# DSA - Seminar 5

1. Consider the following problem: Determine the sum of the largest *k* elements from a vector containing *n* distinct numbers. For example, if the array contains the following 10 elements [6, 12, 91, 9, 3, 5, 25, 81, 11, 23] and k = 3, the result should be: 91 + 81 + 25 = 197.

   I. Find the maximum k times (especially good if k is small)
   - If we just call the maximum function 3 times for our example, it will return 91 each time, so we need a solution where we also have an upper bound, and we are searching for the maximum which is less than that value.
   - First, maximum is 91. At the second call we want the maximum which is less than 91, we will get 81.
   - At the second call we want the maximum which is less than 81, we will get 25.
   - Complexity of the approach: $\Theta(k*n)$ – finding the maximum is $\Theta(n)$ and we do this k times.

   II. Sort the array in a descending order and pick the first k elements (especially good is k is large).
   - Sorting can be done in $\Theta(n*\log_2 n)$ time
   - Computing the sum of the first k elements: $\Theta(k)$
   - In total $\Theta(n*\log_2 n) + \Theta(k) \in \Theta(n*\log_2 n)$

   III. Use a binary max-heap. Add all the elements to the heap and remove the first k.
   - Adding an element to a heap with n elements is $O(\log_2 n)$.
   - Removing an element from a heap with n elements is $O(\log_2 n)$.
   - In total we have $O(n*\log_2 n) + O(k*\log_2 n)$. Since n >= k, this is $O(n*\log_2 n)$

```
function sumOfK(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
       init(h, "≥") //assume we have the Heap data structure implemented. We
       initialize a heap with the relation "≥" (a max-heap)
       for i ← 1, n execute
              add(h, elems[i]) //add operation was discussed at Lecture 8
       end-for
       sum ← 0
       for i ← 1, k execute
              elem ← remove(h) //remove operation was discussed at Lecture 8
              sum ← sum + elem
       end-for
       sumOfK ← sum
end-function
```

   IV. How can we reduce the complexity? Well, we do not need all the elements in the heap, we are always interested in the k largest ones. If we consider the example from above, we can work in the following way, always keeping just the k maximum elements up until now:

- Initially we keep 6, 12, 91
- When we get to 9, we can drop 6, because we know for sure that it is not going to be part of the 3 maximum numbers (we already have 3 numbers greater than this). So we keep 12, 91, 9.
- When we get to 3, we know it is not going to be part of the 3 maximum elements (We already have 3 elements greater than that). Similar with 5.
- When we get to 25, we can drop 9, and go on with 12, 91, 25.
- Etc.

We can keep the k elements that we consider in a heap. Should it be a min-heap or max-heap?

When we have the k largest elements at a given point, we will be interested in the minimum of these elements (this is what we compare to the new element that is considered and this is what we remove if we find a larger one) so it should be a min-heap.

```
function sumOfK2(elems, n, k) is:
//elems is an array of unique integer numbers
//n is the number of elements from elems
//k is the number of elements we want to sum up. Assume k <= n
	init(h, "≤") //assume we have the Heap data structure implemented. We
	initialize a heap with the relation "≤" (a min-heap)
	for i ← 1, k execute //the first k elements are added "by default"
		add(h, elems[i])
	end-for
	for i ← k+1, n execute
		if elems[i] > getFirst(h) then //getFirst is an operation which returns
the first element from the heap.
			remove(h)//it returns the removed element, but we do not need it
			add(h, elems[i])
		end-if
	sum ← 0
	for i ← 1, k execute
		elem ← remove(h) //remove operation was discussed at Lecture 8
		sum ← sum + elem
	end-for
	sumOfK2 ← sum
end-function
```

- Complexity? Our heap has maximum k elements, so operations have a complexity of $O(\log_2 k)$. We call add at most n times (worst case, when every element is greater than the root of the heap) and remove at most n times. So in total we have $O(n*\log_2 k)$
- If you do not use an already implemented heap, but have access to the representation, you can make the previous implementation slightly more efficient (complexity will not change though):
  - the last *for* will not have to remove elements, just simply to add up the sum of the elements from the heap array (but for this you need access to the array)
  - the middle for loop will not have to do a remove and an add. You can just overwrite the element from position 1 (this is what would be removed anyway) with the newly added element and do a bubble-down on it.

V. Can we improve complexity even more? We know that if we have an array we can transform it into a heap in a complexity O(n) – it was discussed at heapsort. So we can do something similar to version III, but instead of adding the elements one by one to the heap we could transform the array into a max-heap and then remove k elements from it. This approach has a complexity of $O(n + k*\log_2 n)$

2. Evaluate an arithmetic expression which contains single digit operands, parantheses and the _, -, *,/ operators. We assume that the expression is correct. For example:
   - 2+3*4 = 12
   - ((2+4)*7)+3*(9-5) = 54
   - 2*(4+3)-4+6/2*(1+2*7)+4 = 59

The expressions from the above example are in the so called *infix* notation. This means that operator are between the two operands (for example: 2+4). This is how we work with arithmetic expressions in general.

For a computer it is a lot easier to work in the *postfix* notation. This is a notation in which the operator comes after the two operands (for example: 2 4 +).

A few examples of expressions in the infix notation and their corresponding postfix notation:

| | |
|---|---|
| 2 + 4 | 2 4 + |
| 4*3+6 | 4 3 * 6 + |
| 4*(3+6) | 4 3 6 + * |
| (5 + 6) * (4 – 1) | 5 6 + 4 1 - * |

Obs:

- Relative order of the operands stays the same
- Relative order of the operators might change.
- We no longer have parentheses

So evaluating an arithmetic expression will have two steps:

- Transform the expression in the corresponding postfix notation
- Evaluate the expression on the postfix notation

1. Transform an infix expression in the corresponding postfix notation. Input is the infix expression, output will be the postfix notation in the form of a queue and for the transformation we use an auxiliary stack.

Steps:

- Start parsing the expression.
- If you find an operand => push it to the queue
- If you find an open paranthesis => push it to the stack
- If you find a closed paranthesis => open paranthesis should be on the stack already, so pop eveything from the stack (and whenever you pop something, push it to the queue) until you get to the first open paranthesis. Do not push the open or closed paranthesis to the stack.
- If you find an operator:
  - o As long as the stack is not empty and the top element of the stack is an operator with a priority greater than or equal to the priority of the current operator, pop from stack and push to the queue.
  - o Push operator to the stack
- When the expression is over, pop whatever you have left on the stack and push it to the queue.

Example: 2*(4+3)-4+6/2*(1+2*7)+4

| Current element | What to do | Stack (top is on the right) | Queue |
|---|---|---|---|
| 2 | Push to queue | | 2 |
| * | Push to stack | * | |
| ( | Push to stack | * ( | |
| 4 | Push to queue | | 2 4 |
| + | Push to stack | * ( + | |
| 3 | Push to queue | | 2 4 3 |
| ) | Pop from stack and push to queue until you find the open paranthesis | * | 2 4 3 + |
| - | Pop from stack as long as top has greater or equal priority.Push - to the stack | - | 2 4 3 + * |
| 4 | Push to queue | | 2 4 3 + * 4 |
| + | Pop from stack as long as top has greater or equal priority. Push + to stack | + | 2 4 3 + * 4 - |
| 6 | Push to queue | | 2 4 3 + * 4 – 6 |
| / | Push to stack | + / | |
| 2 | Push to queue | | 2 4 3 + * 4 – 6 2 |
| * | Pop from stack as long as top has greater or equal priority. Push * to stack | + * | 2 4 3 + * 4 – 6 2 / |
| ( | Push to stack | + * ( | |
| 1 | Push to queue | | 2 4 3 + * 4 – 6 2 / 1 |
| + | Push to stack | + * ( + | |
| 2 | Push to queue | | 2 4 3 + * 4 – 6 2 / 1 2 |
| * | Push to stack | + * ( + * | |
| 7 | Push to queue | | 2 4 3 + * 4 – 6 2 / 1 2 7 |
| ) | Pop from stack and push to queue until | + * | 2 4 3 + * 4 – 6 2 / 1 2 7 * + |

| | | | |
|---|---|---|---|
| | you find the open parenthesis | | |
| + | Pop from stack as long as top has greater or equal priority. Push + to the stack | + | 2 4 3 + * 4 – 6 2 / 1 2 7 * + * + |
| 4 | Push to queue | | 2 4 3 + * 4 – 6 2 / 1 2 7 * + * + 4 |
| | Pop whatever we have in the stack and push to queue | | 2 4 3 + * 4 – 6 2 / 1 2 7 * + * + 4 + |

Pseudocode implementation:

Assume stack and queue is already imeplemented and they have the standard operations: init, push, pop, top, isEmpty.

```
function transform(expression) is:
  //create stack and queue
  init(st)
  init(q)
  for e in expression execute:
    if e is operand then
      push(q, e)
    else if e is '(' then
      push(st, e)
    else if e is ') then
      while top(st) != '(') execute
        elem <- pop(st)
        push(q, elem)
      end-while
      pop(st) //to remove the (
    else //e is operand
      while (not isEmpty(st)) AND (top(st) != '(') AND
            (top(st) has higher priority than e) execute
        elem <- pop(st)
        push(q, elem)
      end-while
      push(st, e)
    end-if
  end-while
  while not isEmpty(st) execute:
    elem <- pop(st)
    push(q, elem)
  end-while
  transform <- q
end-function
```

2. Evaluate the postfix expression. Input is the postfix notation in the form of a queue (actually the output of step 1) and the output will be a value: the result of the evaluation. In the process we use an auxiliary stack.

- Start parsing the expression from the queue.
- If you find an operand => push to the stack

- If you find an operator (this is postfix notation, operators are after operands, so the operands have to be in the stack already) => pop two elements from the stack (these are the operands for this operator), perform the operation and push the result back to the stack
- When the queue is empty, the stack contains one single element, this is the result

| Current elem from queue | What to do | Stack (top is on the right) |
|---|---|---|
| 2 | Push to stack | 2 |
| 4 | Push to stack | 2 4 |
| 3 | Push to stack | 2 4 3 |
| + | Pop 2 elements, perform +, push result to stack | 2 7 |
| * | Pop 2 elements, perform *, push result to stack | 14 |
| 4 | Push to stack | 14 4 |
| - | Pop 2 elements, perform -, push result to stack | 10 |
| 6 | Push to stack | 10 6 |
| 2 | Push to stack | 10 6 2 |
| / | Pop 2 elements, perform /, push result to stack | 10 3 |
| 1 | Push to stack | 10 3 1 |
| 2 | Push to stack | 10 3 1 2 |
| 7 | Push to stack | 10 3 1 2 7 |
| * | Pop 2 elements, perform *, push result to stack | 10 3 1 14 |
| + | Pop 2 elements, perform +. push result to stack | 10 3 15 |
| * | Pop 2 elements, perform *, push result to stack | 10 45 |
| + | Pop 2 elements, perform +, push result to stack | 55 |
| 4 | Push to stack | 55 4 |
| + | Pop 2 elements, perform +, push result to stack | 59 |

Pseudocode implementation

```
function evaluate(postfix) is:
  init(st)
  while not isEmpty(postfix) execute:
    elem <- pop(postfix)
    if elem is operand then
      push (st, elem)
    else
      op1 <- pop(st)
      op2 <- pop(st)
      res<- @compute the result of op2 elem op1
      push(st, res)
    end-if
  end-while
  result <- pop(st)
  evaluate <- result
end-function
```

3. Check for Balanced Brackets in an expression (well-formedness)

Given an expression expr, write a program to examine whether the pairs and the orders of "{", "}", "(", ")", "[", "]" are correct.

```
Function areBracketsBalanced(expr)
      OK <- true
      for e in expr execute
        if (e == '(' || e == '['|| e == '{') then
            push(s, e);
        else if empty(s) then
                  OK <- false  // if empty should return
             else if e = ')' then
                    x = top(s);
                    pop(s);
                    if (x != ')') then OK <- False endif  // == '{' OR '['
                  else if e[i]= '}' then
                        x = top(s);
                        pop(s);
                        if (x != '{') then OK <- False endif  // == '(' || '[')
                        else if e[i]= ']' then
                               x = top(s);
                               pop(s);
                               if (x != '[')then OK <- False endif
                            endif
                        endif  // e= '}'

                  endif  // e = ')'

              endif // empty(s)
         endif
      endfor

      if NOT empty(s) then OK <- False endif
      areBracketsBalanced <- OK

end function
```