Our First Data Structure

- The story so far
  - We understand the notion of an abstract data type.
  - Saw some fundamental operations as well as advanced operations on arrays.
- This week we will
  - use the array ADT but in a slightly different manner(s).
  - and advanced applications of the data structure.

Motivation

- Think of developing a modern editor.
  - supports undo/redo among other things.
  - Suppose that currently words w1, w2, w3 are inserted in that order.
  - When we undo, which word has to be undone.
    - w3
  - Next undo should remove w2.
  - So, the order of undo's should be the reverse order of insertion.

Another Example

- Imagine books piled on top of each other.
- To access a book in the pile, one may need to remove all the books on top of the book we need.

- Similarly, in some cafeterias, plates are piled.
  - The plate we take is the one that is placed the last on the pile.
  - see our dining hall plates.

Motivation

- All these examples suggest that there is a particular order in accessing data items.
  - Last In First Out (LIFO)
- Turns out that this order has several other applications too.
- This week, we will formalize this order and study its important applications.

The Stack ADT

- We can say that the above examples are connected by:
  - a stack of words to be deleted/inserted
  - a stack of books to be removed/repiled
  - a stack of plates
- The common theme is the stack
- This stack can be formalized as an ADT.

The Stack ADT

- We have the following common(fundamental) operations.
- create() -- creates an empty stack
- push(item) – push an item onto the stack.
- pop() -- remove one item from the stack, from the top
- size() -- return the size of the stack.

The Stack ADT

- One can implement a stack in several ways.
- We will start with using an array.
  - Only limitation is that we need to specify the maximum size to which the stack can grow.
  - Let us assume for now that this is not a problem.
  - the parameter n refers to the maximum size of the stack.

Stack Implementation

function create(S)

//depends on the language..

//so left unspecified for now

end-function.

function push(item)

begin

S[top] = item;

top = top + 1;

end

Stack Implementation

```
function pop()
begin
    return S[top--];
end
```

```
function size()
begin
    return top;
end
```

One Small Problem

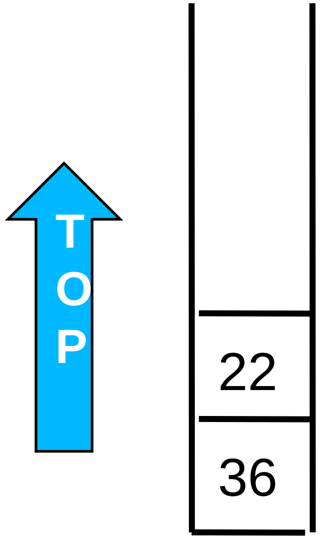
- Suppose you create a stack of 10 elements.
- The stack already has 10 elements
- You issue another push() operation.
- What should happen?
  - Need some error-handling.
  - Modified code looks as follows.

Push, Pop with Error Handling

```
function push(item)
begin
    if top == n then
        return "ERROR: STACK FULL"
    else
        S[top++] = item
end.
```

```
function pop()
begin
    if (top == 0) then
        return "ERROR: STACK EMPTY"
    else
        return S[top--]
end
```

Typical Conventions

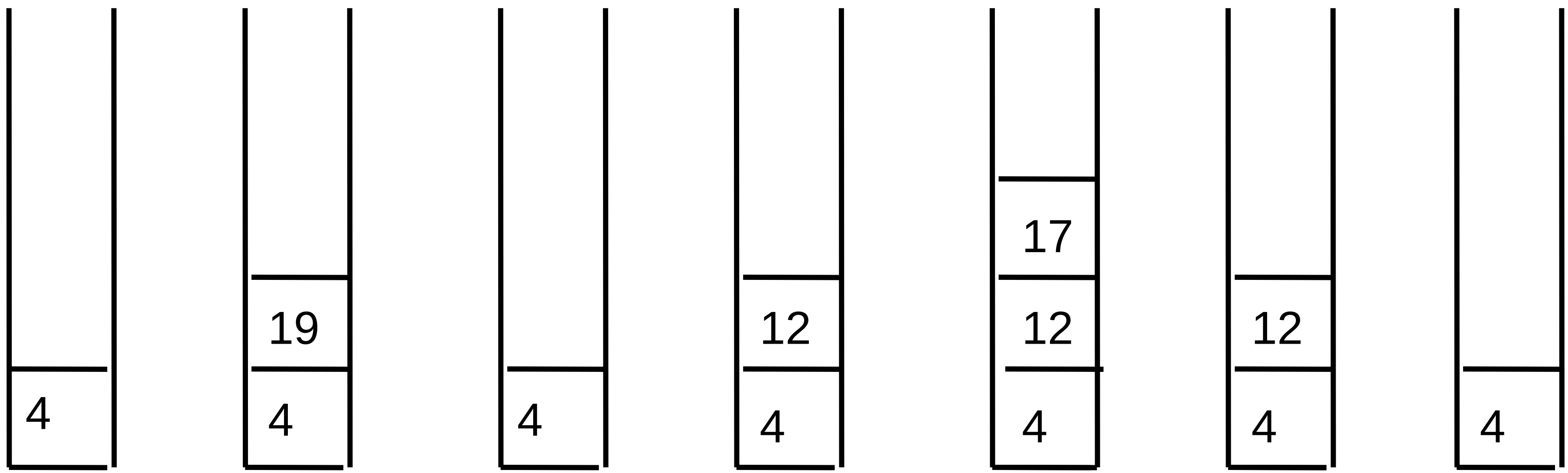


- When drawing stacks, a few standard conventions are as follows:
  - A stack is drawn as a box with one side open.
  - The stack is filled bottom up.
  - So, top of the stack is towards the North.

Example

- Consider an empty stack
- Consider the following sequence of operations
  - Push(4), Push(19), pop(), push(12), push(17), pop(), pop().
- Show the resulting stack.

Solution



- Consider an empty stack
- Consider the following sequence of operations
  - Push(4), Push(19), pop(), push(12), push(17), pop(), pop().
- Show the resulting stack.

Applications of Stacks

- We will consider one applications of our new data structure to
  - Expression Evaluation

Expression Evaluation

- Imagine pocket calculators or the Google calculator.
- One can type in an arithmetic expression and get the results.
  - Example: 4+2 = 6.
  - Another example: 6+3*2 = 12. And not 18. Why?
- For simplicity, let us consider only binary operators.

Expression Evaluation

- How do we evaluate expressions?
- There is a priority among operators.
  - Multiplication has higher priority than addition.
- When we automate expression evaluation, need to consider priority.
- To disambiguate, one also uses parantheses.
  - The previous example written as 6 + (3*2).

How to evaluate an expression?

- We evaluate expressions from left to right.
- All the while worrying about operator precedence.
- What is the difficulty?
- Consider a long expression.
  - 2 + 3 * 8 * 2 * 2 + 1
- When we look at the first 2, we can hopefully remember that 2 is one of the operands.
- The next thing we see is the operator +. But what is the second operand for this operator.

How to Evaluate an Expression

- This second operand may not be seen till the very end.
- Would it be helpful if we could associate the operands easily.
- But the way we write the expression, this is not easy.

How to Evaluate an Expression

- There are other ways of writing expressions.
- The way we write expression is called the infix style.
  - The operator is placed between the operands.
- There is (at least one) another way to write expressions called the prefix style.
- In a prefix expression, operators are written before the operand.
  - The operands immediately succeed the operator.

Prefix Expression

- Turns out if we maintain the previous condition, then there is no ambiguity in evaluating expressions.
- This is also called as Polish Notation.
- For instance, the expression 3+2 is written as + 3 2
- How about the expression 2 + 3 * 6?
  - Notice that the operands for * are 3 and 6.
  - The operands for + are 2 and the result of the expression 3 * 6.
- So how to write the prefix equivalent for 2 + 3 * 6

Prefix Expression

- Given the above observations, we can write it as + 2 * 3 6.
- Another example: 3 + 4 + 2 * 6. The prefix is + 3 + 4 * 2 6.
- But can we write prefix expressions? We are used to writing infix expressions.
- Our next steps are as follows
  - Given an infix expression, convert it into a prefix expressions.
  - Evaluate a prefix expression.

Our Next Steps

- We have two problems. Of these let us consider the second problem first.
- The problem is to evaluate a given prefix expression.
- Our solution closely resembles how we do a manual calculation.

Evaluating a Prefix Expression

- Some observation(s)
  - The operator precedes the operands.
  - Let us evaluate from right to left.
- This helps us in devising an algorithm.
- Imagine that the prefix expression is stored in an array.
  - one operator/operand at an index.

Evaluating a Prefix Expression

- Can we use a stack?
- How can it be used?
- What should we store in the stack?

Evaluating a Prefix Expression

- We have to keep track of operands so that when we see an operator, we should be able to apply the operator immediately.
- Suppose we maintain (translate) the invariant that the operands are on the top (and next to top) of the stack.
- Once we evaluate an operator, where should we now store the result?
  - The result could be the operand of a future operator.
  - So, pile it on the stack.

Evaluating a Prefix Expression

- The above suggests the following approach.
- Start from the right side.
- For every operand, push it onto the stack.
- For every operator, evaluate the operator by taking the top two elements of the stack.
  - place the result on top of the stack.
- The pseudo-code is given next.

Algorithm to Evaluate a Prefix Expression

```
Algorithm EvaluatePrefix(E)
begin
    Stack S;
    for i = n to 1 do
    begin
    if E[i] is an operator, say o then
        operand1 = S.pop();
        operand2 = S.pop();
        value = operand1 o operand2;
        S.push(value);
    else
        S.push(E[i]);
    end-for
end-algorithm
```

Algorithm to Evaluate a Prefix Expression

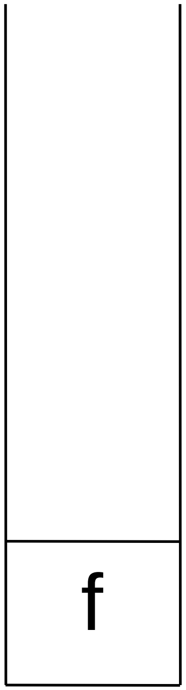
- In the above, n refers to the number of operators + the number of operands.
- The time taken for the above algorithm is linear in n.
  - There is only one for loop which looks at each element, either operand or operator, once.
- We will see an example next.

Example to Evaluate a Prefix Expression

- Consider the expression + * + a b + c d + e f.
- Show the contents of the stack and the output at every step.

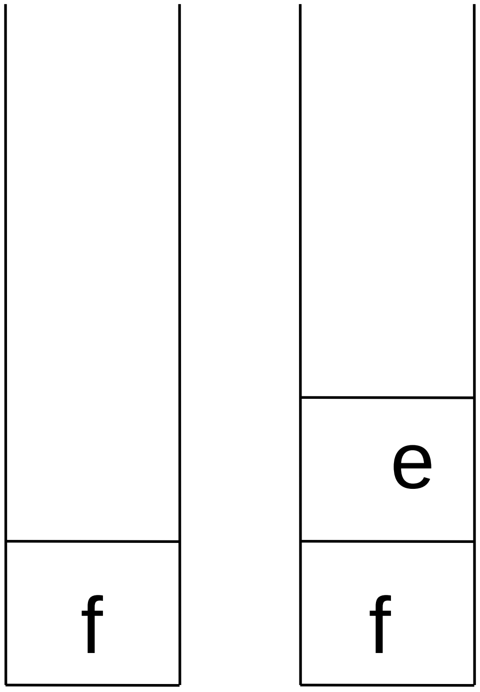
Example

- \+ * + a b + c d + e f.



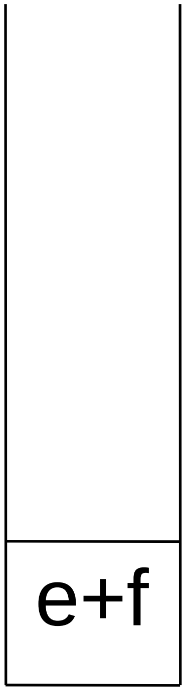
Example

- \+ * + a b + c d + e f



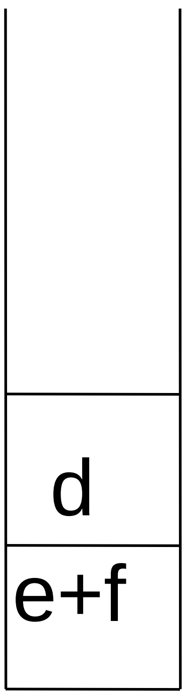
Example

- \+ * + a b + c d + e f.



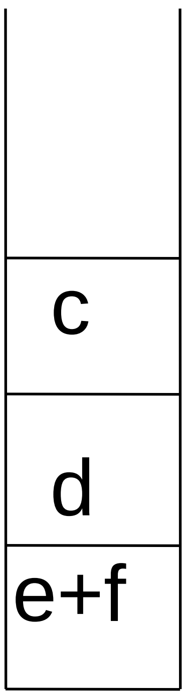
Example

- \+ * + a b + c d + e f.



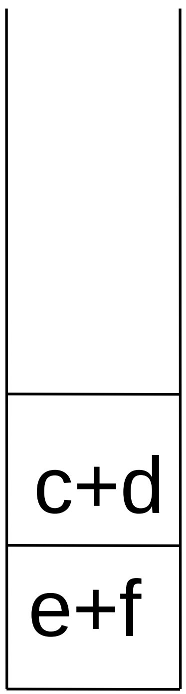
Example

- \+ * + a b + c d + e f.



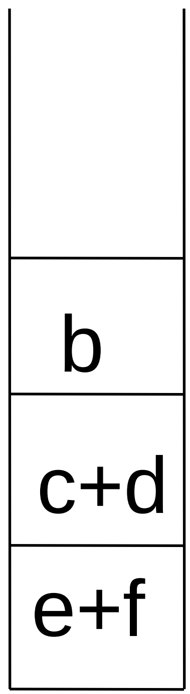
Example

- \+ * + a b + c d + e f.



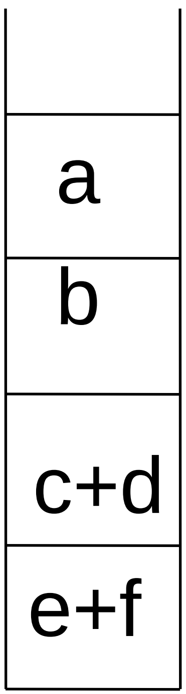
Example

- \+ * + a b + c d + e f.



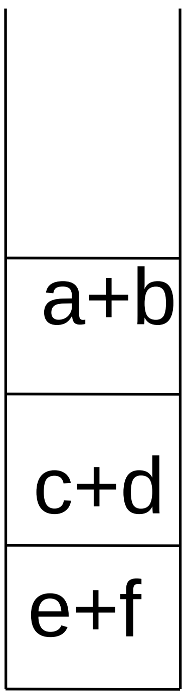
Example

- \+ * + a b + c d + e f.



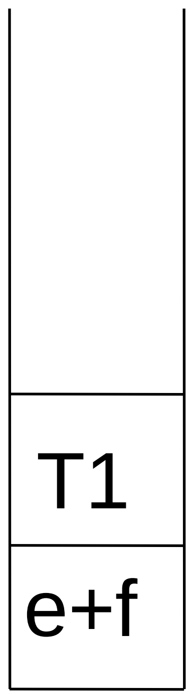
Example

- \+ * + a b + c d + e f.



Example

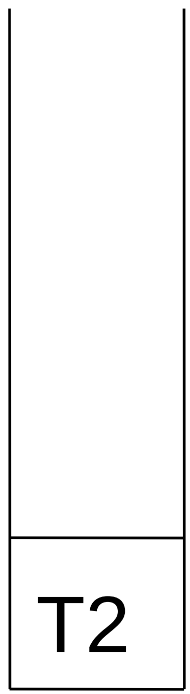
- \+ * + a b + c d + e f.



$$T1 = (a+b) * (c+d)$$

Example

- \+ * + a b + c d + e f.



T1 = (a+b) * (c+d)

T2 = (T1) + (e+f)

Practice Problems

- Evaluate the following prefix expressions. Use a stack and show all your work.

+ 4 + + * 2 2 * 5 1 6

* 2 + * 3 2 + 6 1

Reading Exercise

- We omitted a few details in our description.
- Some of them are:
  - How to handle unary operators?
  - How can this be extended to ternary operators?
- Another possibility is to use postfix expressions.
  - Also called as Reverse Polish Notation.
- They can be evaluated left to right with a stack.
- Try to arrive at the details.

Back to The First Question

- Let us now consider how to convert a given infix expression to its prefix equivalent.
- The issues
  - Operands not easily known
  - There may be parentheses also in the expression.
  - Operators have precedence.

Our Solution

- Consider the example a + b * c + d.
  - The correct evaluation is given by a + (b*c) + d.
- We want to process from right to left.
- We encounter operands and operators.
  - How should each be handled?
- We want to write operands instantly, but operators have to wait for their left operand.

Our Solution

- Let us consider storing operators in the stack.
- They have to wait for their operands to be available.
- How about parentheses?
  - The closed parentheses indicates that some sub-expression is complete.

Our Solution

- Let us consider our example a + b * c + d again.
- When we see the first operand “d” we do not know the other operand or the operator.
- But can still write “d” to the output.
- When we see the “+”, cannot write it to the output yet.
  - The other operand must also be known before the operator is written.

Our Solution

- Have to remember this + somewhere.
  - How long?
- Is “c” the other operand for +?
- Not so, b*c is the operand.
- So have to wait on + for a while.
- When we read “c”, we can output it.
- Next we see the “*” operator. Even for this, only one operand (“c”) is known.
- So have to remember this *.

Our Solution

- Continuing further, we can write the operand “b” the output.
- Seeing “+”, we have to realize that
  - * has higher precedence than +,
  - The operands for * are complete.
- So can print the operator * right away.
- What about the other + that we postponed earlier?

Our Solution

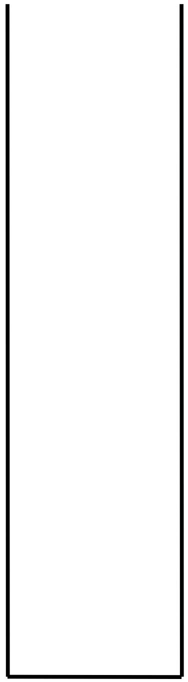
- Depends on the precedence of +.
- For operators of equal precedence, have to look at associativity of the operator.
- Where do we store the operators?
  - Since we are getting to print in the reverse order from what we see,
  - can use a stack to store these.

A Complete Example

- Let us consider an expression of the form a + b + c * d + e * f.

A Complete Example

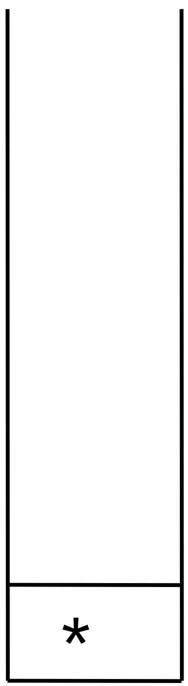
a + b + c * d + e * f



f

A Complete Example

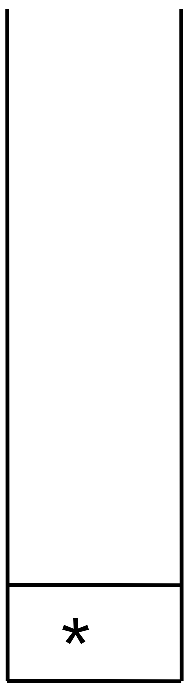
a + b + c * d + e * f



f

A Complete Example

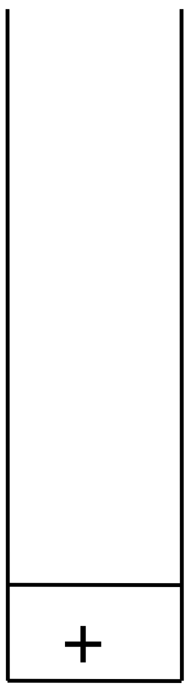
a + b + c * d + e * f



f e

A Complete Example

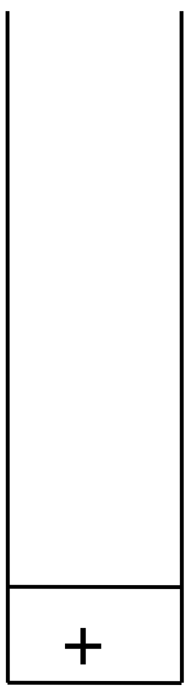
a + b + c * d + e * f



f e *

A Complete Example

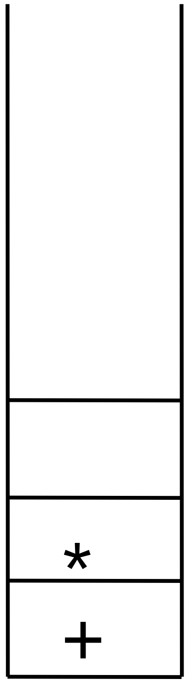
a + b + c * d + e * f



f e * d

A Complete Example

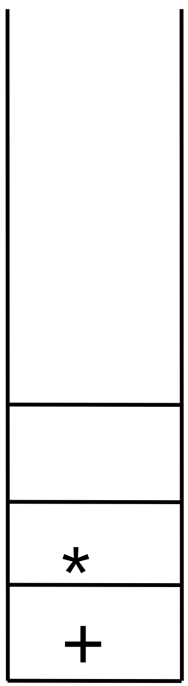
a + b + c * d + e * f



f e * d

A Complete Example

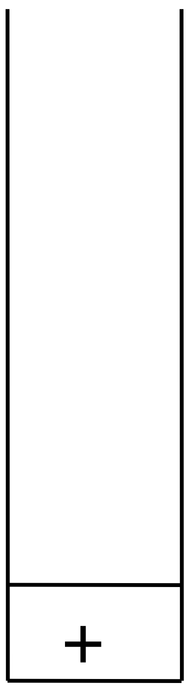
a + b + c * d + e * f



f e * d c

A Complete Example

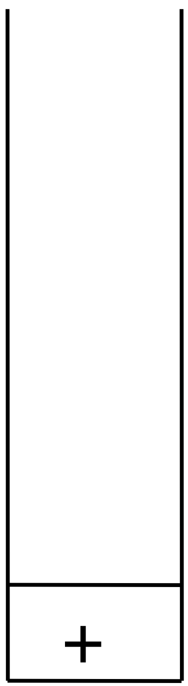
a + b + c * d + e * f



f e * d c *

A Complete Example

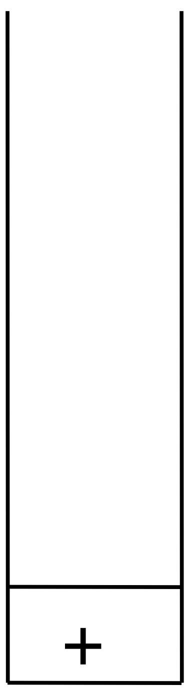
a + b + c * d + e * f



f e * d c * +

A Complete Example

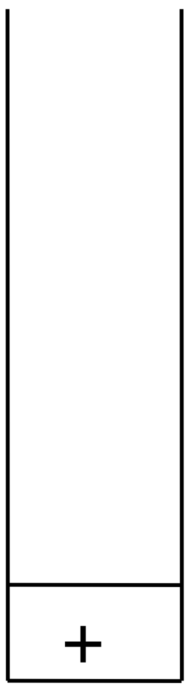
a + b + c * d + e * f



f e * d c * + b

A Complete Example

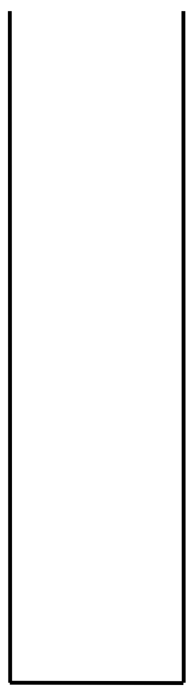
a + b + c * d + e * f



f e * d c * + b +

A Complete Example

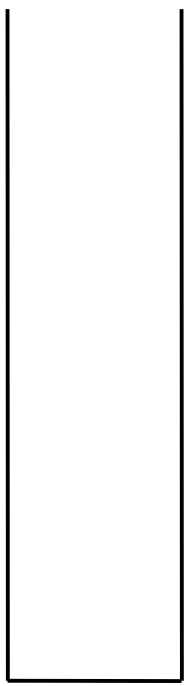
a + b + c * d + e * f



f e * d c * + b + a +

A Complete Example

a + b + c * d + e * f



f e * d c * + b + a +

Invert as:

+ a + b + * c d * e f

Practice Problems

- Convert the following infix expressions to their equivalent prefix.

a + b * c + d * e + f

a + b * c * d + e / f – g

Reading Exercise

- Read or devise ways to handle parentheses.
  - Open parentheses indicates the start of a subexpression, closing parentheses indicates the end of the subexpression.
  - Important to keep track of these.

- Similarly, how to handle unary operators?

Further Applications of the Stack

- Stack used to support recursive programs.
  - Need to store the local variables of every recursive call.
  - Recursive calls end in the reverse order in which they are issued.
  - So, can use a stack to store the details.
- How to verify if a given string of) and (are well-matched?
  - Well matched means that for every (there is a) and
  - A) does not come before a corresponding (.
  - How can we use a stack to solve this problem?

Yet Another Data Structure

- Consider a different setting.
- Think of booking a ticket at a train reservation office.
  - When do you get your chance?
- Think of a traffic junction.
  - On a green light, which vehicle(s) go(es) first.?
- Think of airplanes waiting to take off.
  - Which one takes off first?

Yet Another Data Structure

- All the above scenarios suggest a order of processing data.
  - The order is First-In-First-Out (FIFO)
- We propose a data structure for handling these situations.
  - The name of the data structure is the Queue.

The Queue

- The fundamental operations for such a data structure are:
  - Create : create an empty queue
  - Insert : Insert an item into the queue
  - Delete : Delete an item from the queue.
  - size : return the number of elements in the queue.

The Queue

- Can use an array also to implement a queue.
- We will show how to implement the operations.
  - We will skip create() and size().
- We will store two counters : front and rear
- Insertions happen at the rear
- Deletions happen from the front.

The Queue Routines

Algorithm Insert(x)

begin

if rear == MAXSIZE then
   return ERROR;

Queue[rear] = x;

size = size + 1;

rear = rear + 1;

end

Algorithm Delete()

begin

if size == 0 then return
   ERROR;

size = size - 1;

return Queue[front++];

end

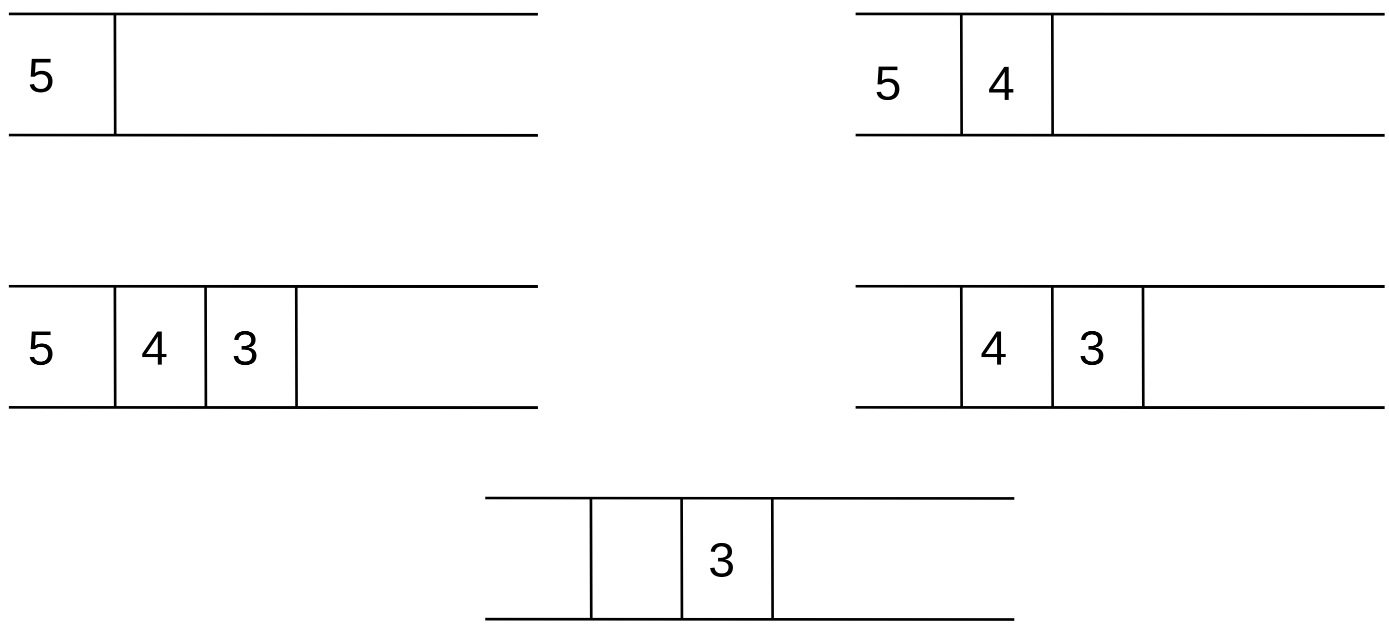
Some Conventions

- Normally, a queue is drawn horizontally
- The front is towards the left, and the rear is towards the right.
- Notice that after a delete, that index is left empty.
- The queue is declared full when rear reaches a value of n.

Queue Example

- Starting from an empty queue, consider the following operations.
  - Insert(5), Insert(4), Insert(3), Delete(), Delete()
- The result is shown in the figure above.

Queue Example -- Solution



- Starting from an empty queue, consider the following operations.
  - Insert(5), Insert(4), Insert(3), Delete(), Delete()
- The result is shown in the figure above.

Other Variations to the Queue

- To save space, a circular queue is also proposed.
- Operations that update front and rear have to be based on modulo arithmetic.
- The circular queue is declared full only when all indices are occupied.

A Sample Application with Stack and Queue

- A palindrome is a string that reads the same forwards and backwards, ignoring non-alphabetic characters.
- Examples:
  - Malayalam
  - Wonton? not now
  - Madam, i'm Adam
- Problem: Given a string, determine if it is a palindrome.
  - May not know the length of the string apriori.

Sample Application

- Need to compare the first character with the last character.
- So, store the characters in a stack and a queue also.
- Once notified of the end of the string, compare the top of the stack with the front of the queue.
  - Continue until both the stack and the queue are empty.

Other Applications of Queue

- A packet queue.
- Several graph algorithms
  - These are advanced applications. We'll study graphs later.