

Data structure and Algorithms

Presented by : Asmaa Ghonaim



Queue



Queue

- A Queue is a linear list of information that is accessed in First-In, First- Out (FIFO) order.
- The first item placed on the queue is the first item retrieved. The second item put in is the second item retrieved, and so on.
- A queue does not allow random access of any specific item.

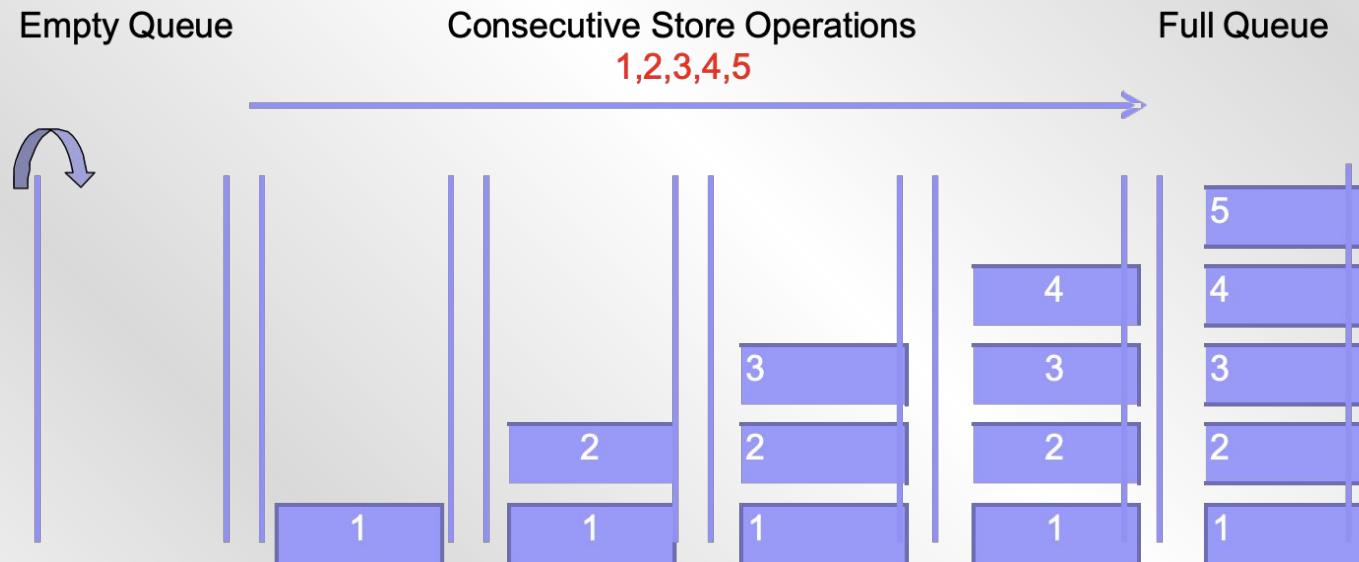
Example:

Queues are very common in everyday life. For example, lines at a bank or a fast-food restaurant are queues.

Used in many types of programming situations such as simulations, event or appointment scheduling (such as in a PERT or Gantt chart), and I/O buffering.

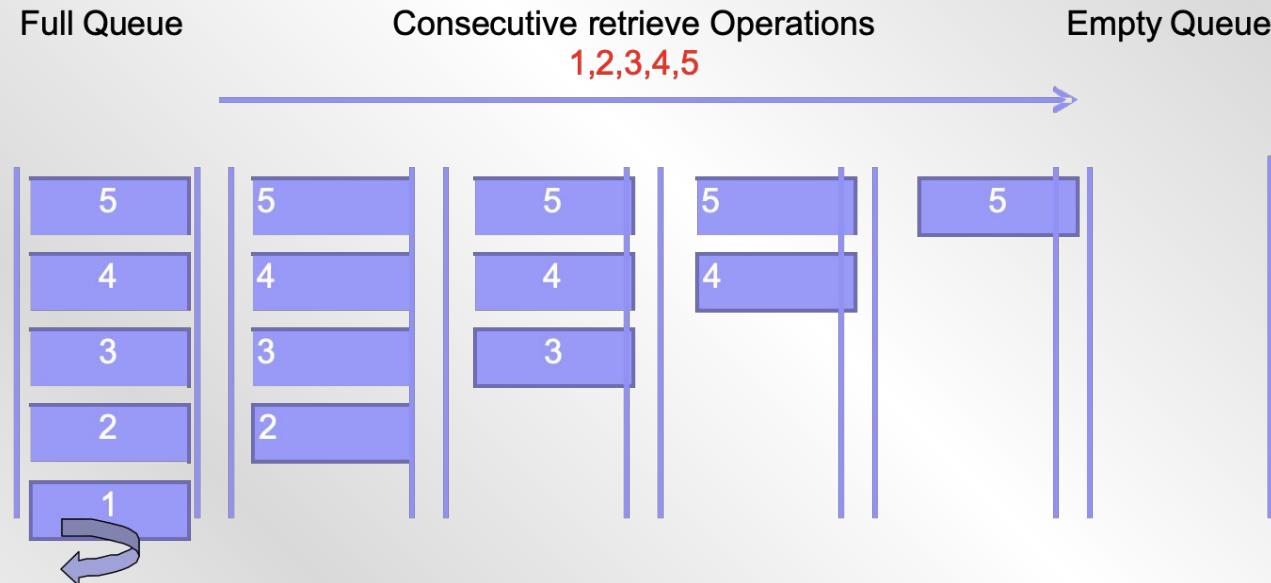
Queue

To visualize how a queue works, consider two functions: **enqueue()** function places an item onto the end of the queue



Queue

To visualize how a queue works, consider two functions: **dequeue()** function removes the first item from the queue and returns its value, if the value is not stored elsewhere, destroys it.



Queue

Queue implementation as a linked list for dynamic length:

Analyze the Linked list class first;

```
class LinkedList {  
protected:  
Employee *pStart; Employee *pEnd;  
public:  
LinkedList() { pStart=pEnd=NULL; }  
~LinkedList() { freeList(); }  
// Setters and getters for pStart and pEnd;  
void addList(Employee *pItem);           / Needed : add to the end as the queue  
void InsertList(Employee *pItem);         // Not needed here  
Employee* searchList(int Code);          // Needed  
int deleteList(int Code);                // Needed : delete element from queue  
void freeList();                        // Needed  
void displayAll();                      // Needed
```

Queue

Queue implementation as a linked list for dynamic length:

Class Queue will inherits private from Linked List

// We need only some methods as the other methods are against the concept of the queue.

```
class Queue : private LinkedList {
    // No need for size, front, rear and Employee as the size is dynamic
public:
    Queue (): LinkedList() {

    } //No need for the constructor which take size
    ~Queue () {
    }
    void enqueue (Employee *e){
        addList(e);           // or LinkedList :: addList(e);
    }
}
```

Queue

Queue implementation as a linked list for dynamic length:

```
Employee* dequeue (){  
    Employee *pNode;  
    pNode = pStart;  
    if(pStart){  
        if(pStart == pEnd){ // there is only node  
            pStart = pEnd =NULL;  
        }  
        else { // first elment  
            pStart = pStart -> pNext;  
            pStart -> pPrevious = NULL;  
        }  
    }  
    return pNode;  
}
```

Queue

Queue implementation as a linked list for dynamic length:

```
Employee* searchQueue(int Code){  
    return searchList(Code);  
}  
  
int DeleteQueue(int Code){  
    return DeleteList(Code);  
}  
void freeQueue(){  
    freeList();  
}  
void displayAll(){  
    LinkedList::displayAll();  
}  
};
```

Stack



Stack

A Stack is the opposite of a Queue because it uses Last-In, First-Out (LIFO) Accessing.

- This means elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.
- Insert: **PUSH**
- Remove: **POP**
- The accessible element is called **TOP**.
- Like the queue, the retrieval function takes a value off the list and, if the value is not stored elsewhere, destroys it.

Stack

Example:

- Imagine a stack of plates. The bottom plate in the stack is the last to be used, and the top plate (the last plate placed on the stack) is the first to be used.
- Stacks are used a great deal in system software including compilers and interpreters. In fact, C uses the computer's stack when passing arguments in functions.

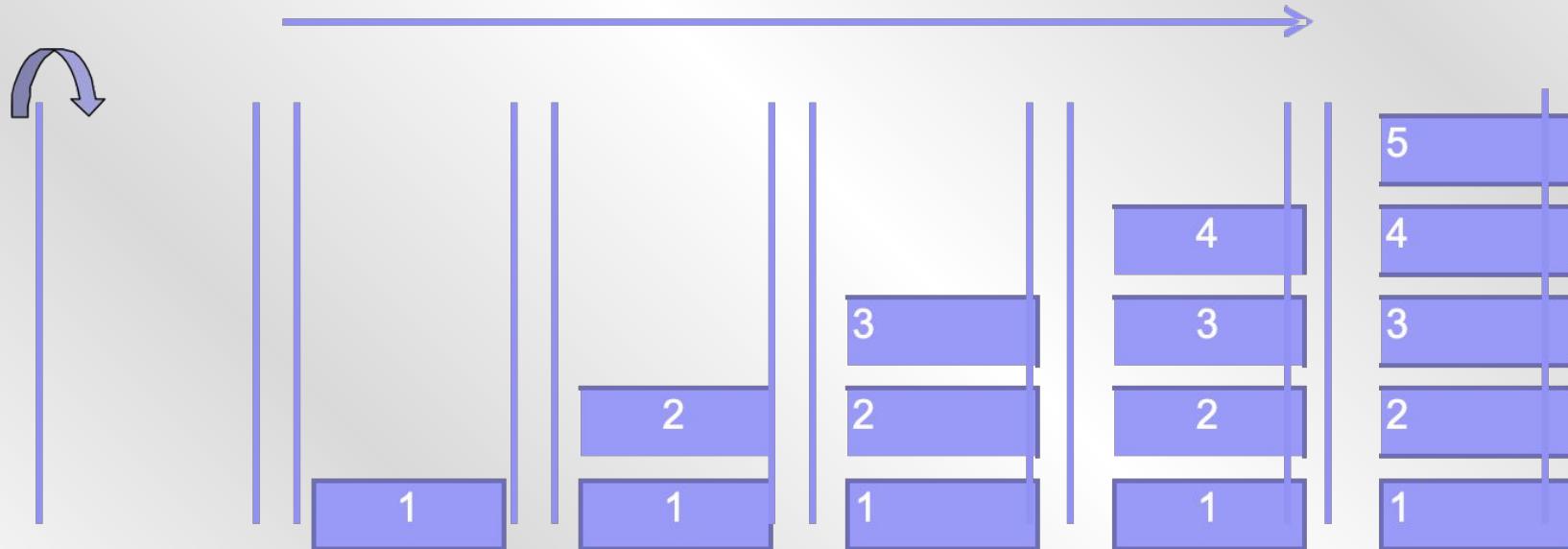


Stack

Empty Stack

Consecutive Push Operations
1,2,3,4,5

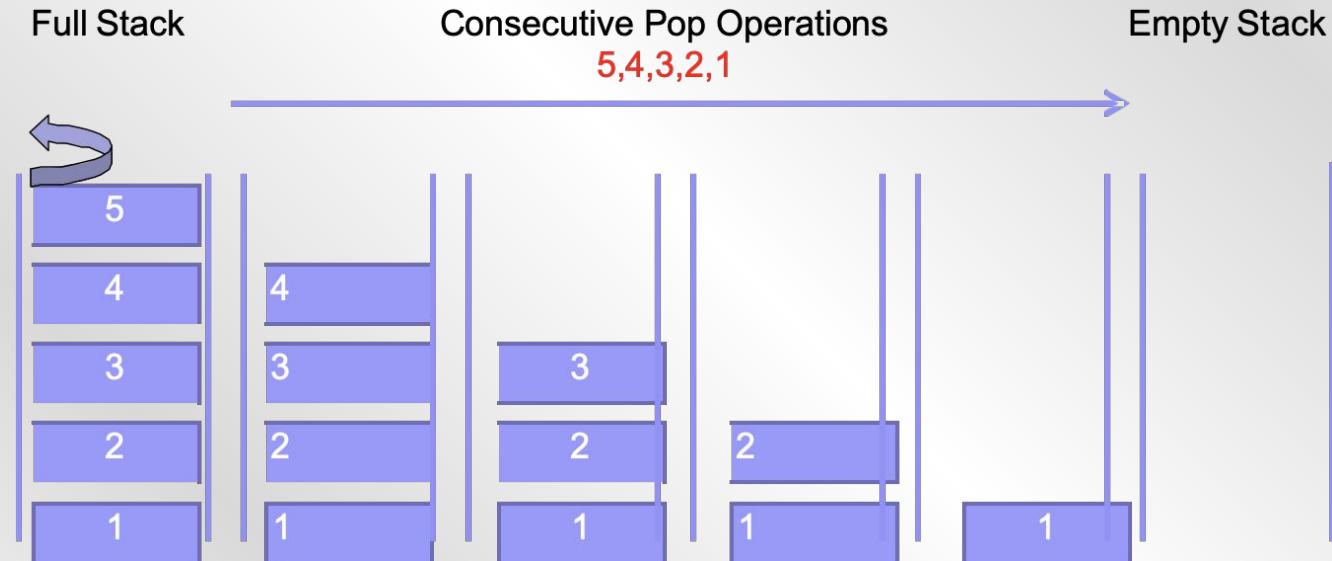
Full Stack



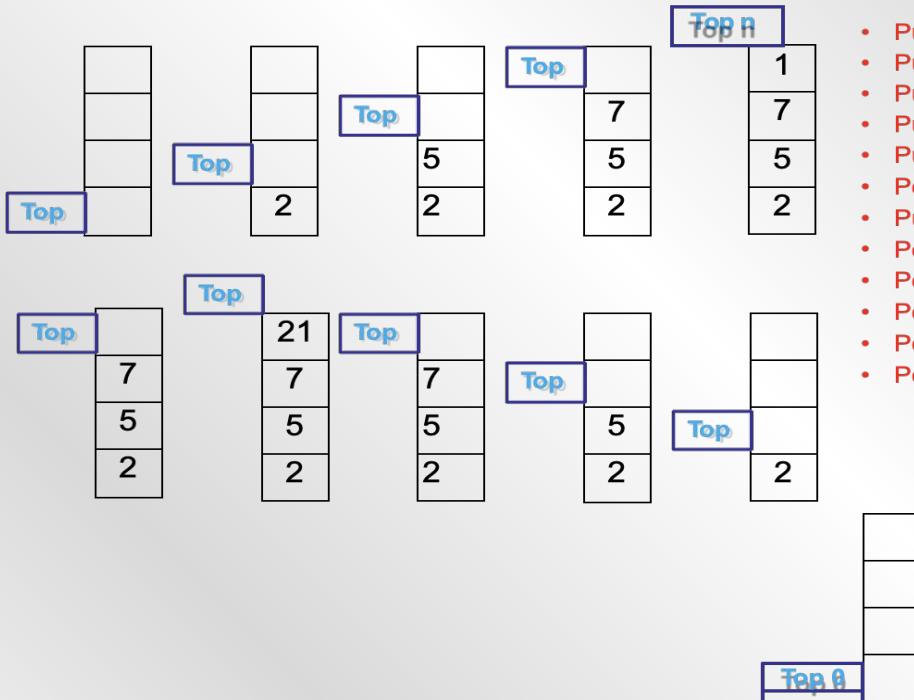
Stack

To visualize how a Stack works, consider two functions:

- **Pop** (Retrieve) Operation : Pop operation retrieves a value from the stack.
-



Stack



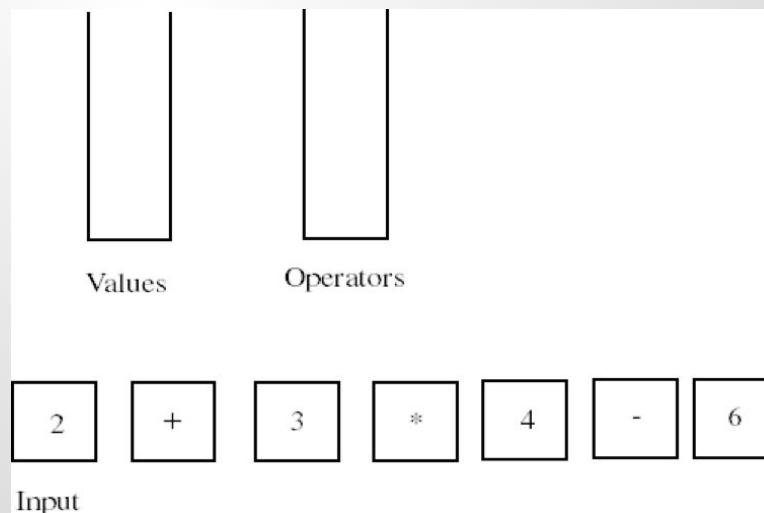
- Push (2)
- Push (5)
- Push (7)
- Push (1)
- Push (3) // stack is full 3 not added
- Pop()→ 1
- Push (21)
- Pop()→21
- Pop()→7
- Pop()→5
- Pop()→2
- Pop()→ // stack is empty

Evaluating Arithmetic Expressions

Operator precedence:

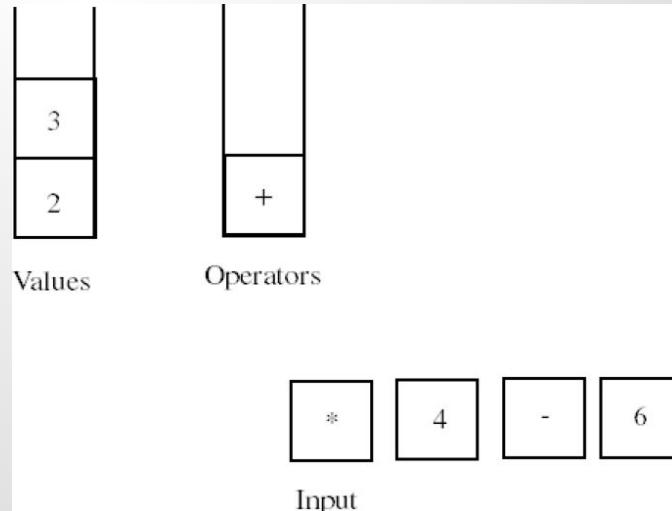
- * has precedence over +/–
- operators of the same precedence group evaluated from left to right \

$2 + 3 * 4 - 6$



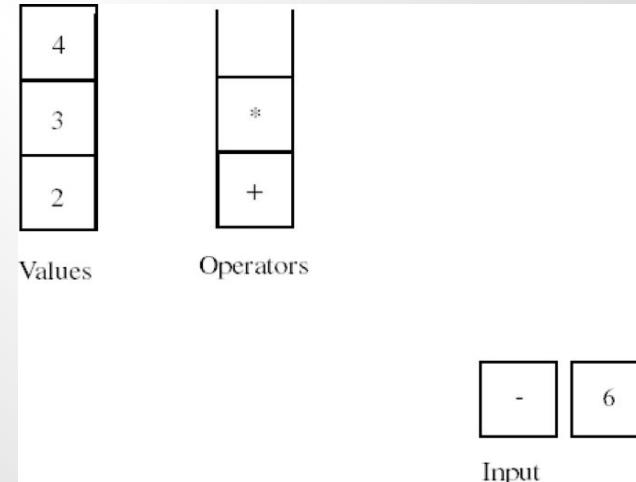
Evaluating Arithmetic Expressions

- $2 + 3 * 4 - 6 = 8$
- After handling the first three tokens the next operator in the input has a higher precedence than the operator at the top of the operator stack, so we push the next operator. The next number token also gets pushed on the value stack.



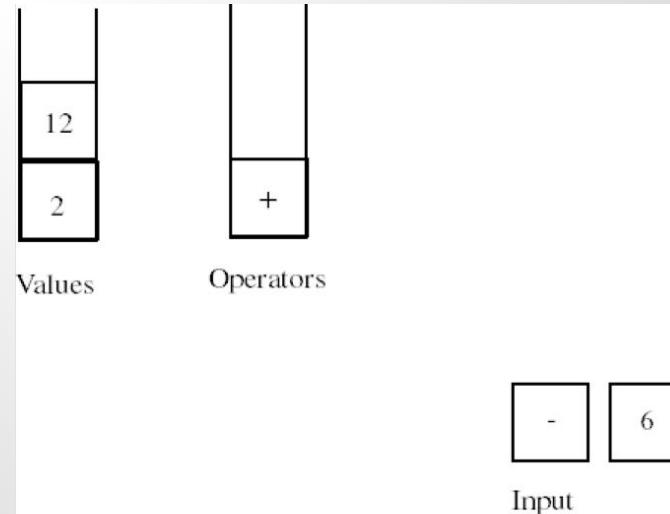
Evaluating Arithmetic Expressions

- $2 + 3 * 4 - 6 = 8$
- The next operator in the input sequence has a precedence lower than that of the operator at the top of the operator stack. This causes us to process and remove the operator at the top of the operator stack. **3 * 4**



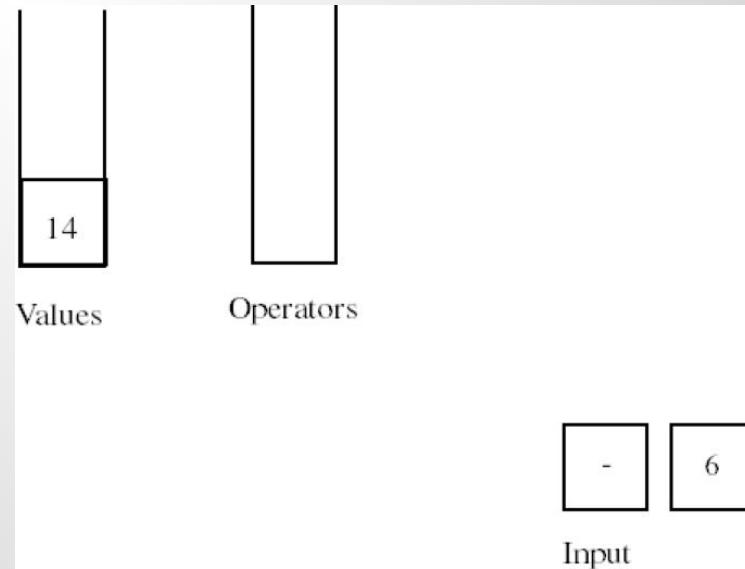
Evaluating Arithmetic Expressions

- $2 + 3 * 4 - 6 = 8$
- Once again, the operator in the input has a precedence equal to that of the operator at the top of the operator stack, so we process and remove the operator from the operator stack. **2 + 12**



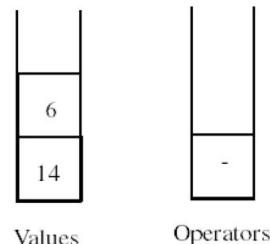
Evaluating Arithmetic Expressions

- $2 + 3 * 4 - 6 = 8$
- At this point the operator stack is empty, so the operator token at the front of the input gets pushed on the operator stack. The number token at the end of the input gets pushed on the value stack.



Evaluating Arithmetic Expressions

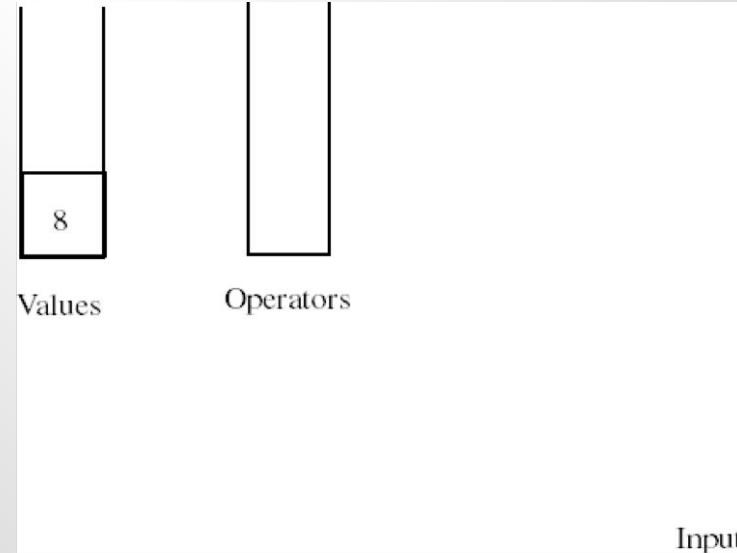
- $2 + 3 * 4 - 6 = 8$
- Once the input has emptied out, we process any operators that remain on the operator stack. Once all of those operators have been processed, the sole remaining number on the value stack is the result of the computation. **14 - 6**



Input

Evaluating Arithmetic Expressions

- $2 + 3 * 4 - 6 = 8$



Try **$2 + 4 - 5 * 10 / 2 + 1$** ?

Evaluating Arithmetic Expressions

The basic algorithm:

1. Make use of **two** TokenStacks [a *value stack* ,*an operator stack*].
2. If character is **operand** push on the operand stack.
3. Else if character is **operator**
 1. While the top of the operator stack is not of smaller precedence than this character.
 2. **Pop operator** from operator stack.
 3. **Pop** two **operands** (op1 and op2) from operand stack.
 4. Store **op1 op op2** on the operand stack back to 2.1.
4. Else (no more character left to read):
 5. **Pop operators** until operator stack is not empty.
 6. **Pop** top 2 **operands** and push op1 op op2 on the operand stack.
 7. return the top value from operand stack.

Lab Exercise

Assignments :

- Use Stack to evaluate this operation **$3 + 4 / 2 + 2 * 12 - 6$**
- Implement stack using linked list