LISTS

An ADT

- A set of objects
- A set of operations

- Same set of objects, different sets of operations \(\rightarrow\) different ADTs

- ADTs are implemented using classes, hiding implementation details: \textit{encapsulation}

List Abstraction

- Definition:
  - A linear configuration of elements, called nodes.
Characteristics

- Insert and delete nodes in any order
- The nodes are connected
- Each node has two components
  - Information (data)
  - Link to the next node
- The nodes are accessed through the links between them

Predecessor/Successor

- For each node the node that is in front of it is called predecessor
- The node that is after it is called successor

Terminology

- Head (front, first node):
  - The node without any predecessor, the node that starts the lists
- Tail (end, last node):
  - The node that has no successor, the last node in the list
- Current node: The node being processed.
  - From the current node, we can access the next node
- Empty list: No nodes exist

Basic operations

- To create/destroy a list
- To expand/shrink the list
- Read/Write operations
- Changing the current node (moving along the list)
- To report current position in the list
- To report status of the list
ADT List Notation

L - list
e - item of the same type as the information part of an element (a node) in the list
b - boolean value

Operations in ADT Notation

Insert(L,e)
• Inserts a node with information e before the current position

Delete(L)
• Deletes the current node in L, the current position indicates the next node.

RetrieveInfo(L) → e
• Returns the information in the current node.

Insertion and Deletion

A. Insertion
To insert a node X between the nodes A and B:
- Create a link from X to B
- Create a link from A to X

Insertion
Insertion and Deletion

B. Deletion
To delete a node X between A and B:
- Create a link from A to B
- Remove node X

Node Linking

1. Single linked lists:
   Each node contains a link only to the next node.
2. Doubly linked lists:
   Each node contains two links - to the previous and to the next node.
3. Circular lists:
   The tail is linked to the head.

Deletion

List Implementation

- Static - using an array
- Dynamic - using linear nodes
Array Implementation

Two parallel arrays are used:

- **Index array** - the number stored in the \( i \)th element shows the index of the "next" node, i.e. node that follows the \( i \)th node

- **Data array** - used to store the informational part of the nodes.

Stacks

- **Definition:**
  - The last stored element is the first to be accessed
  - (LIFO: last in - first out)
Basic operations

• **Push:** Store a data item at the top of the stack

• **Pop:** Retrieve a data item from the top of the stack

ADT Definition of STACK

• Notation:
  
  - $s$ stack
  - $e$ item of same type as the elements of $S$
  - $b$ boolean value

Operations

**Init_Stack(S)**
Procedure to initialize $S$ to an empty stack

**Destroy_Stack(S)**
Procedure to delete all elements in $S$

Operations

**Stack.Empty(S) $\to b$**
Boolean function that returns TRUE if $S$ is empty.

**Stack.Full(S) $\to b$**
Boolean function that returns TRUE if $S$ is full.
**Operations**

**Push(S,e)**
Procedure to place an item $e$ into $S$ (if there is room, i.e. $S$ is not full)

**Pop(S) → e**
Procedure to take the last item stored in $S$ (this item is called also - top element) if $S$ is not empty

**Example**

- A procedure to replace the elements of a nonempty stack, assuming they are of type integers, with their sum.

  - **Pre**: A nonempty stack with elements of type integers.
  
  - **Post**: $S$ contains only one element - the sum of previously stored elements.

**Algorithm**

1. $e_1 \leftarrow \text{Pop}(S)$
2. while stack is not empty repeat
   2.1. $e_2 \leftarrow \text{pop}(S)$
   2.2. push($S$, $e_1 + e_2$)
   2.3. $e_1 \leftarrow \text{pop}(S)$
3. push($S$, $e_1$)

** QUEUES **
Queues

- **Definition**: A sequence of elements of the same type.
- The first stored element is first accessible.
- The structure is known also under the name FIFO - first in first out or FCFS - first come first served

Basic operations

- **EnQueue**: store a data item at the end of the queue
- **DeQueue**: retrieve a data item from the beginning of the queue

ADT Definition of QUEUE

- **Notation**:
  
  - \( Q \) : queue
  - \( e \) : item of same type as the elements of \( Q \)
  - \( b \) : boolean value

Operations

- **Init_Queue(Q)**
  - Initialize \( Q \) to an empty queue

- **Queue_Empty(Q) \( \rightarrow b \)**
  - Boolean function that returns TRUE if \( Q \) is empty

- **Queue_Full(Q) \( \rightarrow b \)**
  - Boolean function that returns TRUE if \( Q \) is full: array-based implementations
### Operations

**EnQueue(Q, e)**
- Procedure to place an item $e$ into $Q$ at the end (if $Q$ is not full)

**DeQueue(Q) → e**
- Procedure to take the first item stored in $Q$ if $Q$ is not empty

### Problem 1

- **Append_Queue(Q, P):** A procedure to append a queue $P$ onto the end of a queue $Q$, leaving $P$ empty.
- **Pre:** queue $P$ and queue $Q$, initialized
- **Post:** $Q$ contains all elements originally in $Q$, followed by the elements that were in $P$ in same order. $P$ is empty.
- Design an algorithm to solve the problem

### Problem 2

- **Reverse_Queue(Q):** A procedure to reverse the elements in a queue $Q$
- **Pre:** queue $Q$, initialized
- **Post:** $Q$ contains all elements re-written in reverse order

- Design a non-recursive algorithm using a stack
- Design a recursive algorithm
- Find the complexity of the algorithms

### Solutions to Problem 2: A. Non-recursive

**Init_Stack(S)**

While not Queue_Empty(Q)
- $e \leftarrow$ DeQueue(Q)
- Push(S, e)

While not Stack_Empty(S)
- $e \leftarrow$ Pop(S)
- EnQueue(Q, e)

Complexity $O(N)$

$N$ - the number of elements in $Q$
Solutions to Problem 2:  
B. Recursive  

Reverse_Queue(Q):

If not Queue_Empty(Q)  
  e ← DeQueue(Q)  
  Reverse_Queue(Q)  
  EnQueue(Q,e)  
return

Complexity $O(N)$  

N - the number of elements in Q

Problem 3

• Append_Reverse_Queue(Q,P): Append a queue P in reverse order onto the end of a queue Q, leaving P empty.

• Pre: P and Q initialized (possibly empty)

• Post: Q contains all elements originally in Q, followed by the elements that were in P in reverse order. P is empty

• Design a recursive algorithm