Infix- till postfixomvandling

infix: *operand* **operator** *operand* exempel: `a + b`

postfix: *operand* *operand* **operator** exempel: `a b +`

`a + b - c`	⇒	`a b + c -`
`a + (b - c)`	⇒	`a b c - +`
`a * b + c`	⇒	`a b * c +`
`a * (b + c)`	⇒	`a b c + *`
`x = a / b + c * (d + e) - f`	⇒	`x a b / c d e + * + f - =`

Varför postfix?

- Kan läsas och behandlas sekvensiellt
  - ingen vetskap om vad som följer (”look-ahead”) krävs
  - redan erhållna delresultat kan sparas med hjälp av stackteknik
- Väl lämpad för maskinell bearbetning – t.ex. beräkning, kodgenerering, etc.

Infixuttryck med additiva, multiplikativa operatorer och tilldelning

- *Operanderna* är heltal eller reella tal, utan tecken
- *Operatorerna* är `=`, `+`, `-`, `*`, `/` och `^`, samtliga binära
- *Parenteser* kan förekomma för att avgränsa deluttryck och styra beräkningsordningen

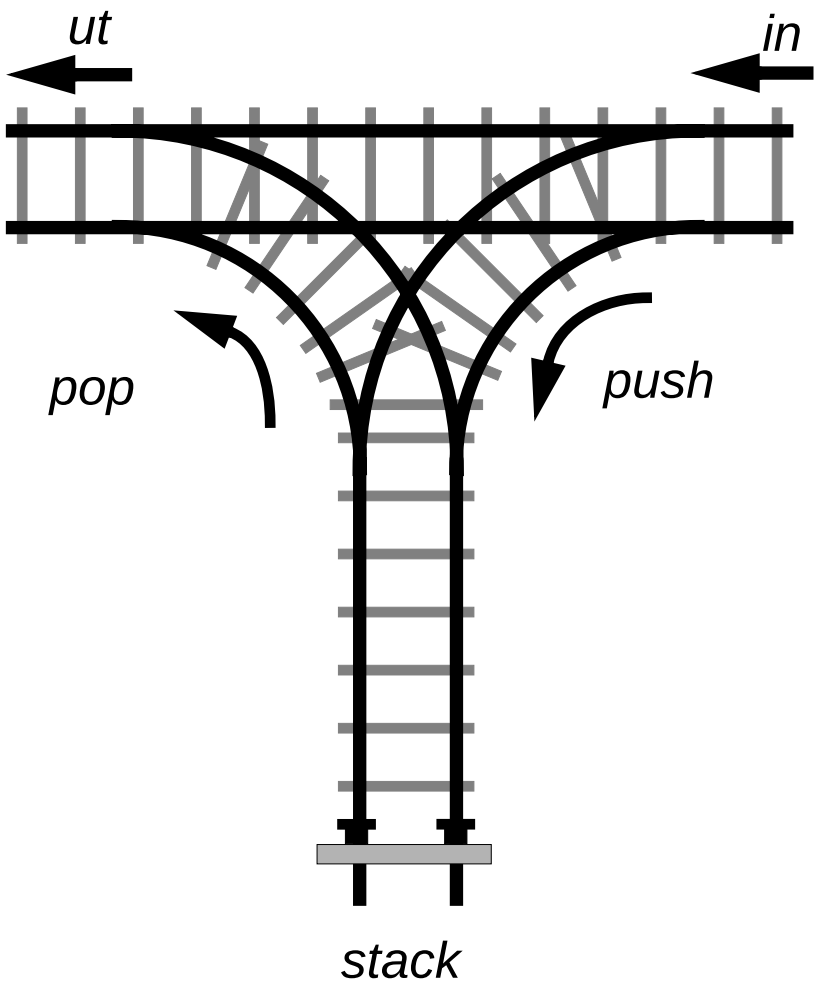
Prioritetsordning för operatorerna:

Hög	^		högerassociativ
	*	/	vänsterassociativa
	+	-	vänsterassociativa
Låg	=		högerassociativ

Parenteser kan inordnas i prioritetshierarkin eller särbehandlas.

”Dijkstras Järnvägsalgoritm”

Enkel ”rangerbangård” realiserad med en stack.



Algoritm

Inkommande	Åtgärd

^
* /
+ -
=
	sålänge *en operator finns överst på stacken* **och**
	prioriteten för operatorn på stacken >= prioriteten för inkommande operator
	{
	ta bort och skriv ut operatorn överst på stacken
	}
	stacka den inkommande operatorn

(	*stacka '(' *

)	**sålänge** *stacken inte innehåller '(' överst*
	{
	ta bort och skriv ut operatorn överst på stacken
	}
	ta bort '(' från stacken – ska finnas om infixuttrycket sålångt är korrekt

slut	*ta bort och skriv ut alla operatorer på stacken*
operand	*skriv ut operanden*

Felhanteringskontroller ej införda ovan!

Exempel 1

```
a * (b + c)
```

Inkommande	*Stack*	*Ut*
a		a
*	*	a
(	* (	a
b	* (	a b
+	* (+	a b
c	* (+	a b c
)	*	a b c +
slut		a b c + *

Exempel 2

```
a + b * (c + d) - e
```

Inkommande	*Stack*	*Ut*
a		a
+	+	a
b	+	a b
*	+ *	a b
(	+ * (	a b
c	+ * (	a b c
+	+ * (+	a b c
d	+ * (+	a b c d
)	+ *	a b c d +
-	-	a b c d + * +
e	-	a b c d + * + e
slut		a b c d + * + e -

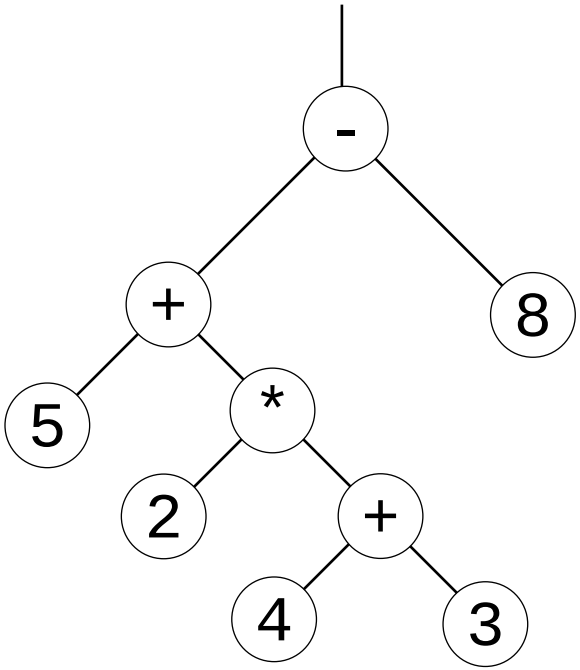
Infix => postfix ⇒ uttryckträd

```
5 + 2 * (4 + 3) - 8
```

⇓

```
5 2 4 3 + * + 8 -
```

⇓

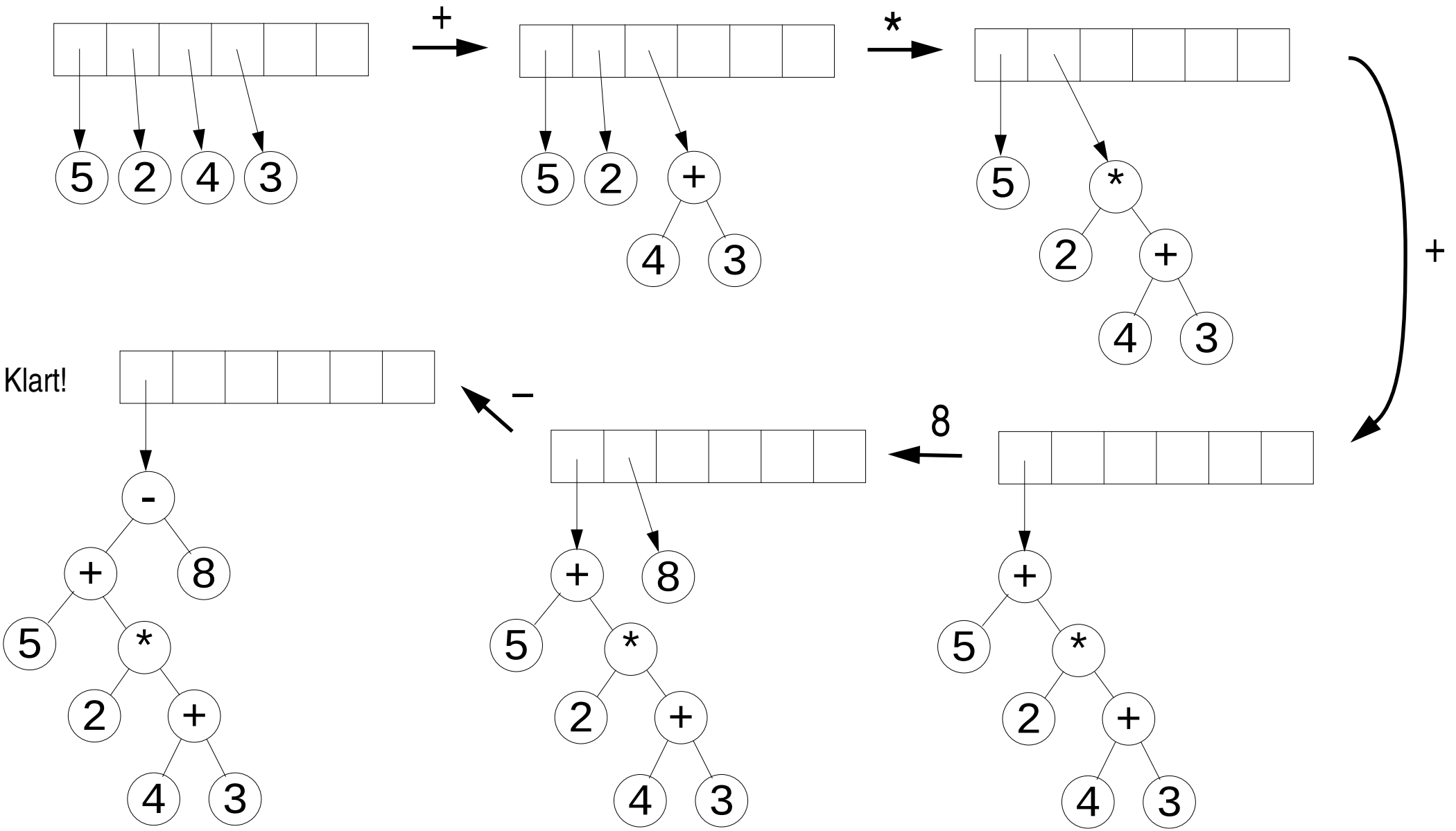


Generering av uttryckträd

Ett postfixuttryck läses sekventiellt och subträd byggs successivt upp på en stack:

- För en *operand* skapas en ny nod och pekaren stackas.
  - noden har inga subträd.
- För en *operator* skapas en ny nod där operatorn lagras:
  - de två översta subträden på stacken (ska finnas minst två) hämtas från stacken
  - det översta subträdet på stacken ska bli den nya operatornodens högra subträd
  - det näst översta subträdet på stacken ska bli den nya operatornodens vänstra subträd
  - den nya operatornoden stackas

Exempel: 5 2 4 3 + * + 8 -



Binärt träd

```
struct Tree_Node
{
    explicit Tree_Node(char symbol, Tree_Node* left = nullptr, Tree_Node* right = nullptr)
      : symbol_{symbol}, left_{left}, right_{right} {}

    ~Tree_Node() { delete left_; delete right_; }


    char symbol_;

    Binary_Tree left_;
    Binary_Tree right_;
};
```

Trädtraversering

Traversering kan användas för att t.ex. skriva ut uttryckträd, generera postfix, …

Den operation som ska utföras på varje nod görs genom funktionen `node_op`.

Inordertraversering – djupet först, från vänster till höger:

```
void traverse(Tree_Node* tree, void (*op)(Tree_Node*))
{
    if (tree != nullptr)
    {
        traverse(tree->left_, node_op);          // först vänster subträd
        op(tree);                                // utför operationen op på noden
        traverse(tree->right_, node_op);         // sedan höger subträd
    }
}
```

Varianterna preorder- och postordertraversering erhålls genom att flytta anropet av funktionen `op` först respektive sist.