Introduction to Computing

Data Structures - Stacks, Queues, Linked Lists

Malay Bhattacharyya

Associate Professor

MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

November, 2024

1 Basics

2 Types of Data Structures

3 Stacks
- Implementing a Stack
- Applications of Stack

4 Queues
- Implementing a Queue
- Applications of Queue

5 Linked Lists
- Implementing a Linked List

Basics

A data structure is a logical organization of data

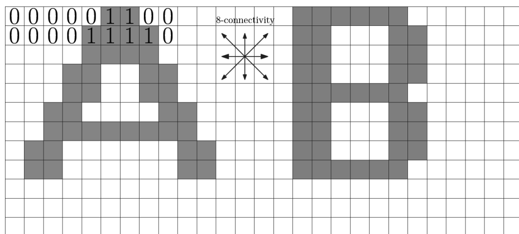
- A data structure can accommodate one or more data elements (storage).
- The logical organization defines
  - how the elements will get added (insertion)?
  - how the elements will get removed (deletion)?
  - how the elements will be browsed (traversal)?

Types of Data Structures

- Linear
  - Stack
  - Queue
  - Linked List
- Nonlinear
  - Tree
  - Graph

Motivating example

Problem: Find the *connected components* in an image.



Data structure: Stack (LIFO)

Operations:

- `PUSH`: Insert at the top
- `POP`: Remove and return element from the top
- `DISPLAY`: Show the elements
- `LENGTH` or `HEIGHT`
- `ISEMPTY`, `ISFULL`

Initialization

Considering the stack size as static:

```
StackDS = []
SizeStackDS = 10
```

Push operation

```
def StackPUSH(Data, StackDS):
    if len(StackDS) < SizeStackDS:
        StackDS.append(Data)
    else:
        print("Stack overflow!!!")
```

Pop operation

```
def StackPOP(StackDS):
    if len(StackDS) > 0:
        Data = StackDS.pop()
        return Data
    else:
        print("Stack underflow!!!")
```

Note: If you do not wish to receive the popped element, exclude return statement.

Displaying the elements

```
def StackDISPLAY(StackDS):
    print("The elements in the stack are: ")
    for Element in StackDS:
        print(Element)
```

Note: You can directly write `print(StackDS)`.

Infix, Prefix and Postfix expression

Infix expression:

Operand1 Operator Operand2

Prefix expression:

Operator Operand1 Operand2

Postfix expression:

Operand1 Operand2 Operator

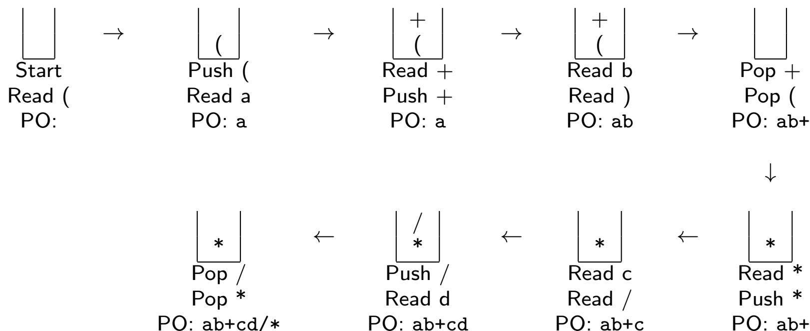
Converting Infix to Postfix expression

1. Create an empty stack and an empty postfix expression (say `PO`).
2. Iterate through the infix expression from left to right and append operands to the postfix expression.
3. If an operator is encountered, pop operators from the stack and append them to the postfix expression until an operator with lower or equal precedence is found.
4. Push the current operator onto the stack.
5. If a left parenthesis is encountered, push it onto the stack.
6. If a right parenthesis is encountered, pop operators from the stack and append them to the postfix expression until a left parenthesis is found.
7. Pop any remaining operators from the stack and append them to the postfix expression.
8. Return the postfix expression

Converting Infix to Postfix expression

Input:

(a + b) * c / d



Output:

a b + c d / *

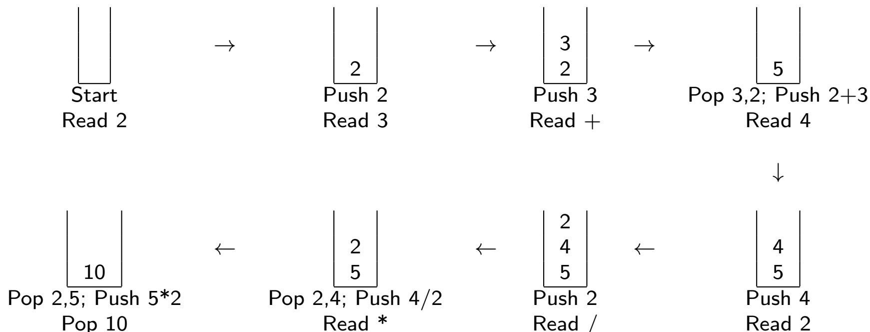
Evaluating Postfix expression

1. Create a stack to store operands or values
2. Scan the expression from left to right
3. If the element is an operand, push it onto the stack.
4. If the element is an operator, pop two operands from the stack.
5. Evaluate the operands using the operator
6. Push the result back onto the stack
7. After finishing the scanning of the expression, return the value on the stack.

Evaluating Postfix expression

Input:

2 3 + 4 2 / *



Output:

10

Depth-first Search (DFS)

We can perform depth-first search on a graph given in the form of adjacency list in a file. The DFS algorithm works as follows:

i) Keep any one of the graph's vertices on top of a stack.

ii) Pop out the top data item from the stack and add it to the Visited list.

iii) Create a list of that vertex's adjacent nodes. Add the ones which are not in the Visited list to the top of stack.

iv) Repeat steps (ii)-(iii) until the stack is empty.

Implementing the DFS

```
# Adjacency list defined as a dictionary
Graph = {
    '0' : ['1','2'], '1' : ['3', '4'], '2' : ['5'],
    '3' : [], '4' : ['5'], '5' : []
}
Visited = [] # Array to keep track of visited vertices
def DFS(Visited, Graph, Vertex):
    if Vertex not in Visited:
        Visited.append(Vertex)
        print(Visited[len(Visited) - 1]) # Top of stack
        for Adjacent in Graph[Vertex]:
            DFS(Visited, Graph, Adjacent)
DFS(Visited, Graph, '0') # Function call with vertex '0'
```

Motivating example

Problem: Maintain a list of patients waiting to consult a doctor.
Data structure: Queue (FIFO)
Operations:

- `INSERT` or `ENQUEUE`: insert at the end
- `DELETE` or `DEQUEUE`: remove and return element from the front
- `DISPLAY`: Show the elements
- `LENGTH` or `SIZE`
- `ISEMPTY`, `ISFULL`

Standard implementation – Initialization

Considering the queue size as static:

```
QueueDS = []
SizeQueueDS = 10
```

Standard implementation – Insert operation

```
def QueueINSERT(QueueDS, Data):
    if len(QueueDS) < SizeQueueDS:
        QueueDS.insert(0, Data)
    else:
        print("Queue overflow!!!")
```

Standard implementation – Delete operation

```
def QueueDELETE(QueueDS):
    if len(QueueDS) > 0:
        Data = QueueDS.pop()
        return Data
    else:
        print("Queue underflow!!!")
```

Note: If you do not wish to receive the deleted element, exclude the return statement.

Standard implementation – Displaying the elements

```
def QueueDISPLAY(QueueDS):
    print("The elements in the queue are: ")
    for Element in QueueDS:
        print(Element)
```

Note: You can directly write `print(QueusDS)`.

Efficient implementation – Initialization

Considering the queue size as static:

```
QueueDS = []
SizeQueueDS = 10
```

Efficient implementation – Insert operation

```
def QueueINSERT(QueueDS, Data):
    global rear
    if (rear + 1) % SizeQueueDS == front:
        print("Queue overflow!!!")
    else:
        QueueDS[rear] = Data
        rear = (rear + 1) % SizeQueueDS
```

Efficient implementation – Delete operation

```
def QueueDELETE(QueueDS):
    global front, rear, SizeQueueDS
    if front == rear:
        print("Queue underflow!!!")
    else:
        Data = QueueDS[front]
        front = (front + 1) % SizeQueueDS
        return Data
```

Note: If you wish to receive the deleted element, include a return statement.

Efficient implementation – Displaying the elements

```
def QueueDISPLAY(QueueDS):
    print("The elements in the queue are: ")
    if front <= rear:
        print(QueueDS[front:rear])
    else:
        print(QueueDS[front:] + QueueDS[:rear])
```

Note: You cannot directly write `print(QueusDS)`.

Breadth-first Search (BFS)

We can perform breadth-first search on a graph given in the form of adjacency list in a file. The BFS algorithm works as follows:

i) Insert any one of the graph's vertices in the front of a queue.

ii) Delete the rear data item from the queue and add it to the `Visited` list.

iii) Create a list of that vertex's adjacent nodes. Add the ones which are not in the `Visited` list to the front of the queue.

iv) Repeat steps (ii)-(iii) until the queue is empty.

Implementing BFS

```
# Adjacency list defined as a dictionary
Graph = {
    '0' : ['1','2'], '1' : ['3', '4'], '2' : ['5'],
    '3' : [], '4' : ['5'], '5' : []
}
Visited = [] # Array to keep track of visited vertices
def BFS(Visited, Graph, Vertex):
    if Vertex not in Visited:
        Visited.insert(0, Vertex)
        print(Visited[0]) # Front of queue
        for Adjacent in Graph[Vertex]:
            BFS(Visited, Graph, Adjacent)
BFS(Visited, Graph, '0') # Function call with vertex '0'
```

Motivating example

Problem: Maintain a database of student information for the class teacher: roll number, name, stream, courses taken, etc.
Data structure: Linked list (AIAO)
Operations:

- `INSERT`
  - at the beginning, end, other position (by index)
  - before / after a specific element
- `DELETE`
  - at the beginning, end, other position (by index)
  - before / after a specific element
- `LENGTH`: number of elements in the list
- `GET`: return the value of an item by index
- `SEARCH`: lookup
- `ITERATE` or `FOREACH`: to do something
- `SORT`: on a specified attribute

Standard implementation – Initialization

Considering the linked list size as static:

```
LinkedListDS = []
SizeLinkedListDS = 10
```

Standard implementation – Insert operation

```
def LinkedListINSERT(QueueDS, Data, Index):
    if len(LinkedListDS) < SizeLinkedListDS:
        LinkedListDS.insert(index, Data)
    else:
        print("Linked List overflow!!!")
```

Standard implementation – Delete operation

```
def LinkedListDELETE(LinkedListDS, Index):
    if len(QueueDS) > 0:
        Data = LinkedListDS.remove(Index)
        return Data
    else:
        print("Linked List underflow!!!")
```

Note: If you do not wish to receive the deleted element, exclude the return statement.

Standard implementation – Displaying the elements

```
def LinkedListDISPLAY(LinkedListDS):
    print("The elements in the linked list are: ")
    for Element in LinkedListDS:
        print(Element)
```

Note: You can directly write `print(LinkedListDS)`.