Stack

definitions are highlighted!

Data structures

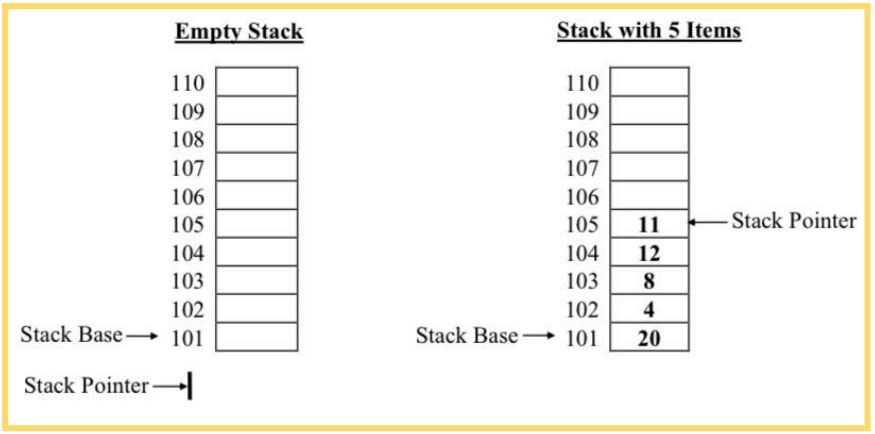
- A group of related data items organised in the computer
- Allows a large number of pieces of data to be managed as a single set
- e.g. array, stack, queue, linked list, binary search tree, hash table
- Static and dynamic
    - Static — do not change in size while the program is running; once the size (upper bound) is declared, it cannot be changed
        - e.g. array
    - Dynamic — can increase and decrease in size while a program is running
        - e.g. linked list

Type	Advantages	Disadvantages
Static	• Space allocated during compilation • Easier to program • Easier to check for overflow • Random access	• Space allocation is fixed • Wastes space when partially used
Dynamic	• Only uses space required • Efficient memory usage • Emptied storage can be returned to the system	• Difficult to program • Searches can be slow • Serial access

Stack

- *A collection of data items which can only be accessed at one end (at the top of the stack)*

- Frequently used for temporary storage of data
- Operations:
    - Push — adding a new item at the top of the stack
    - Pop — removing the item at the top of the stack
- Last In, First Out (LIFO) data structure
    - The first value that is popped from the stack is the value that has remained on the stack for the shortest amount of time.
    - The very last item popped from the stack (before it becomes empty) will have remained on this stack for the longest time.
- Implementation:
    - A block of memory large enough to accommodate the stack at its maximum size is reserved.
        - The size of this block is a critical design decision — if it is too small, the stack will exceed the allotted storage space; but if too much room is reserved, memory space will be wasted.
    - Indicated by 2 pointers in computer memory:
        - Stack base — indicates the base of the stack
            - Has a fixed value as long as the stack exists
        - Stack pointer — indicates the position of the top of the stack
            - Value varies as data is pushed or popped from the stack
            - When this is null, the stack is empty.

- Logical view:
  - 

- If an attempt is made to pop an item from an empty stack, underflow will occur.
- If an attempt is made to push another item onto a full stack, overflow will occur.
  - A stack always has a maximum size because memory cannot grow indefinitely.
  - We can check the stack pointer to determine whether the stack is full, if the stack is implemented as an array.

Algorithms

- Setting up a stack
  - **#1:** Reserve … spaces for the stack.
  - **#2:** Let stackPointer = null.
  - **#3:** Let stackBase = smallest location allocated.
  - **#4:** Exit.
- Pushing an item onto a stack
  - **#1:** If stackPointer = largest location allocated for the stack, print “Stack full!” and exit.
  - **#2:** If stackPointer = null, set stackPointer = stackBase and store the given item in the location pointed to by stackPointer.
  - **#3:** Else, increment stackPointer by 1 and store the given item in the location pointed to by stackPointer.

- **#4:** Exit.
- Pseudocode:

```
PROCEDURE PUSH
    IF StackPointer = Limit THEN
        OUTPUT "STACK FULL" and EXIT
    ENDIF
    IF StackPointer = Null THEN
        StackPointer ← StackBase
        Stack[StackPointer] ← NewItem
    ELSE
        StackPointer ← StackPointer + 1
        Stack[StackPointer] ← NewItem
    ENDIF
ENDPROCEDURE
```

- Popping an item from a stack
  - **#1:** If stackPointer = null, print “Empty stack!” and exit.
  - **#2:** Read the content of the location pointed to by stackPointer.
  - **#3:** If stackBase is not equal to stackPointer, decrease stackPointer by 1.
  - **#4:** Else, set stackPointer to null.
  - **#5:** Exit.
  - Pseudocode:

```
FUNCTION POP
    IF StackPointer = Null THEN
        OUTPUT "EMPTY STACK" and EXIT
    ELSE
        PoppedItem ← Stack[StackPointer]
    ENDIF
    IF StackBase <> StackPointer THEN
        StackPointer ← StackPointer - 1
    ELSE
        StackPointer ← Null
    ENDIF
    RETURN PoppedItem
ENDFUNCTION
```

- Peeking at the top item
    - Pseudocode:

```
FUNCTION peek
    IF isEmpty THEN
        print("Stack is empty")
    ELSE
        item ← stack[top]
        RETURN item
    ENDIF
ENDFUNCTION
```

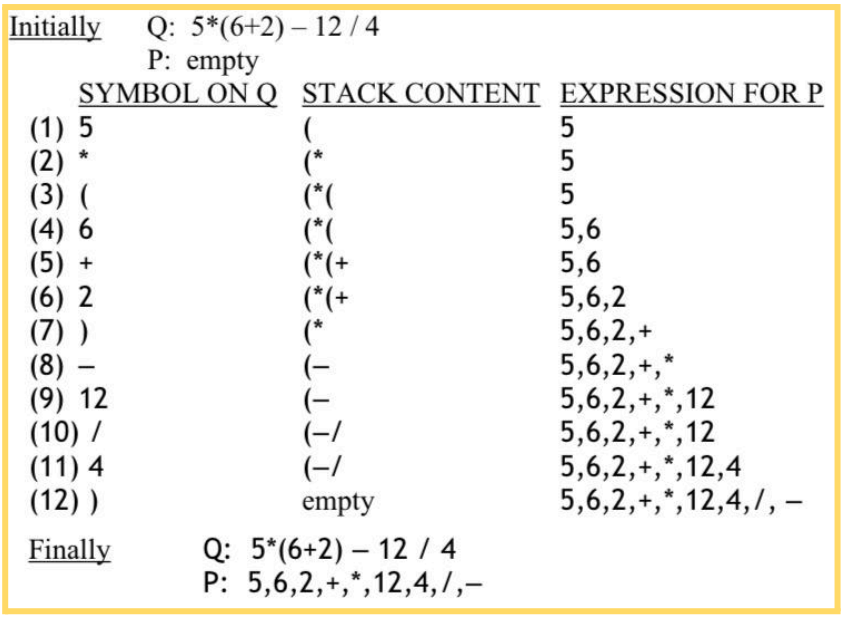
- Checking whether the stack is empty
- Checking whether the stack if full

Implementations

- Array implementation (static)
    - stackBase is at the first index (0 or 1) as it always points to the first location. So, it does not need to be stored in a variable.
    - stackPointer indicates the top of the stack.
    - maxSize holds the maximum size of the stack.
- Linked list implementation (dynamic)
    - The 'Start' pointer of the linked list is the stackPointer of the stack.
        - New nodes are inserted at the start of the list, because it's a LIFO data structure.
    - Overflow is unlikely to happen as each new node is dynamically allocated; the stack can shrink or grow as much as needed.
- Python list implementation (dynamic)
    - The top of the stack is the last element in the list.
    - The 'append' function is used to push items onto the stack.
    - The 'pop' function is used to pop items from the stack.
    - [-1] is used to peek at the top item of the stack.

Usage

- Call stack in recursion
    - As each function is called, the (#1) return address, (#2) input parameter(s) and (#3) local variables are saved in a stack frame and pushed onto the call stack.
    - As each function call is completed, the stack frame is popped from the call stack and the program continues the next instruction as indicated by the return address.
- Evaluating arithmetic expressions
    - Converting Infix notation to Postfix notation
        - **#1:** Push '(' onto the theStack and append ')' to the end of the input string.
        - **#2:** While theStack is not empty, read the input string from left to right and repeat the following:
            - If the current character is an operand (i.e. number), copy it to the output string.
            - If the current character is '(', push it onto theStack.
            - If the current character is an operator:
                - Pop operators at the top of theStack and insert them into the output string while they have equal or higher precedence than the current operator.
                - Push the current operator to theStack.
            - If the current character is ')':
                - Pop operators at the top of the stack and insert them into the output string until '(' is at the top of theStack.
                - Pop the '(' from theStack.
        - **#3:** Exit.



Initially Q: 5*(6+2) – 12 / 4
P: empty

	SYMBOL ON Q	STACK CONTENT	EXPRESSION FOR P
(1)	5	(	5
(2)	*	(*	5
(3)	(	(*(	5
(4)	6	(*(	5,6
(5)	+	(*(+	5,6
(6)	2	(*(+	5,6,2
(7)	)	(*	5,6,2,+
(8)	–	(–	5,6,2,+,*
(9)	12	(–	5,6,2,+,*,12
(10)	/	(–/	5,6,2,+,*,12
(11)	4	(–/	5,6,2,+,*,12,4
(12)	)	empty	5,6,2,+,*,12,4,/, –

Finally Q: 5*(6+2) – 12 / 4
P: 5,6,2,+,*,12,4,/,–

- Evaluating Postfix notation
  - **#1:** Append the null character at the end of the input string.
  - **#2:** While null is not encountered, read P from left to right and repeat the following:
    - If the current character is an operand, push it onto theStack.
    - If the current character is an operator @:
      - Pop the top 2 elements of theStack into variables x and y.
      - Calculate y @ x.
      - Push the result to theStack.
  - **#3:** When null is encountered, pop the top value of theStack (this is the result of the input string).
  - **#4:** Exit.

■

	SYMBOL ON P	STACK CONTENT
(1)	5	5
(2)	6	5,6
(3)	2	5,6,2
(4)	+	5,8
(5)	*	40
(6)	12	40,12
(7)	4	40,12,4
(8)	/	40,3
(9)	–	37

@star.notes