Data structures and algorithms

Tutorial 4

Amr Keleg

Faculty of Engineering, Ain Shams University

March 12, 2020

Contact: amr_mohamed@live.com

Outline

Outline

We are all familiar with writing mathematical equations as follows:

- 2 + 3 * 5
- a * (b + c) - d

This notations is called the IN-fix notation (IN since operators are between the operands).

There are two other ways to represent equations:

- PRE-fix notation
- POST-fix notation (Reverse Polish Notation)

[1] The description "Polish" refers to the nationality of logician Jan ukasiewicz, who invented Polish notation in 1924.

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$
- $2 - 2 + 2$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$
- $2 - 2 + 2 = 0 + 2 = 2$ (Left to right evaluation)

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$
- $2 - 2 + 2 = 0 + 2 = 2$ (Left to right evaluation)
- 2 - 3 - 4

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$
- 2 - $2 + 2 = 0 + 2 = 2$ (Left to right evaluation)
- 2 - 3 - 4 = -5 (Left to right evaluation)

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$
- $2 - 2 + 2 = 0 + 2 = 2$ (Left to right evaluation)
- $2 - 3 - 4 = -5$ (Left to right evaluation)
- $16 / 4 / 4$

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$
- $2 - 2 + 2 = 0 + 2 = 2$ (Left to right evaluation)
- $2 - 3 - 4 = -5$ (Left to right evaluation)
- $16 / 4 / 4 = 1$ (Left to right evaluation)

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- $2 + 3 * 2 = 2 + 6 = 8$
- $(2 + 3) * 2 = 5 * 2 = 10$
- $2 + 2 + 2 = 6$
- 2 - $2 + 2 = 0 + 2 = 2$ (Left to right evaluation)
- 2 - 3 - 4 = -5 (Left to right evaluation)
- $16 / 4 / 4 = 1$ (Left to right evaluation)
- 2 ^ 3 ^ 2

We will focus on the Post-fix notation.
Before doing so, let's revise the precedence of operators. What are the results of the following expressions?

- 2 + 3 * 2 = 2 + 6 = 8
- (2 + 3) * 2 = 5 * 2 = 10
- 2 + 2 + 2 = 6
- 2 - 2 + 2 = 0 + 2 = 2 (Left to right evaluation)
- 2 - 3 - 4 = -5 (Left to right evaluation)
- 16 / 4 / 4 = 1 (Left to right evaluation)
- 2 ^ 3 ^ 2 = 2 ^ (3^ 2) = 2 ^ 9 = 512 (Right to left evaluation)

Infix notation	Postfix notation
a + b	

Infix notation	Postfix notation
a + b	a b +

Infix notation	Postfix notation
a + b	a b +
a * b	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -
a + b + c	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -
a + b + c	a b + c +

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -
a + b + c	a b + c +
a + b * c	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -
a + b + c	a b + c +
a + b * c	a b c * +

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -
a + b + c	a b + c +
a + b * c	a b c * +
a * b + c	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -
a + b + c	a b + c +
a + b * c	a b c * +
a * b + c	a b * c +

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	a b -
a + b + c	a b + c +
a + b * c	a b c * +
a * b + c	a b * c +
(a + b) * c	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	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 *

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	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 *
a - b - c	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	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 *
a - b - c	a b - c - (Left to right evaluation)

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	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 *
a - b - c	a b - c - (Left to right evaluation)
a ^ b ^ c	

Infix notation	Postfix notation
a + b	a b +
a * b	a b *
a - b	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 *
a - b - c	a b - c - (Left to right evaluation)
a ^ b ^ c	a b c ^ ^ (Right to left evaluation)

Outline

- X y * t +

- X y * t + -> (X*y) + t

- X y * t + -> (X*y) + t
- A B * X Y - /

- X y * t + -> (X*y) + t
- A B * X Y - / -> (A*B) / (X-Y)

Outline

Let's check some examples first:

- a + b * c

Let's check some examples first:

- a + b * c a b c * +

Let's check some examples first:

- a + b * c a b c * +
- a * b + c

Let's check some examples first:

- a + b * c a b c * +
- a * b + c a b * c +

Let's check some examples first:

- a + b * c a b c * +
- a * b + c a b * c +
- a + b * c ^ d

Let's check some examples first:

- a + b * c a b c * +
- a * b + c a b * c +
- a + b * c ^ d a b c d ^ * +

Let's check some examples first:

- a + b * c a b c * +
- a * b + c a b * c +
- a + b * c ^ d a b c d ^ * +
- a + b - c

Let's check some examples first:

- a + b * c a b c * +
- a * b + c a b * c +
- a + b * c ^ d a b c d ^ * +
- a + b - c a b + c -

Let's check some examples first:

- a + b * c a b c * +
- a * b + c a b * c +
- a + b * c ^ d a b c d ^ * +
- a + b - c a b + c -
- a + (b - c) * d

Let's check some examples first:

- a + b * c a b c * +
- a * b + c a b * c +
- a + b * c ^ d a b c d ^ * +
- a + b - c a b + c -
- a + (b - c) * d a b c - d * +

- Scan input string from left to right character by character.

- Scan input string from left to right character by character.
- If the character is an operand, print the operand to the output.

- Scan input string from left to right character by character.
- If the character is an operand, print the operand to the output.
- If the character is an operator and operator's stack is empty, push operator into operators' stack.

- Scan input string from left to right character by character.
- If the character is an operand, print the operand to the output.
- If the character is an operator and operator's stack is empty, push operator into operators' stack.
- If the operator's stack is not empty, there may be following possibilities:

- If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operators' stack.

- If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operators' stack.
- If the precedence of scanned operator is less than the top most operator of operators' stack, pop the operators from operators' stack until we find a low precedence operator than the scanned character. Never pop out '(' whatever may be the precedence level of scanned character.

- If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operators' stack.
- If the precedence of scanned operator is less than the top most operator of operators' stack, pop the operators from operators' stack until we find a low precedence operator than the scanned character. Never pop out '(' whatever may be the precedence level of scanned character.
- If the precedence of scanned operator is equal the top most operator of operators' stack, pop the operators from operators' stack ONLY IF THE OPERATOR FOLLOWS A LEFT TO RIGHT EVALUATION (+-/*).

- If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operators' stack.
- If the precedence of scanned operator is less than the top most operator of operators' stack, pop the operators from operators' stack until we find a low precedence operator than the scanned character. Never pop out '(' whatever may be the precedence level of scanned character.
- If the precedence of scanned operator is equal the top most operator of operators' stack, pop the operators from operators' stack ONLY IF THE OPERATOR FOLLOWS A LEFT TO RIGHT EVALUATION (+-/*).
- If the character is opening round bracket ('('), push it into operator's stack.

- If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operators' stack.
- If the precedence of scanned operator is less than the top most operator of operators' stack, pop the operators from operators' stack until we find a low precedence operator than the scanned character. Never pop out '(' whatever may be the precedence level of scanned character.
- If the precedence of scanned operator is equal the top most operator of operators' stack, pop the operators from operators' stack ONLY IF THE OPERATOR FOLLOWS A LEFT TO RIGHT EVALUATION (+-/*).
- If the character is opening round bracket ('('), push it into operator's stack.
- If the character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

- If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operators' stack.
- If the precedence of scanned operator is less than the top most operator of operators' stack, pop the operators from operators' stack until we find a low precedence operator than the scanned character. Never pop out '(' whatever may be the precedence level of scanned character.
- If the precedence of scanned operator is equal the top most operator of operators' stack, pop the operators from operators' stack ONLY IF THE OPERATOR FOLLOWS A LEFT TO RIGHT EVALUATION (+-/*).
- If the character is opening round bracket ('('), push it into operator's stack.
- If the character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

- Now pop out all the remaining operators from the operator's stack and print it to the output.

Outline

- (4+3)*5-2 -> 4 3 + 5 * 2 -
- (x+t*y)/2 -> x t y * + 2 /
- (-b+sqrt(b*b+4*a*c))/(2*a) ->
  b ~ b b * 4 a * c * + sqrt 2 a * /

Outline

Outline

```
#include<stack>

stack<string> st;
st.push("str1");
st.push("str2");
while(!st.empty()){
   cout<<st.top()<<endl;
   st.pop();
}
```

- Stack: `http://www.cplusplus.com/reference/stack/stack/`

Outline

Write a function that takes and array and mutliplies each element by 2.

```
int * double_arr(int arr[], int size){
  int result[size];
  for(int i=0; i<size; i++){
    result[i] = 2 * arr[i];
  }
  return result;
}
```

Problem: result is local to the function and will be deleted once the function ends.

```
int * double_arr(int arr[], int size){
  for(int i=0; i<size; i++){
    arr[i] = 2* arr[i];
  }
  return arr;
}
```

The code works since the array won't be deleted when the function execution ends.

```
void double_arr(int arr[], int size){
  for(int i=0; i<size; i++){
    arr[i] = 2* arr[i];
  }
}
```

We can just make a void function.

What if we don't want to modify the input array?

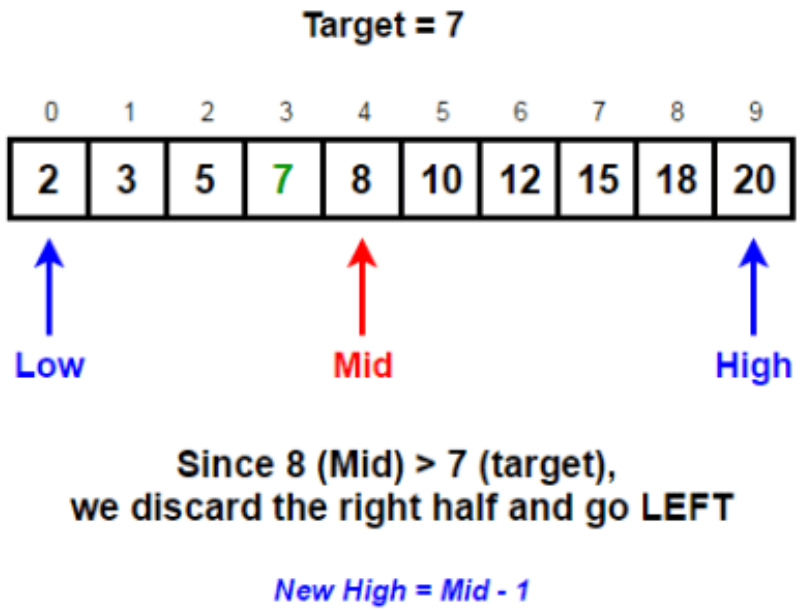
What if we don't want to modify the input array?

```
void double_arr(int arr[], int size, int result[]){
  for(int i=0; i<size; i++){
    result[i] = 2* arr[i];
  }
}
```

Outline

Outline

In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array.



```
int binary_search(int arr[], int item, int l, int r){
  if (l<r){
    // This might happen if the array was {7,8}
    // and we are searching for item 1
    return -1;
  }

  if (l==r){
    if (arr[l] != item) return -1;
    return l;
  }

  int mid = (l+r)>>1;
  if (arr[mid] == item)
    return mid;
  if(arr[mid] >item)
    return binary_search(arr, item, l, mid-1);
  // arr[mid] < item
  return binary_search(arr, item, mid+1, r);
}
```

Outline

We will discuss it later.
Maybe in the tutorial after the midterms :D

Outline

Outline

Write a function that sorts an array given that its two halves are sorted.
What is the expected complexity?

```
void merge(int l_arr[], int r_arr[],
        int l_size, int r_size, int sorted_arr[]){
   ...
}

int main(){
   int l_arr[3] = {3, 5, 7};
   int r_arr[3] = {1, 2, 6};
   int arr[6];

   merge(l_arr, r_arr, 3, 3, arr);
}
```

Write a function that sorts an array given that its two halves are sorted.

```
void merge(int l_arr[], int r_arr[],
        int l_size, int r_size, int sorted_arr[]){
  int index = 0;
  int l_index = 0;
  int r_index = 0;
  while(l_index < l_size && r_index < r_size){
    if (arr[l_index] <= arr[r_index])
      sorted_arr[index++] = l_arr[l_index++];
    else
      sorted_arr[index++] = r_arr[r_index++];
  }

  // Copy the remaining elements of either the left or t
}
```

Write a function that sorts an array given that its two halves are sorted.

```
void merge(int l_arr[], int r_arr[],
           int l_size, int r_size, int sorted_arr[]){
  // Copy the remaining elements of either the left or t
  while(l_index < l_size)
    result[index++] = l_arr[l_index++];
  while(r_index < r_size)
    result[index++] = r_arr[r_index++];
}
```

```
void merge(int l_arr[], int r_arr[],
        int l_size, int r_size, int sorted_arr[]){ ... }

void merge_sort(int arr[], int arr_len,
         int sorted_arr[]){
  if(arr_len==1){
    sorted_arr[0] = arr[0];
    return ;
  }
  int half_len = (arr_len)/2;
  int * l_arr = new int[half_len];
  int * sorted_l_arr = new int[half_len];
  int i;
  for(i=0; i< half_len; i++)
    l_arr[i] = arr[i];
  int * r_arr = new int[arr_len - half_len];
  int * sorted_r_arr = new int[arr_len - half_len];
  for(int j=0; i< arr_len; i++, j++)
    r_arr[j] = arr[i];
```

```
void merge_sort(int arr[], int arr_len,
        int sorted_arr[]){

  merge_sort(l_arr, half_len, sorted_l_arr);
  merge_sort(r_arr, arr_len - half_len, sorted_r_arr);
  merge(sorted_l_arr, sorted_r_arr, half_len,
      arr_len - half_len, sorted_arr);
  delete [] l_arr;
  delete [] sorted_l_arr;
  delete [] r_arr;
  delete [] sorted_r_arr;
}
```

How to trace merge sort for this example?
int arr[] =8, 2, 1, 3, 5, 0, 4, 6

Feedback form:

Amr: `https://forms.gle/Kgav5jCeoFN1nDzV9`

Fady: `TODO`