Linear Data Structures

Raimondas Čiegis

Matematinio modeliavimo katedra, e-paštas: rc@vgtu.lt

March 3 d., 2025

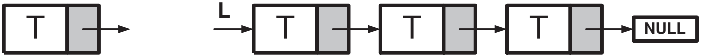
Singly linked list

We note that this data structure is used to implement blockchains – a backbone structure for most cryptocurrencies including bitcoins.

Singly linked list

We note that this data structure is used to implement blockchains – a backbone structure for most cryptocurrencies including bitcoins.

Singly linked lists contain nodes (*a*) which have a value field as well as the next link (pointer), which points to the next node in line of nodes (b):

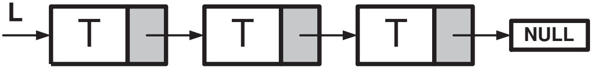


a) b)

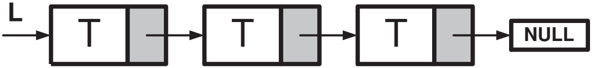
As a basic element we define a *node*, which contains the information data field of type T and the next link (pointer) which defines the address of the next node:

$$
\begin{aligned}
& struct\ node\,\{ \\
& \quad T\ data; \\
& \quad node *\ next; \\
& \}
\end{aligned}
$$

Thus a singly linked list is a sequence (chain) of nodes. An entry link L points to the head of a list (its first node).



Thus a singly linked list is a sequence (chain) of nodes. An entry link L points to the head of a list (its first node).



The next link of the last node is a null pointer. It points to nil (an invalid link) and signals that the last node of the list is reached.

Strong points of this data structure:

A linked list data structure might work well in one case, but cause problems in another.

- A singly linked list is initialized by defining only an entry link, which points to the end of this list (a minimum of memory).

Strong points of this data structure:

A linked list data structure might work well in one case, but cause problems in another.

- A singly linked list is initialized by defining only an entry link, which points to the end of this list (a minimum of memory).
- at any moment the linked list uses a minimum amount of memory, sufficient to define all its nodes,

Strong points of this data structure:

A linked list data structure might work well in one case, but cause problems in another.

- A singly linked list is initialized by defining only an entry link, which points to the end of this list (a minimum of memory).
- at any moment the linked list uses a minimum amount of memory, sufficient to define all its nodes,
- a node of the list can be stored at any free place of the physical memory of a computer, thus neighbour nodes can have very different addresses.

Strong points of this data structure:

A linked list data structure might work well in one case, but cause problems in another.

- A singly linked list is initialized by defining only an entry link, which points to the end of this list (a minimum of memory).
- at any moment the linked list uses a minimum amount of memory, sufficient to define all its nodes,
- a node of the list can be stored at any free place of the physical memory of a computer, thus neighbour nodes can have very different addresses.
- new nodes can be included into or deleted from a list very efficiently.

Disadvantages of singly linked lists:

any node of a list can be reached by starting a search from the beginning of the list and after checking in turn all nodes before it.

The duration of such a process depends where the node is stored in the list.

Disadvantages of singly linked lists:

any node of a list can be reached by starting a search from the beginning of the list and after checking in turn all nodes before it.

The duration of such a process depends where the node is stored in the list.

In the case of arrays each element can be reached directly by using the index of this element, since its address can be computed explicitly.

Thus the duration of reading/writing information from/into any element is the same and don't depend on the index value.

The main operations of linked lists

Let us assume that data is written in a set A and we want to move it into a new singly linked list.

The main operations of linked lists

Let us assume that data is written in a set A and we want to move it into a new singly linked list.

We take elements from A one by one and add them to the beginning of the linked list.

The main operations of linked lists

Let us assume that data is written in a set A and we want to move it into a new singly linked list.

We take elements from A one by one and add them to the beginning of the linked list.

At each step first, we allocate a new node and add the information from the element of A to the value field of the new node. The next link of it is initialized to nill.

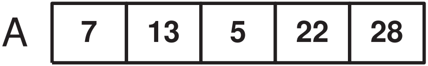
The main operations of linked lists

Let us assume that data is written in a set A and we want to move it into a new singly linked list.

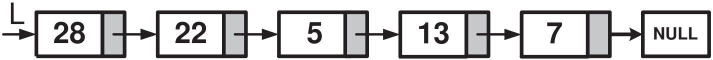
We take elements from A one by one and add them to the beginning of the linked list.

At each step first, we allocate a new node and add the information from the element of A to the value field of the new node. The next link of it is initialized to nill.

Then this node is added to the head of the list.



a)



b)

Finding a node that contains a given datum

In a list L we want to find a node that contains a given datum/key t.

Finding a node that contains a given datum

In a list L we want to find a node that contains a given datum/key t.

The link NULL is returned if such datum is not stored in the list.

Finding a node that contains a given datum

In a list L we want to find a node that contains a given datum/key t.

The link NULL is returned if such datum is not stored in the list.

We start a search from the entry node and iterate through the remaining list elements.

In the worst case it may require iterating through most or all of the list nodes.

How to insert a new node?

How to insert a new node?

The first task is to insert a new element e after the element v.

How to insert a new node?

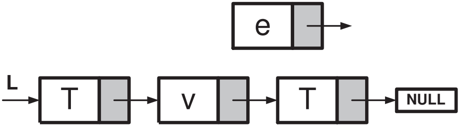
The first task is to insert a new element e after the element v.

It is sufficient to change values of two links.

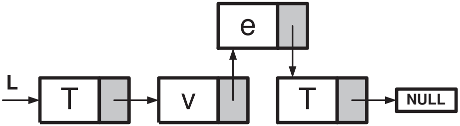
How to insert a new node?

The first task is to insert a new element e after the element v.

It is sufficient to change values of two links.



a)



b)

Insertion of a new element into the linked list: a) the list before inserting element e, b) the list after the insertion .

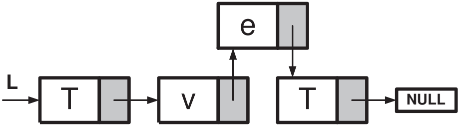
How to delete an element from the list?

How to delete an element from the list?

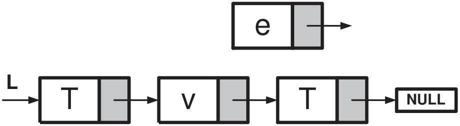
It is a simple task to delete the element e which is the next node after the given element v.

How to delete an element from the list?

It is a simple task to delete the element e which is the next node after the given element v.



a)



b)

Deletion of an element from the linked list: a) the list before deleteing element e, b) the list after the deletion procedure.

It is much harder to delete from a singly linked list the element v itself, since we don't know the address of the node stored before v.

It is much harder to delete from a singly linked list the element v itself, since we don't know the address of the node stored before v.

Please propose your version of an algorithm how to implement this operation avoiding iterations from the head element till we find v and store the address of the previous node.

Stack

Now we consider data structures, that are obtained from singly linked lists by restricting a set of methods (operations) defined for these new data structures.

Stack

Now we consider data structures, that are obtained from singly linked lists by restricting a set of methods (operations) defined for these new data structures.

Yes, we are not enlarging this set of methods, but reducing it.

Stack

Now we consider data structures, that are obtained from singly linked lists by restricting a set of methods (operations) defined for these new data structures.

Yes, we are not enlarging this set of methods, but reducing it.

A stack is the most important data structure in computers.

In computer science, a stack is an abstract data type that serves as a collection of elements, with two main operations:

Push, which adds an element to the collection,

Pop, which removes the most recently added element that was not yet removed.

In computer science, a stack is an abstract data type that serves as a collection of elements, with two main operations:

Push, which adds an element to the collection,

Pop, which removes the most recently added element that was not yet removed.

The order in which an element added to or removed from a stack is described as Last In, First Out, referred to by the acronym LIFO.

Similar to a stack of plates, adding or removing is only possible at the top (Vikipedia).

If a stack is implemented by using singly linked lists, then elements are

added (*push()* method) into a stack,

and removed (*pop()* method) from a stack

only at the head of a list.

If a stack is implemented by using singly linked lists, then elements are

added (*push()* method) into a stack,

and removed (*pop()* method) from a stack

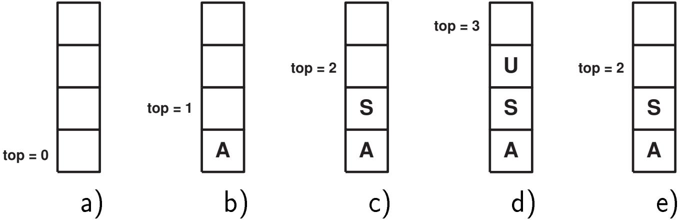
only at the head of a list.

We note, that it possible to implement a stack not only as a singly linked list but also as a pointer to the top element in an array.

$$
\begin{aligned}
&struct\ stack\ \{ \\
&\quad T\ data[N]; \\
&\quad int\ top = 0; \\
&\}
\end{aligned}
$$

In this case a stack has a bounded capacity, thus users must check if it is full before adding a new element.

This figure shows a simple representation of a stack runtime with push and pop operations: first letters A, S, U are added into a stack, and then one letter is removed from it.



Runtime (an array type implementation): a) an empty stack, b) push(A), c) push(S), d) push(U), e) pop()

Postfix form of mathematical expressions

For writing complex mathematical expressions, we generally prefer parentheses to make them more readable.

Postfix form of mathematical expressions

For writing complex mathematical expressions, we generally prefer parentheses to make them more readable.

In computers, expressions with parentheses add unnecessary work while computing.

Postfix form of mathematical expressions

For writing complex mathematical expressions, we generally prefer parentheses to make them more readable.

In computers, expressions with parentheses add unnecessary work while computing.

In order to minimize the computational work, new notations have been made.

Infix form:

The typical mathematical form of expression. In infix form, an operator is written in between two operands.

Infix form:

The typical mathematical form of expression. In infix form, an operator is written in between two operands.

An infix form of a sum of two numbers a and b is written as

$$a + b.$$

Infix form:

The typical mathematical form of expression. In infix form, an operator is written in between two operands.

An infix form of a sum of two numbers a and b is written as

$$a + b.$$

Prefix form:

In prefix expression, an operator is written before its operands.

This notation is known as Polish notation. It is used in calculators.

Infix form:

The typical mathematical form of expression. In infix form, an operator is written in between two operands.

An infix form of a sum of two numbers a and b is written as

$$a + b.$$

Prefix form:

In prefix expression, an operator is written before its operands.

This notation is known as Polish notation. It is used in calculators.

A prefix form of a sum of two numbers a and b is written as

$$+ab.$$

Postfix form:

In postfix expression, an operator is written after its operands.

Postfix form:

In postfix expression, an operator is written after its operands.

A postfix form of a sum of two numbers a and b is written as

$$ab+.$$

Prefix and *postfix* forms are very convenient, since parentheses are not required to define the priority of operations in any mathematical expression.

Prefix and *postfix* forms are very convenient, since parentheses are not required to define the priority of operations in any mathematical expression.

Let's consider the following arithmetical expression $a + b * c$.

Prefix and *postfix* forms are very convenient, since parentheses are not required to define the priority of operations in any mathematical expression.

Let's consider the following arithmetical expression $a + b * c$.

The postfix form is written as

$$\begin{aligned} a + b * c \quad &\longrightarrow \quad a + (b * c) \quad \longrightarrow \quad a + (b\, c\, *) \\ &\longrightarrow \quad a\, (b\, c\, *) + \quad \longrightarrow \quad a\, b\, c\, *\, +. \end{aligned}$$

Prefix and *postfix* forms are very convenient, since parentheses are not required to define the priority of operations in any mathematical expression.

Let's consider the following arithmetical expression $a + b * c$.

The postfix form is written as

$$\begin{aligned} a + b * c \quad &\longrightarrow \quad a + (b * c) \quad \longrightarrow \quad a + (b\, c\, *) \\ &\longrightarrow \quad a\,(b\, c\, *) + \quad \longrightarrow \quad a\, b\, c\, *\, +. \end{aligned}$$

Now let's consider another arithmetical expression $(a + b) * c$. No parentheses are required in its postfix form:

$$(a + b) * c \;\longrightarrow\; (a\, b +) * c \;\longrightarrow\; (a\, b +)\, c\, * \;\longrightarrow\; a\, b\, +\, c\, *\, .$$

The rank of an operator is called its precedence, and an operation with a higher precedence is performed before operations with lower precedence.

- The exponentiations are given precedence over both addition and multiplication. We denote it as $\wedge$:

$$a^b = a \wedge b\,.$$

The rank of an operator is called its precedence, and an operation with a higher precedence is performed before operations with lower precedence.

- The exponentiations are given precedence over both addition and multiplication. We denote it as $\wedge$:

$$a^b = a \wedge b\,.$$

The rank of an operator is called its precedence, and an operation with a higher precedence is performed before operations with lower precedence.

- The exponentiations are given precedence over both addition and multiplication. We denote it as $\wedge$:

$$a^b = a \wedge b\,.$$

  This operation is computed from right to left, thus a new exponentiation operator has high precedence over previous exponents:

$$a \wedge b \wedge c = a \wedge (b \wedge c)\,.$$

The rank of an operator is called its precedence, and an operation with a higher precedence is performed before operations with lower precedence.

- The exponentiations are given precedence over both addition and multiplication. We denote it as $\wedge$:

$$a^b = a \wedge b\,.$$

  This operation is computed from right to left, thus a new exponentiation operator has high precedence over previous exponents:

$$a \wedge b \wedge c = a \wedge (b \wedge c)\,.$$

Let's calculate a simple example:

$$3^{3^3} = 3^{27}.$$

- A lower precedence is given for multiplication and division. Operations with the same precedence are computed from left to right:

$$a * b/c = (a * b)/c\,.$$

- A lower precedence is given for multiplication and division. Operations with the same precedence are computed from left to right:

$$a * b/c = (a * b)/c\,.$$

- Addition and subtraction are granted the lowest precedence. Operations with the same precedence are computed from left to right:

$$a - b + c = (a - b) + c\,.$$

- A lower precedence is given for multiplication and division. Operations with the same precedence are computed from left to right:

$$a * b/c = (a * b)/c\,.$$

- Addition and subtraction are granted the lowest precedence. Operations with the same precedence are computed from left to right:

$$a - b + c = (a - b) + c\,.$$

- Perentheses define the highest precedence, thus a mathematical expression inside perentheses is computed before any arithmetical operation and the obtained result defines a new operand.

How to convert Infix expression to Postfix form

Add the start [and end] symbols to bound the infix ezpression.

How to convert Infix expression to Postfix form

Add the start [and end] symbols to bound the infix ezpression.

1. While the infix expression is not empty scan it from left to right: Read the next symbol s.

How to convert Infix expression to Postfix form

Add the start [and end] symbols to bound the infix ezpression.

1. While the infix expression is not empty scan it from left to right: Read the next symbol s.
2. If the scanned character is an operand, put s in the postfix expression (print it or put it into the second stack P).

How to convert Infix expression to Postfix form

Add the start [and end] symbols to bound the infix ezpression.

1. While the infix expression is not empty scan it from left to right:
Read the next symbol s.
2. If the scanned character is an operand, put s in the postfix expression (print it or put it into the second stack P).
If the scanned character is [or (then push it into the stack S.

How to convert Infix expression to Postfix form

Add the start [and end] symbols to bound the infix ezpression.

1. While the infix expression is not empty scan it from left to right: Read the next symbol s.
2. If the scanned character is an operand, put s in the postfix expression (print it or put it into the second stack P).
If the scanned character is [or (then push it into the stack S.
3. Otherwise, if s is an operator, then do the following:
   - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack S or the stack is empty, or the stack contains symbols (, [, then push s into the stack S.

Exponeniation operator $\wedge$ is right associative and other operators $+$, $-$, $*$, $/$ are left-associative.

- Check especially for a condition when the operator at the top of the stack and the scanned operator both are $\wedge$. In this situation, the precedence of the scanned operator is higher due to its right associativity. So it will be Pushed into the stack S.

- In all the other cases when the rank of the top operator in the stack is the same as for the scanned operator, then we Pop the operator from the stack because of left associativity due to which the scanned operator has less precedence and print it.

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
  - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and Push the scanned operator into the operator stack.)

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
  - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and Push the scanned operator into the operator stack.)

4. If the scanned character is (, push it to the stack.

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
  - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and Push the scanned operator into the operator stack.)

4. If the scanned character is (, push it to the stack.
5. If the scanned character is), pop the stack and output it until (is encountered, and discard both the parenthesis.

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
  - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and Push the scanned operator into the operator stack.)

4. If the scanned character is (, push it to the stack.
5. If the scanned character is), pop the stack and output it until (is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the infix expression is fully scanned.

- Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
  - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and Push the scanned operator into the operator stack.)

4. If the scanned character is (, push it to the stack.
5. If the scanned character is), pop the stack and output it until (is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the infix expression is fully scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.

Let us consider Infix arithmetical expression

$$[a + (b * c + d) \wedge f].$$

By using the presented algorithm we convert it to the Postfix form.

Let us consider Infix arithmetical expression

$$[a + (b * c + d) \wedge f].$$

By using the presented algorithm we convert it to the Postfix form.

Infix	Stack					Postfix
[	[					
a	[					a
+	[	+				
(	[	+	(			
b	[	+	(			b
*	[	+	(	*		
c	[	+	(	*		c

We consider *infix* expression $[a + (b * c + d) \wedge f]$.

Infix	Stack					Postfix
c	[	+	(	*		c
+	[	+	(			*
	[	+	(	+		
d	[	+	(	+		d
)	[	+				+
^	[	+	^			
f	[	+	^			f
]	[	+				^
	[					+

Thus the postfix form of the infix arithmetical expression $a + (b * c + d) \wedge f$ is written as

$$abc * d + f \wedge + .$$

Thus the postfix form of the infix arithmetical expression $a + (b * c + d) \wedge f$ is written as

$$abc * d + f \wedge + .$$

It is interesting to note that in order to compute a value of the postfix form we use stacks again.