor**, and every node (except the first) has a **predecessor**.
- The list **length** of the list is the number of nodes.
- An empty list contains no nodes.
- In linked lists you can:
  - access to/read/delete each element (node).
  - change the links, thereby change the list structure
    - This is not possible with arrays



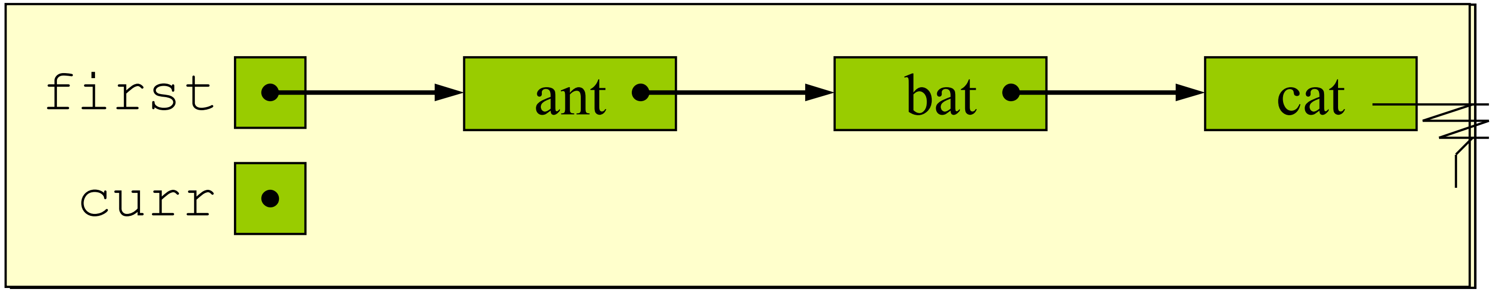
Single linked lists

- SLL operations:
  - Creating empty list
  - List traversal
  - Element insertion in the list
  - Element deletion from the list
  - Element search in the list
  - List deletion
  - etc.

Single linked lists

- Single linked list traversal:

```
public void printFirstToLast () {
// Print all elements in this SLL, in first-to-last order.
    for (SLLNode curr = this.first;
            curr != null; curr = curr.succ)
        System.out.println(curr.element);
}
```



Single linked lists

- Node insertion in SLL includes 4 cases:
  1. Insertion in an empty list
  2. Insertion at the beginning of a nonempty SLL
  3. Insertion at the end of a nonempty SLL
  4. Insertion between two nodes in a nonempty SLL

- When inserting a node, one should pay attention to the links of its predecessors/successors of that node

Single linked lists

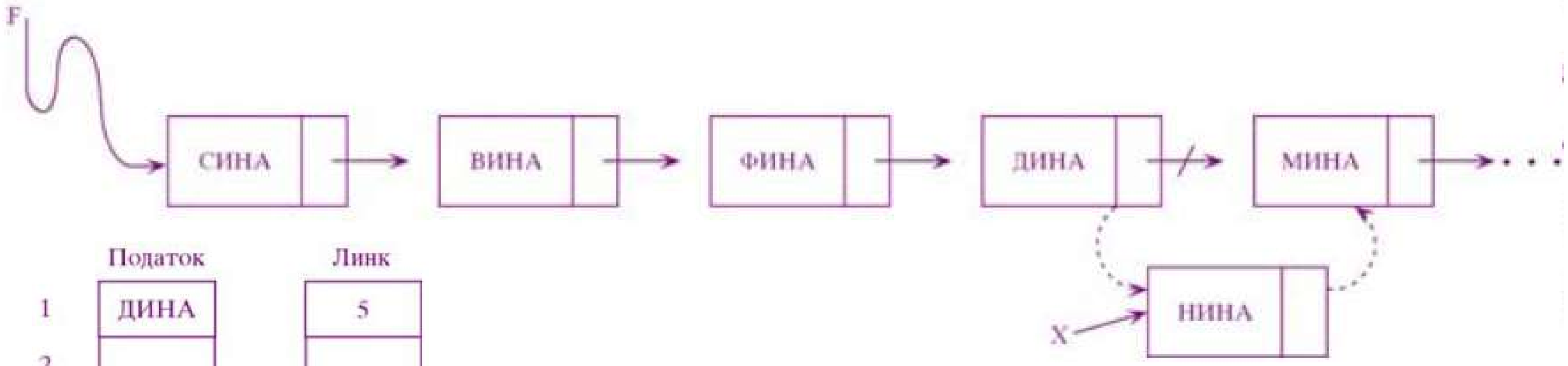
- SLL node insertion algorithm:
  1. a node that is not currently in use in the available memory space is selected
  2. the appropriate value is assigned to the info node field
  3. the link value is filled with the address of the node that should be the follower of the new node
  4. the link value field of the node that will be the predecessor of the new node should be changed with the address of the new node

Single linked lists

- SLL node insertion:
  - an element (node) that is not currently in use in the available memory space is selected
  - the appropriate value is assigned to the info node field
  - the link field value is filled with the address of the node that should be the follower of the new node
  - the link field value of the node that will be the predecessor of the new node should be changed with the address of the new node

Single linked lists

- SLL node insertion:



	Податок	Линк
1	ДИНА	5
2		
3	СИНА	11
4	МИНА	14
5	НИНА	4
6		
7	ШИНА	8
8	БИНА	0
9	ФИНА	1
10		
11	ВИНА	9

Advantages of this approach:

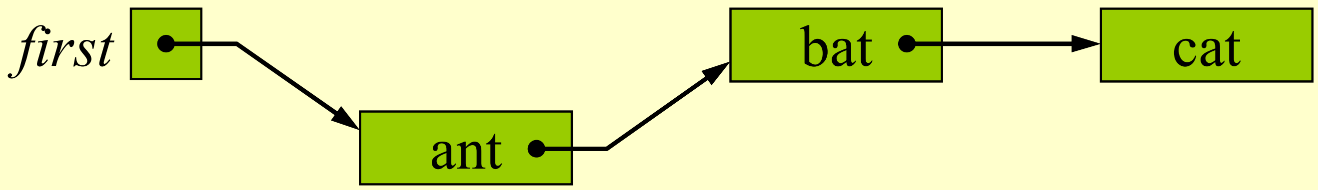
- no shifting of elements to maintain ordering
- there is no need to declare the maximum allowed array length

Single linked lists usage

- Animation (insertion before the first list node):

Insert *elem* in the given SLL at the beginning:

1. Make node *ins* new node with info *elem* and successor null.
2. Put successor of node *ins* to be *first*.
3. Put *first* to be *ins*.
4. End.

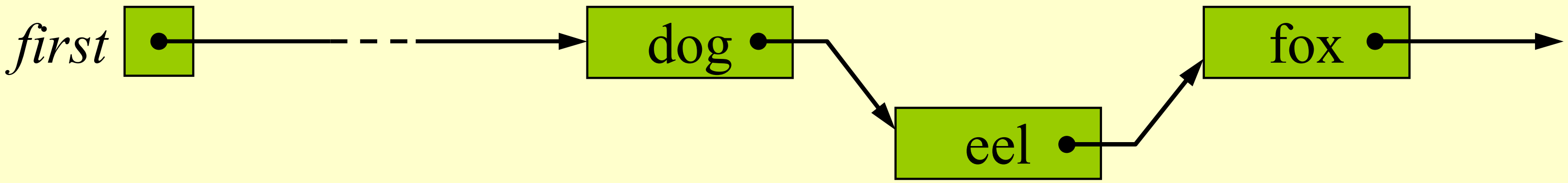


Single linked lists usage

- Animation (insertion after some list node):

Insert *elem* at a given SLL position, after node *pred*:

1. Make node *ins* new node with info *elem* and successor null.
2. Put a successor of node *ins* to be the successor of *pred*
3 Put the successor of *pred* to be *ins*
4. End.

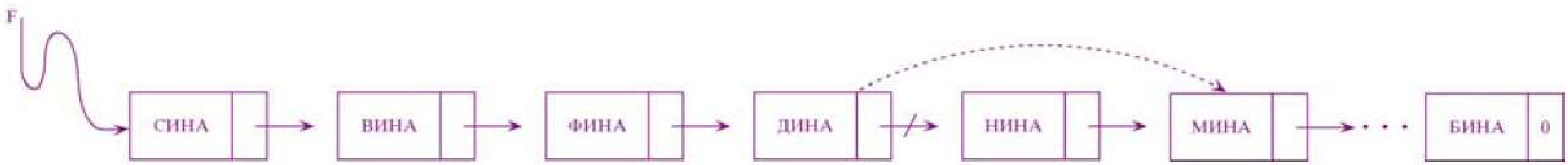


Single linked lists

- SLL node deletion:
  - an element that precedes the element that we want to delete is selected
  - the value of the link field of the predecessor should be changed with the value of the address located in the link field of the node being deleted

Single linked lists

- SLL node deletion:



Single linked lists

- SLL node deletion includes 4 cases:
  1. Deletion from a list with only one node
  2. Deletion of the first node (but not the last)
  3. Deletion of the last node (but not the first)
  4. Deletion of a middle node

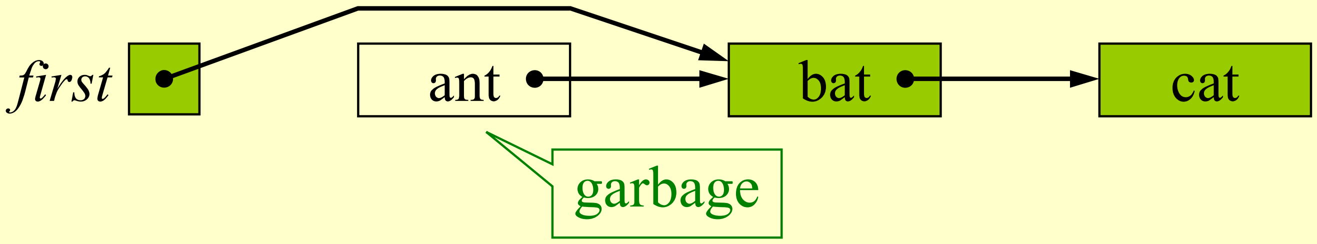
- When deleting a node, attention should be paid to the links of its predecessors/successors to that node

Single linked lists usage

- Animation (deletion of the first node):

To delete the first node in a given SLL:

1. Let *succ* be the successor of the first node
2. We put the successor of the node *first* to *succ*
3. End.

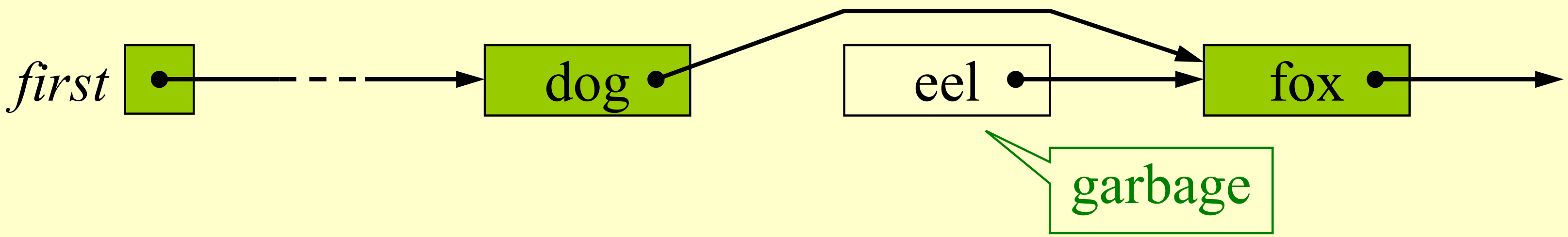


Single linked lists usage

- Animation (middle or last node deletion)

To delete a given node *del* from SLL:

1. Let *succ* be the successor of the node *del*.
2. In the case when the node *del* is not *first*
   - 2.1. Let *pred* be the node predecessor of *del*
   - 2.2. Put the successor of node *pred* to point to *succ*
3. End



Advantages and disadvantages

- SLL usage advantages:
  - lists can never be filled and there will always be room to add new items (unless memory is full)
  - adding and deleting an element is simpler than with arrays
- SLL usage disadvantages:
  - lists require additional memory to store pointers (successors) that do not actually carry useful information
  - lists do not allow efficient access to an arbitrary element, but an entire list must be traversed to reach a given element

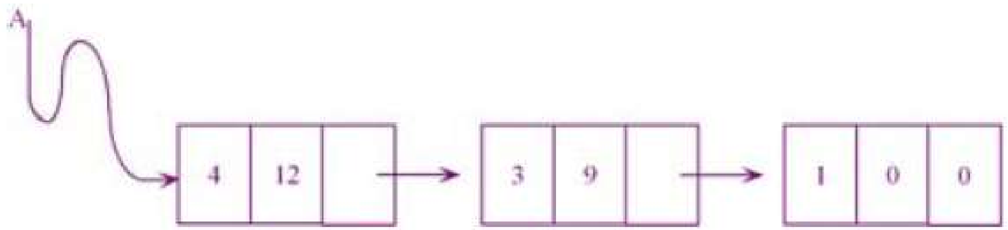
Single linked lists usage

- Representation of polynomials with linked lists :

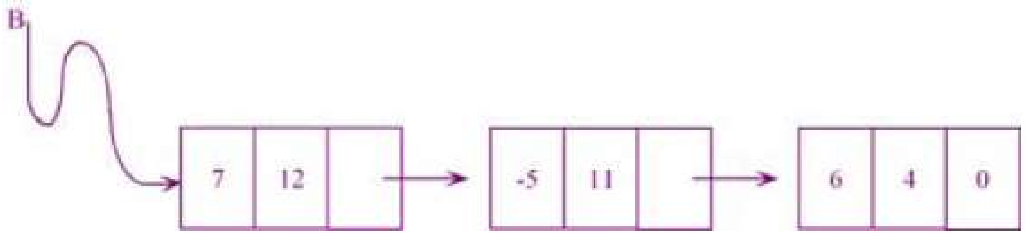
$$A(x) = a_m x^{em} + \ldots + a_1 x^{e1}$$

COEF	EXP	LINK

$$A = 4x^{12} + 3x^{9} + 1$$

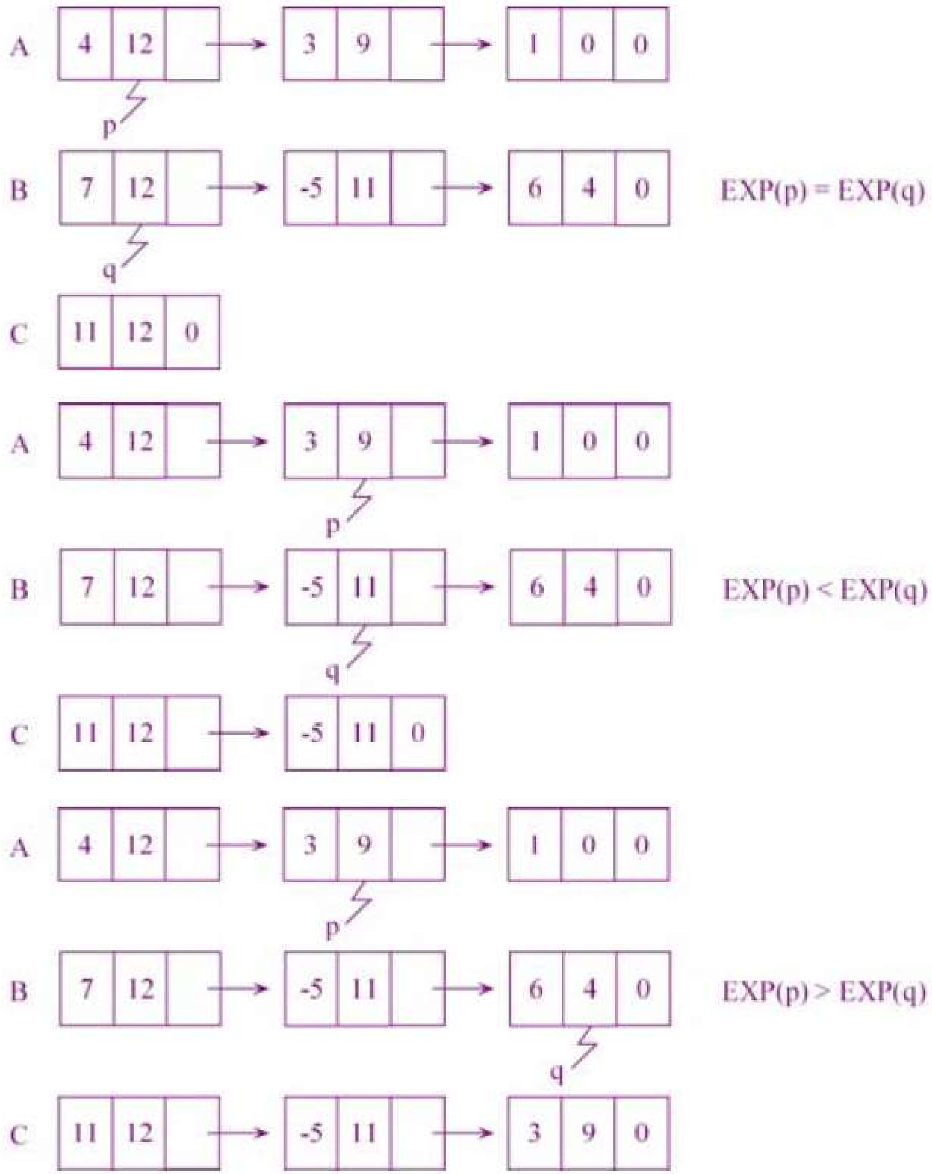


$$B = 7x^{12} - 5x^{11} + 6x^{4}$$



Single linked lists usage

- Realization of the polynomial addition operation:



What is the complexity of this solution?