Chapter 3 - STACK

- Definition of Stack
- Specifications for Stack
- Implementations of Stack
- Linked Stack
- Contiguous Stack
- Applications of Stack
Linear List Concepts

- Linear lists
  - General
    - Unordered
    - Ordered
  - Restricted
    - FIFO (queue)
    - LIFO (Stack)

CuuDuongThanCong.com
https://fb.com/tailieudientucntt
Stack ADT

**DEFINITION:** A Stack of elements of type T is a finite sequence of elements of T, in which all insertions and deletions are restricted to one end, called the top.

Stack is a Last In - First Out (LIFO) data structure.

**Basic operations:**
- *Construct* a stack, leaving it empty.
- *Push* an element.
- *Pop* an element.
- *Top* an element.
Basic operation of Stack (Push)

Before

After

a) **Successful operation**: function returns *success*

b) **Unsuccessful operation**: function returns *overflow*
Basic operation of Stack (Pop)

**Before**

- **top**
  - pop data

**After**

- **top**
  - pop data

**a) Successful operation**: function returns *success*

**b) Unsuccessful operation**: function returns *underflow*

(Stack remains unchanged)
Basic operation of Stack (Top)

Before

top

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After

Received data: X

Stack remains unchanged

a) **Successful operation**: function returns *success*

b) **Unsuccessful operation**: function returns *underflow*
Stack ADT (cont.)

Extended operations:

- Determine whether the stack is *empty* or not.
- Determine whether the stack is *full* or not.
- Find the *size* of the stack.
- *Clear* the stack to make it empty.
- Determine the total number of elements that have ever been placed in the stack.
- Determine the average number of elements processed through the stack in a given period.
- …
Specifications for Stack ADT

<void> Create()

<ErrorCode> Push (val DataIn <DataType>)

<ErrorCode> Pop ()

<ErrorCode> Top (ref DataOut <DataType>)

<boolean> isEmpty ()

<boolean> isFull ()

<integer> Size () // the current number of elements in the stack.

Variants of similar methods:

ErrorCode Pop (ref DataOut <DataType>)

…
Built a Stack ADT

Stack may be fully inherited from a List ADT, inside its operations calling List’s operations.

Ex.:

```cpp
<ErrorCode> Push (val DataIn <DataType>)
// Call List::InsertHead(DataIn)
    or
// Call List::Insert(DataIn, 0) // 0: insert to the 1st position
end Push
```

```cpp
<ErrorCode> Pop ()
// Call List::RemoveHead()
end Pop
```

Other operations of Stack are similar …
Built a List ADT from Stack ADT

If the Stack ADT has been built first, List ADT may be inherited from the Stack. Some of its operations call Stack's operations; the others will be added.
Implementations of Stack

- **Contiguous Implementation**: use an array.  
  (May be Automatically or Dynamically Allocated Array)

- **Linked Implementation**: linked stack.
Linked Stack

a) Conceptual

Node
Data <DataType>
link <pointer>
end Node

Stack
top <pointer>
count <integer>
end Stack

b) Physical
Create Linked Stack

<void> Create ()

Creates an empty linked stack.

Pre none

Post An empty linked stack has been created.

1. top = NULL
2. count = 0
3. return
end Create
Push data into a Linked Stack

1. Allocate memory for the new node and set up data.

2. Update pointers and count:
   - Point the new node to the top node.
   - Point top to the new node.

\[
\begin{align*}
\text{pNew->link} &= \text{top} \\ 
\text{top} &= \text{pNew} \\
\text{count} &= \text{count} + 1
\end{align*}
\]
Push data into a Linked Stack (cont.)

• Push is successful when allocation memory for the new node is successful.

• There is no difference between push data into a stack having elements and push data into an empty stack (top having NULL value is assigned to pNew->link: that’s corresponding to a list having only one element).
Push Algorithm (cont.)

<Error Code> Push (val DataIn <DataType>)
Pushes new data into the stack.

Pre  DataIn contains data to be pushed.
Post  If stack is not full, DataIn has been pushed in; otherwise, stack remains unchanged.
Return success or overflow.
Push Algorithm (cont.)

<ErrorCode> Push (val DataIn <DataType>)

// For Linked Stack
1. Allocate pNew
2. If (allocation was successful)
   1. pNew->data = DataIn
   2. pNew->link = top
   3. top = pNew
   4. count = count + 1
   5. return success
3. Else
   1. return overflow
end Push
Pop Linked Stack

1. \texttt{pDel} holds the element on the top of the stack.

2. \texttt{top} points to the next element.

3. Recycle \texttt{pDel}. Decrease \texttt{count} by 1.
Pop Linked Stack (cont.)

- Pop is successful when the stack is not empty.
- There is **no difference** between pop an element from a stack having elements and pop the only-remained element in the stack (*pDel-*->link having NULL value is assigned to top: that’s corresponding to an empty stack).
Pop Algorithm

<ErrorCode> Pop()

Pops an element from the top of the stack

Pre none

Post If the stack is not empty, the element on the top has been removed; otherwise, the stack remains unchanged.

Return success or underflow.
Pop Algorithm (cont.)

<ErrorCode> Pop()

Pops an element from the top of the stack

// For Linked Stack

1. If (count > 0)
   1. pDel = top
   2. top = pDel->link
   3. recycle pDel
   4. count = count - 1
   5. return success

2. else
   1. return underflow

3. end Pop
<ErrorCode> Top (ref DataOut <DataType>)
Retrieves data on the top of the stack without changing the stack.
Pre none.
Post if the stack is not empty, DataOut receives data on its top. The stack remains unchanged.
Return success or underflow.

// For Linked Stack
1. If (count > 0)
   1. DataOut = top->data
   2. Return success
2. Else
   1. Return underflow
3. End Top
isEmpty Linked Stack

<boolean> isEmpty()
Determines if the stack is empty.

Pre none
Post return stack status
Return TRUE if the stack is empty, FALSE otherwise

1. if (count = 0)
   1. Return TRUE
2. else
   1. Return FALSE
end isEmpty
isFull Linked Stack

<boolean> isFull()
Determines if the stack is full.

Pre    none
Post   return stack status
Return TRUE if the stack is full, FALSE otherwise

// For Linked Stack
1. Allocate pNew // pNew is NULL if unsuccessful.
2. if (pNew is not NULL)
   1. recycle pNew
   2. return TRUE
3. else
   1. return FALSE
end isFull
Contiguous Implementation of Stack

(Automatically Allocated Array)

**Conceptual**

**Stack**

- **top** <integer>
- **count** <integer>
- **data** <array of <DataType>>

End Stack

**Physical**

Stack with pre-defined maxsize and has n elements.

[Diagram of stack with push and pop operations]

CuuDuongThanCong.com

https://fb.com/tailieudientucntt
Contiguous Implementation of Stack (cont.)

Stack is empty

Push the 1\textsuperscript{st} element

Stack having n elements
Create Stack

```c
<void> Create()
// Specifications here are similar to specifications for Linked Stack
1. count = 0
2. top = -1
end Create
```
<errorCode> Push(val DataIn <DataType>)

// Specifications here are similar to specifications for Linked Stack

1. if (count = maxsize)
   1. return overflow

2. else
   1. top = top + 1
   2. data[top] = DataIn
   3. count = count + 1
   4. return success

end Push
Pop Stack

<ErrorCode> Pop()

// Specifications here are similar to specifications for Linked Stack

1. if (stack is empty)
   1. return underflow

2. else
   1. top = top – 1
   2. count = count - 1
   3. return success

end Pop
Top Stack

<ErrorCode> Top(ref DataOut <DataType>)

// Specifications here are similar to specifications for Linked Stack

1. if (count = 0)
   1. return underflow

2. else
   1. DataOut = data[top]
   2. return success

end Top
Stack status

**isEmpty()**

1. if (count = 0)
   1. return TRUE

2. Else
   1. return FALSE

end isEmpty

**isFull()**

1. if (count = maxsize)
   1. return TRUE

2. Else
   1. return FALSE

end isFull

**size()**

1. return count

end size