Contents

[Elementary Data structures 2](#_Toc54732064)

[ Arrays 2](#_Toc54732065)

[ Queues 2](#_Toc54732066)

[ Stacks 3](#_Toc54732067)

[ Linked List 4](#_Toc54732068)

[Mathematical evaluation 4](#_Toc54732069)

[ Infix 4](#_Toc54732070)

[ Prefix 4](#_Toc54732071)

[ Postfix 4](#_Toc54732072)

[ Infix to Postfix conversion 4](#_Toc54732073)

[ Check parentheses correction 5](#_Toc54732074)

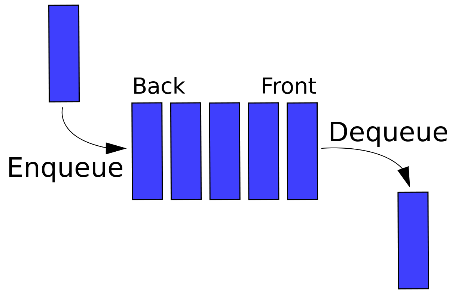
* Keivan (:

# Elementary Data structures

## Arrays

|  |  |
| --- | --- |
| Naming convention | *Address of the starting node (2000, 2004, 2008, …, 2036 for an array of 10 32-bit integer values)* |
| Addressing |  |
| Good aspects | *Easy access, Time complexity of O(1)* |
| Bad aspects | Fixed size, Fixed type |

## Queues

First in First out (FIFO)

Enqueue (Insert): Items are added to the tail of the queue

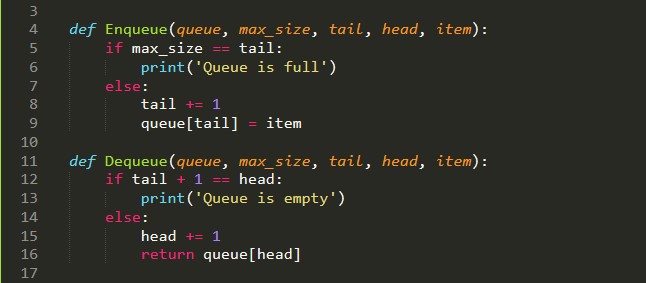
Dequeue (Delete): Items are deleted from the head of the queue

|  |  |  |  |
| --- | --- | --- | --- |
| items, linear queue | HEAD | Tail | Condition |
| Zero | 1 | 0 | TAIL + 1 = Head |
| One | 1 | 1 | Head = TAIL |
| More than one | - | Max size | TAIL = Max size |

Q. tail: Stores index of the latest added item

Q. head: Stores the index of the earlies item added

Implementation of Enqueue & Dequeue



When empty queue is to be Dequeue, we receive an *Underflow* error.

When full queue is to be Enqueue, we receive an *Overflow* error.

Problem with linear queues is that they can be used only once because *Head* reaches *Tail* and insert will be impossible. Solution? Circular queue!

Note: **in circular queue, condition of filled queue is equal to empty queue (TAIL + 1 = Head).**

## Stacks

Last in, First out (LIFO)

Push (Insert): Items are added to the stack.

Pop (Delete): Items are deleted from the stack.

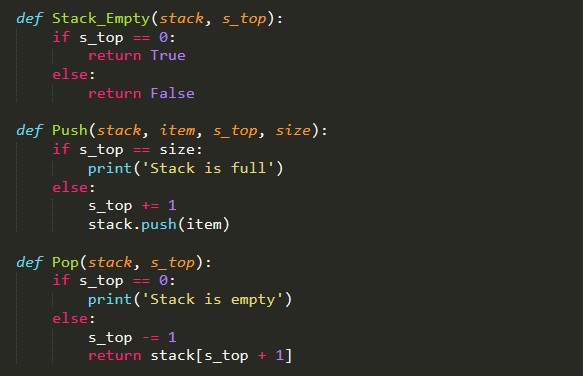
S. top: Stores index of the last item added to the stack.

Stack-Empty: Is a function that checks whether stack is empty or not.

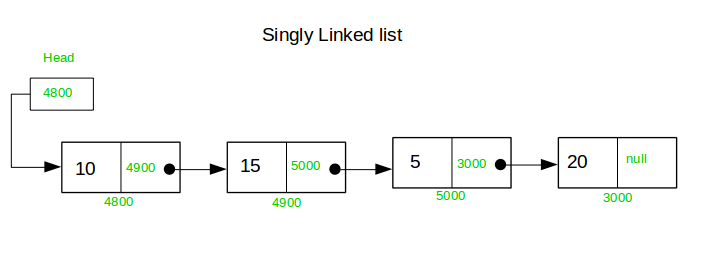
When empty stack is to be popped, we receive an *Underflow* error.

When full stack is to be pushed, we receive an *Overflow* error.

When (*S. top* == 1), stack is empty.

Implementation of Stack Empty, Push & Pop

## Linked List



Information in memory are not in physically in a sequence, rather they are linked using pointers.

Good aspects: Dynamic, Hardly no limit or boundaries

Bad aspects: Time consuming, Time complexity of O(n)

Doubly linked lists: Nodes contain three attributes:

* Value
* Next (Pointer)
* Previous (Pointer)

Circular linked lists: Last node contains a pointer to the first node

# Mathematical evaluation

## Infix

Form: A + B => expression + operator + expression

Example: A +

Note: **Requires use of parentheses in order to clarify the expression orders.**

## Prefix

Form: +A B => operator + expression + expression

Example:

## Postfix

Form: A B + => expression + expression + operation

Example:

Example:

## Infix to Postfix conversion

We use *Stack* to convert *Infix* to *Postfix* for further evaluations. Process:

* Create empty stack
* Process expressions from left to right and check tokens (expressions, operators, parenthesizes)
  + If token is open parenthesis, add to stack
  + If token is expression, add to output
  + If token is operator, and if its priority is higher than the top element of the stack, add to stack. Otherwise, add the top stack item to the output and replace the item with the operator. Continue at as needed.
  + If above scenarios are not the case, add to stack
  + If token is close parenthesis, remove all items from stack and add to output until open parenthesis is reached. (Open parenthesis is removed from stack but NOT added to output)
* When Infix processing is finished, all that remain in stack are added to output.

Example:

Solution:

## Check parentheses correction

Wrong form: )) ((, ())(

Correct form: (((a + b) \* c) / d)

Process: While traversing from left to right, each close parenthesis we reach, there must be an open parenthesis before it.

# Linked Lists

For full implementation of doubly-linked-list visit my [GitHub](https://github.com/keivanipchihagh/AUT-CE/blob/main/Algorithms%20%26%20Data%20Structure%20(DS)/Data%20Structures/DoublyLinkedList.py).

# Binary Tree

For full implementation of binary-tree visit my [GitHub](https://github.com/keivanipchihagh/AUT-CE/blob/main/Algorithms%20%26%20Data%20Structure%20(DS)/Data%20Structures/BinaryTree.py).

# Heap

Almost a complete (binary) tree data structure.

* Max Heap

Parent’s value is equal or greater than child’s value. Maximum value is always the **root**.

Rules are as followed:

* Left(i) =
* Right(i) =
* Parent(i) =
* Min Heap

Parent’s value is equal or less than child’s value.