

Chapter 3

Abstract Data Types (ADT)

Abstract Data Types

- ADT: set of **objects** together with a set of **operations**
- Mathematical **abstraction** ; no mention in ADT of *how* the set of operations is implemented
- Examples: Objects such as *lists*, *sets*, and *graphs*, along with their operations, can be viewed as ADTs
- The C++ *class* allows for the implementation of ADTs, with appropriate hiding of implementation details
- Any implementation changes should not affect the use of the ADT (completely transparent to the rest of the programme)

Usual Data Types

Integer, Double, Boolean

Implementation details of these are not known

Some operations on these can be defined: *add*,
multiply, *read*, *write*

The programmer just uses these operations
without knowing their implementation.

The List ADT

- We will deal with a general list of the form $A_0, A_1, A_2, \dots, A_{N-1}$

Popular operations on Lists:

- *printList* and *makeEmpty*,
- *find* (returns the position of the first occurrence of an item)
- *Access/findkth* (*read/change*) an element at the beginning/end/ith position.
- *Add* an element at the beginning/end/ith position
- *Delete* an element at the beginning/end/ith position

Array Implementation of a list

- A List can be implemented as an array
- Need to know the maximum number of elements in the list at the start of the program (The C++ *vector* class is different)
- Adding/Deleting an element at/from a specific position can take $O(n)$ operations if the list has n elements.
- Accessing/changing an element anywhere takes $O(1)$ operations independent of n

Adding an element at the beginning

Suppose first position ($A[0]$) stores the current size of the list

Actual number of elements $\text{currsize} + 1$

Adding at the beginning:

Move all elements one position behind

Add at position 1; Increment the current size by 1

For ($j = A[0]+1$; $j > 0$; $j--$) **Complexity: $O(n)$**

$A[j] = A[j-1];$

$A[1] = \text{new element};$

$A[0] \rightarrow A[0]+1;$

Of course, need to check if there is space for the new element? (Same for the add operations into arrays in next slides)

Adding an element at the End

Add the element at the end

Increment current size by 1;

$A[A[0]+1] = \text{new element};$

$A[0] \rightarrow A[0]+1;$

Complexity: $O(1)$

Adding at k^{th} position

Move all elements one position behind, from k^{th} position onwards;

Add the element at the k^{th} position

Increment current size by 1;

For ($j = A[0]+1$; $j > k$; $j--$)

$A[j] = A[j-1];$

$A[k] = \text{new element};$

$A[0] \rightarrow A[0]+1;$

Complexity: $O(n)$

Deleting an Element

Deleting at the beginning:

Move all elements one position ahead;

Decrement the current size by 1

For ($j = 1$; $j < A[0]$; $j++$)

$A[j] = A[j+1];$

$A[0] \rightarrow A[0]-1;$

Complexity: $O(n)$

Deleting at the End

Delete the element at the end



Decrement current size by 1;

$A[0] \rightarrow A[0]-1;$

Complexity: $O(1)$

Deleting at the k^{th} position

Move all elements one position ahead, from $(k+1)^{\text{th}}$ position onwards;

Decrement the current size by 1;

For ($j = k; j < A[0]+1; j++$)

$A[j] = A[j+1];$

$A[0] \rightarrow A[0]-1;$

Complexity: $O(n)$

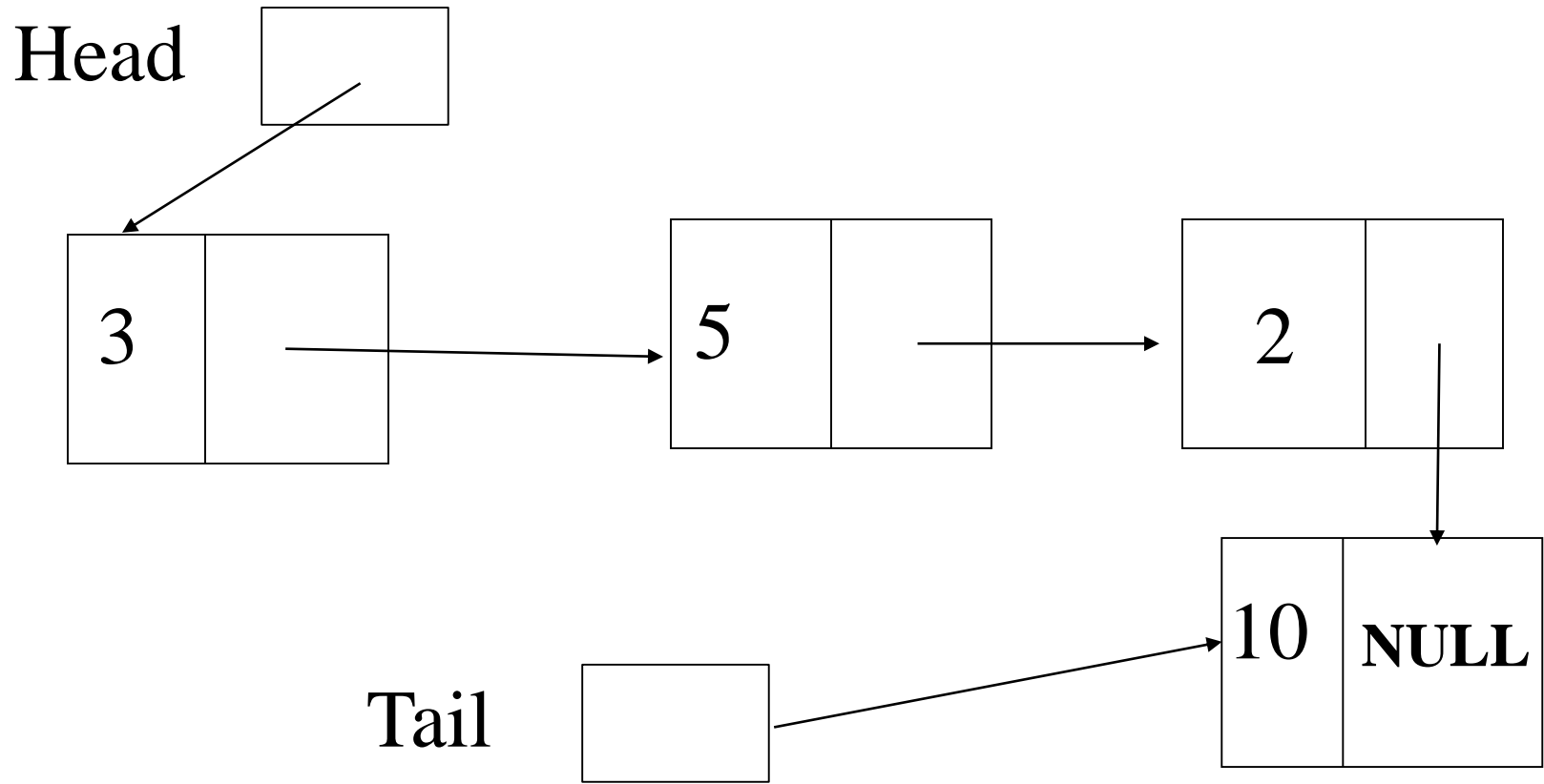
Accessing an Element at the k^{th} position

$A[k];$

$O(1)$ operation;

Linked List

- To avoid the linear cost of insertion and deletion, we need to ensure the list not contiguously stored
- Otherwise, entire parts of the list will need to be moved.
- A Linked List consists of a sequence of nodes
- A node in the linked list consists of two elements
 - Value of the corresponding element in the list
 - Pointer to the next element `nextptr`.
- In last element, the pointer is a null pointer `nullptr`
- A pointer to the first node referred to as *head*, and, possibly, a pointer to the last node, the *tail*.



```
struct ListNode {  
    int value;  
    ListNode *next;  
};
```

Adding an element at the beginning

- Create a new node;
- Assign the value of the new element to the *value* part in this node;
- Assign the *head* ptr content to this new node pointer (i.e. make it point to the first list node);
- Make the *head* pointer point to the new node;

```
ListNode* newptr = new ListNode
```

```
newptr → value = val;
```

```
newptr → next = head;
```

```
head = newptr;
```

O(1)

Adding an element at the end

- Create a new node;
- Assign the value of the new element to the *value* part in this node;
- Assign to the ptr of the last node the address of the new node (i.e. content of *newptr*);
- Make the *tail* pointer point to the new node;

```
ListNode* newptr = new ListNode;
```

```
newptr → value = val;
```

```
Tail → nextPtr = newPtr;
```

```
tail = newPtr;
```

O(1)

Accessing a specific element

```
ListNode* current = head;  
while (current →value != val && current →next!= 0)  
    current = current →next;  
if (current →value == val )  
    return true;  
else  
    cout << "element not found";  
    return;
```

Complexity: $O(n)$

Adding at the k^{th} position

```
ListNode* newptr = new ListNode;  
newptr → value = val;  
newptr → next = NULL;  
ListNode* current = head;  
for (count = 1; count < k-1 && current != 0; count++)  
    current = current → next;  
if (count == k-1 ) {  
    newptr → next = current → next;  
    current → next = newptr;  
    return true;  
}  
else  
    cout << "position out of range" ;  
    return;
```

Complexity: $O(n)$

Deleting a node at the k^{th} position

```
ListNode* current = head;
ListNode* prevPtr;
for (count = 1; count < k-1 && current != 0; count++) {
    prevPtr = current;
    current = current → next; }
if (count == k-1 ) {
    prevPtr →next = current → next;
    delete current;
    return true;
}
else
    return false;
```

Complexity: $O(n)$

Delete a node at the k^{th} position

Complexity depends on access complexity

$O(1)$ for deleting first element;

$O(1)$ or $O(n)$ for deleting the last element;
(depending on whether tail pointer exists or not)

$O(k)$ for any other element;

Advantage of Linked Lists

No need to know the maximum number of elements ahead of time.

Disadvantages of Singly LLs

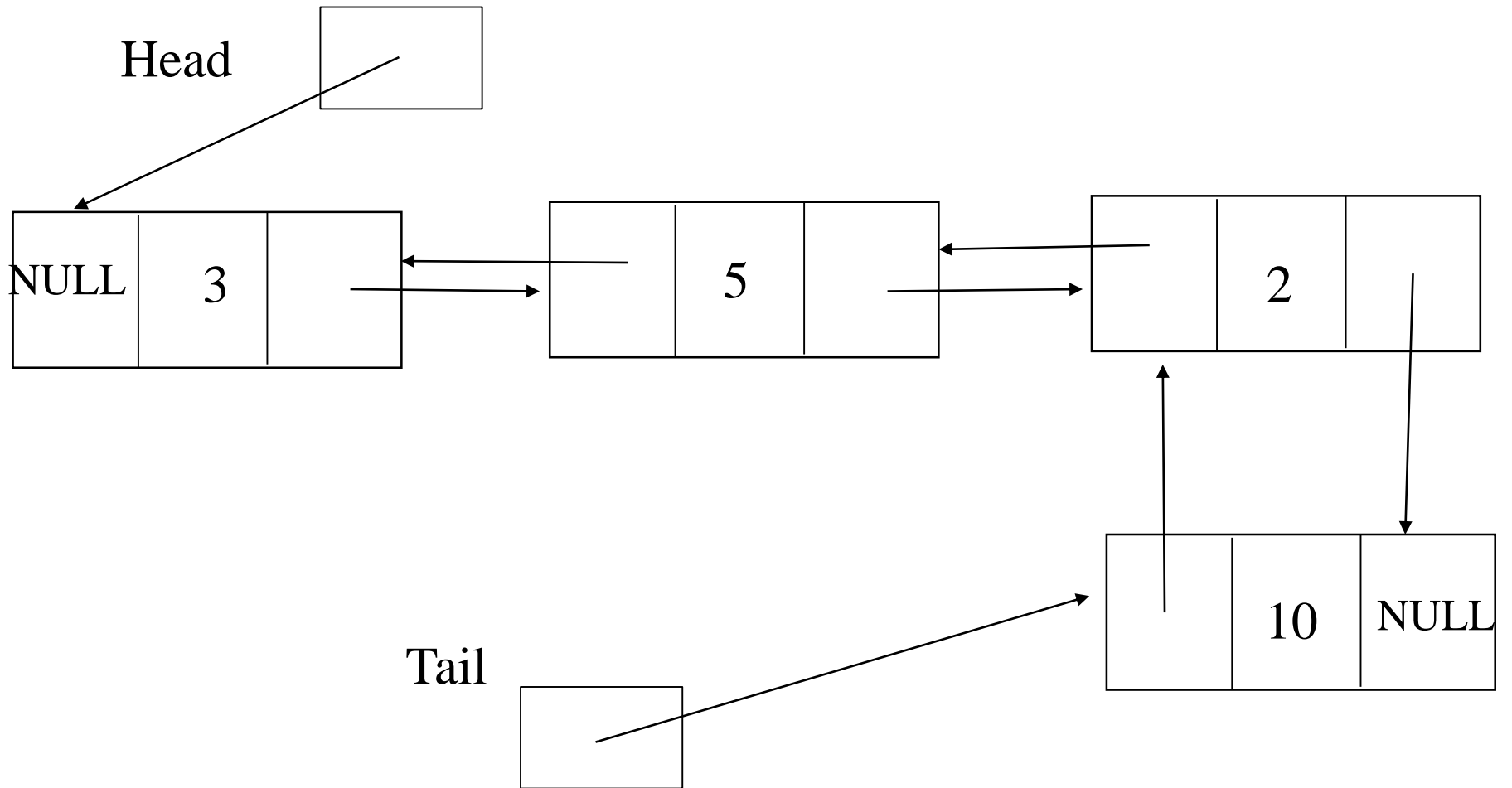
- It uses one extra memory when compared to the array.
- Since the elements in memory are stored randomly, the elements are accessed sequentially; no direct access is allowed.

Doubly Linked Lists

A node contains pointers to previous and next element

One can move in both directions

Complexities?



Advantages Of DLLs

- Reversing the doubly linked list is very easy.
- It can allocate or reallocate memory easily during its execution.
- As with a singly linked list, it is the easiest data structure to implement (not considering arrays).
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node to be deleted but in the doubly linked list, it only requires the pointer which is to be deleted.

Disadvantages of DLLs

- It uses extra memory when compared to the array and singly linked list.
- Since the elements in memory are stored randomly, the elements are accessed sequentially; no direct access is allowed.

Uses of DLLs

- Used by the browser to implement backward and forward navigation of visited web pages (back and forward buttons)
- Used by various applications to implement undo and redo functionality.
- Also in many operating systems, the **thread scheduler** (the code that chooses what process needs to run at which time) maintains a doubly-linked list of all processes running at that time.
- Other data structures like stacks, Hash Tables, Binary trees can also be constructed or programmed using a doubly-linked list.
- Used in the navigation systems where front and back navigation are required. (See also Search in AI)

Circular Lists

- The last node points to the first one

Potentially needed operations:

- Insertion / deletion: front, end, some specific point



Circular Lists

Advantages:

- We can go to any node and traverse from any node. We just need to stop when we visit the same node again.
- As the last node points to the first node, going to the first node from the last node just takes a single step

Applications of Circular Linked Lists

- Used in multiplayer games to swap between players.
- On Operating Systems: Multiple running applications can be placed in a CLL so the OS keeps on iterating over these applications giving them time slices. (Round Robin Scheduling)
- Music or media player: the use of CLLs allows for a continuous playback of the playlist

Stacks

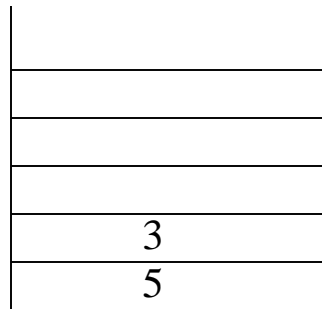
- A Stack is a list where you can access add or delete elements at one end only.
- Stacks are called “**last in first out**” (LIFO): the last added element among the current ones can be accessed or deleted.
- All of these operations take constant time.
 - Deletion is called “*pop*”
 - Addition is called “*push*”
- You can also test whether the stack is empty but nothing else.

EXAMPLE

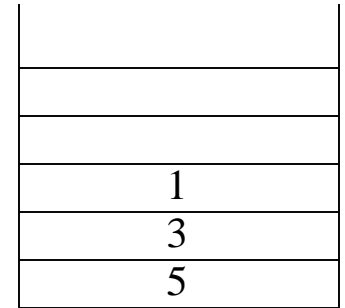
Push 5, 3, 1, Pop, Pop, Push 7



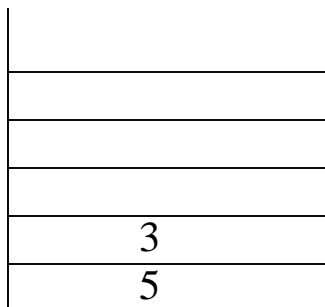
Push 5



Push 3



Push 1



Get 1

Pop



Get 3

Pop



Push 7

Function Calls

- When a subroutine calls another subroutine (including itself) the system must first store its register contents.
- So, the system pushes the register contents onto a stack.
- The new function uses the registers.
- When the program returns to the old function, the system pops the stack.
- Nested Functions and Stack Overflow!

Stack Implementation using an Array

- Need to know the maximum number of elements
- Delete, Access, and Add at the end are all $O(1)$
- Store array length in the 0^{th} position (to test if stack is empty)
- Delete/Access/Add at $A[\text{array length}+1]$

Example

Push 5, 3, 1, Pop, Pop, Push 7

| | | | | | | | | |
|---|---|--|--|--|--|--|--|--|
| 1 | 5 | | | | | | | |
|---|---|--|--|--|--|--|--|--|

| | | | | | | | | |
|---|---|---|--|--|--|--|--|--|
| 2 | 5 | 3 | | | | | | |
|---|---|---|--|--|--|--|--|--|

| | | | | | | | | |
|---|---|---|---|--|--|--|--|--|
| 3 | 5 | 3 | 1 | | | | | |
|---|---|---|---|--|--|--|--|--|

| | | | | | | | | |
|---|---|---|--|--|--|--|--|--|
| 2 | 5 | 3 | | | | | | |
|---|---|---|--|--|--|--|--|--|

| | | | | | | | | |
|---|---|--|--|--|--|--|--|--|
| 1 | 5 | | | | | | | |
|---|---|--|--|--|--|--|--|--|

| | | | | | | | | |
|---|---|---|--|--|--|--|--|--|
| 2 | 5 | 7 | | | | | | |
|---|---|---|--|--|--|--|--|--|

Stack Implementation using a Singly Linked List

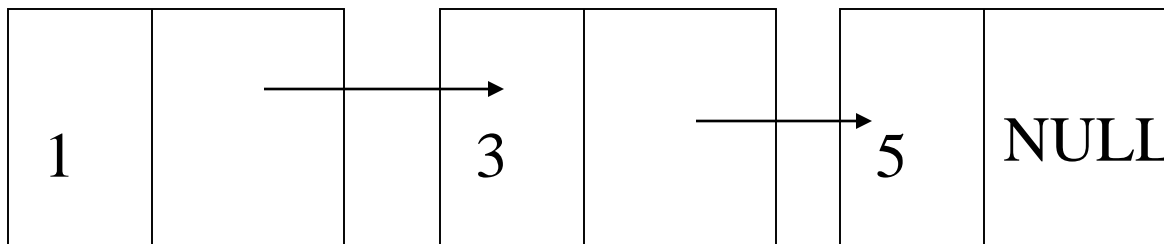
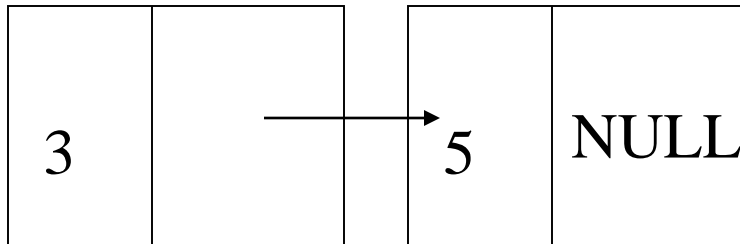
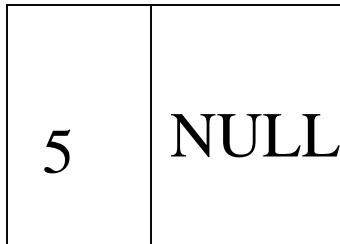
Need not know the maximum size

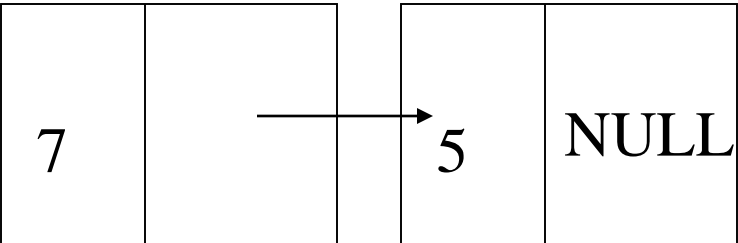
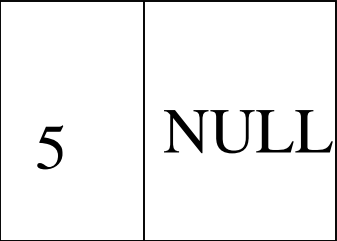
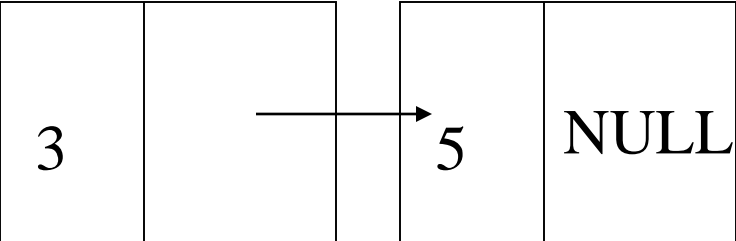
Need not have a special header.

Add/Access/Delete in the beginning, $O(1)$

One would not use a stack if *Add / Access / Delete* operations are needed at some position k different from the beginning (top).

Push 5, 3, 1, Pop, Pop, Push 7





Uses of Stacks: Symbol Matching

Braces, parentheses, brackets, begin-ends must match each other

[{ [()] }]

[{] } ()

Easy check using stacks

Start from beginning of the file.

Push the beginning symbols you wish to match, ignore the rest.

Push brace, parentheses, brackets, ignore the alphabets

Whenever you encounter a right symbol, pop an element from the stack. If stack is empty, then error.

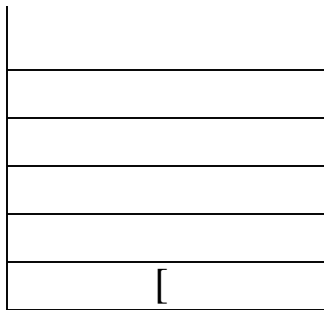
Otherwise, if the popped element is the corresponding left symbol, then fine; else there is an error.

What is the complexity of the operation? $O(n)$

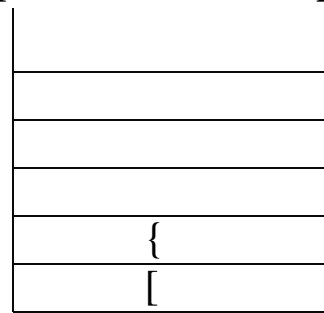
EXAMPLE

Check braces, brackets, parentheses matching
[a+b{ 1*2]9*1 }+(2-1)

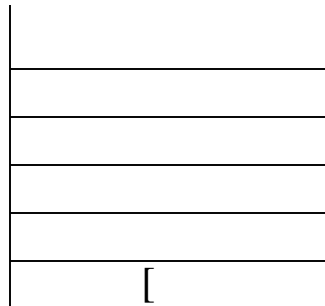
Push [, Push {, Pop, Pop, Push (, Pop



Push [



Push {



Get {

Pop **Expecting [**

Oops! Something wrong, was expecting [

EXAMPLE

Check brace, bracket parentheses matching
[a+b{ 1*2}9*1]+(2-1)

Push [, Push {, Pop , Pop, Push (, Pop



Push [



Push {

Get {



Pop

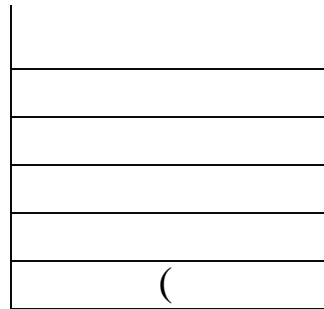
Expect {

Perfect!



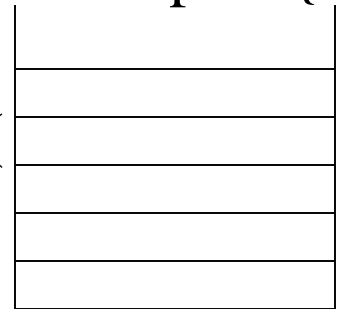
Get [

Pop Expect [



Push (

Get (



Pop Expect (

Evaluation of arithmetic expressions

$$A * B + C$$

$$A * (B + C)$$

$$A + B + C * D$$

$$A * D + B + C * E$$

To do the correct operation, the calculator needs to know the priorities of the operators

We don't need any priority nor parentheses if the operation is expressed in postfix or reverse polish notation.

Postfix Notation

Infix notation

Postfix notation

$A * B + C$

$AB * C +$

$A * (B + C)$

$ABC + *$

$A + B + C * D$

$AB + CD * +$

$A * D + B + C * E$

$AD * B + CE * +$

Suppose the expression is in postfix, how do we compute the value? (N.B.: Postfix notation already takes into account operators priorities.)

Computation of a Postfix Expression

Whenever you see a number push it in the stack.

Whenever you see an operator,

Pop 2 operands,

Apply the operator to the 2 operands

Push the result

At the end Pop to get the answer

Complexity?

$O(n)$

EXAMPLE

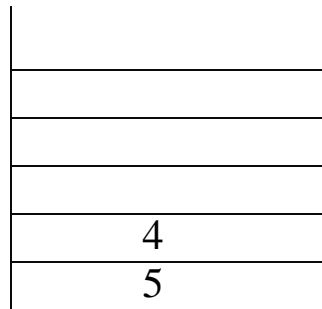
Compute $5*4+6+7*2$

Result: 40

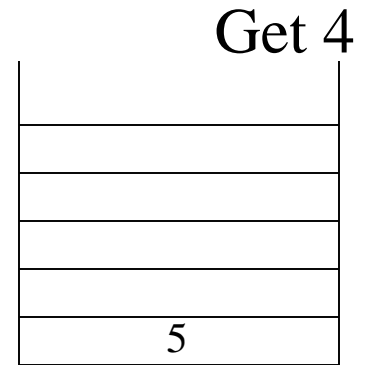
Postfix: 5 4*6+7 2*+



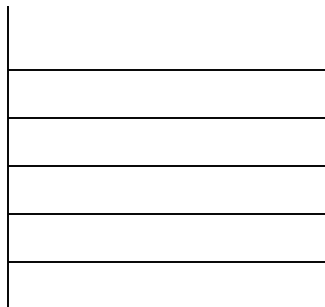
Push 5



Push 4

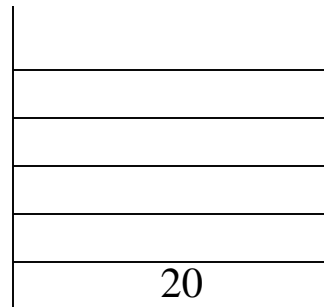


Pop

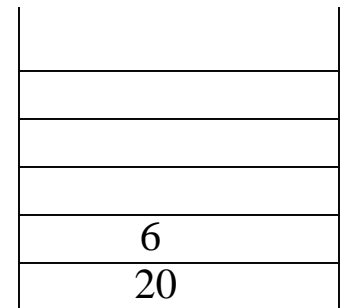


Pop

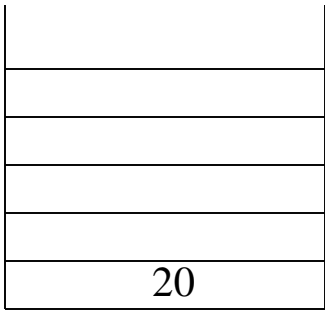
Get 5



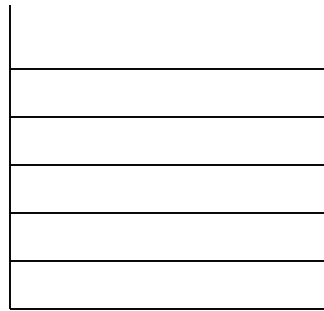
Push 20



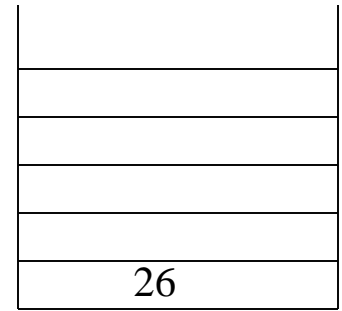
Push 6



Get 6



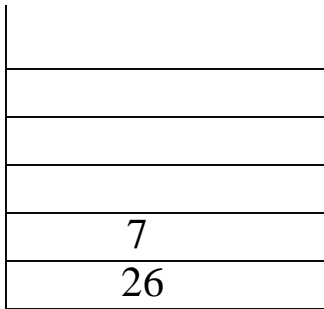
Get 20



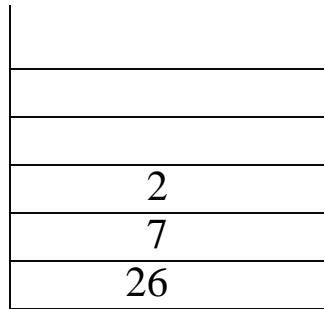
Pop

Pop

Push 26

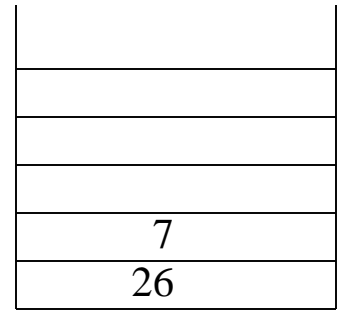


Push 7



Get 2

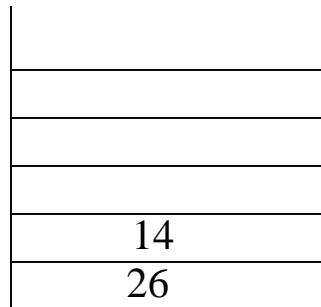
Push 2



Pop

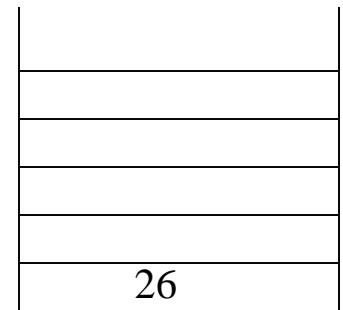


Get 7



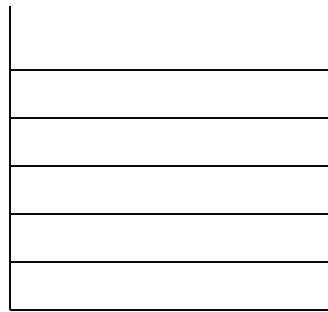
Get 14

Push 14



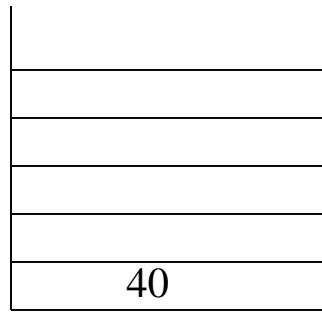
Pop

Pop



Get 26

Pop



Push 40

Pop to get the value of the arithmetic expression.

Prefix (or Polish) Notation

Infix notation

Prefix notation

$A * B + C$

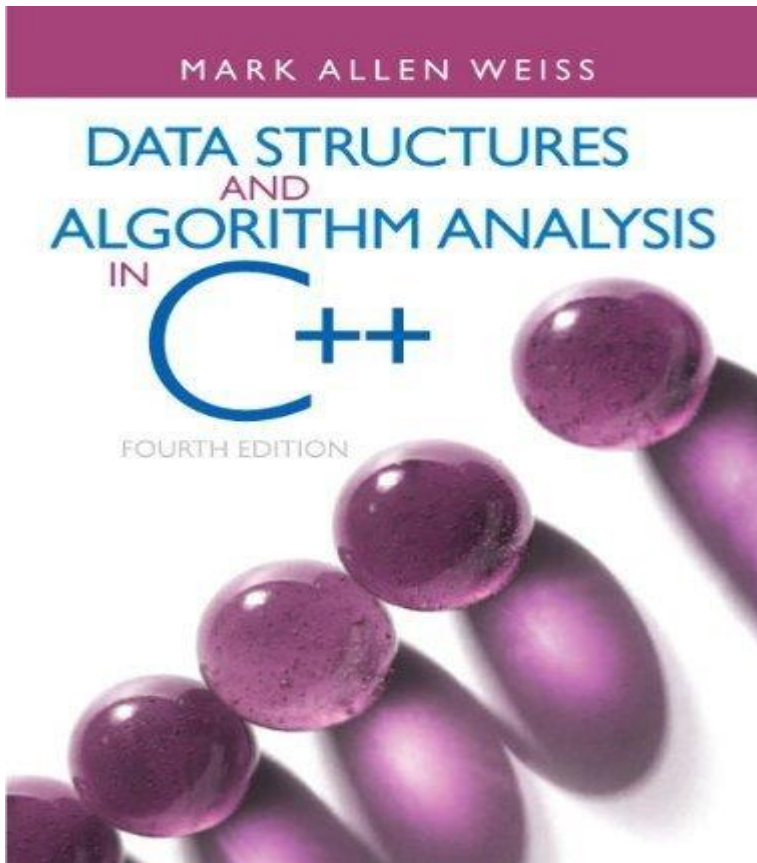
$*AB + C$

$A + B) * C - (D - E) * (F + G)$

$- * + ABC * - DE + FG$

Exercise: Think which data structure you could use to implement the evaluation of an expression written in polish (prefix) notation. And how would you do it?

Slides based on the textbook



Mark Allen Weiss,
(2014) Data
Structures and
Algorithm Analysis
in C++, 4th edition,
Pearson.

Acknowledgement: This **course PowerPoint**s make substantial (non-exclusive) use of the PPT chapters prepared by Prof. Saswati Sarkar from the University of Pennsylvania, USA, themselves developed on the basis of the course textbook. Other references, if any, will be mentioned wherever applicable.