Abstract Data Types

Section 1.3

Abstract Data Types

The Metaphors of Data

Abstract Data Types

- Arrays and lists are both "data structures"
- They are methods of storing and organizing data
- Depending on how data is accessed, arrays and linked lists have areas where they excel and falter

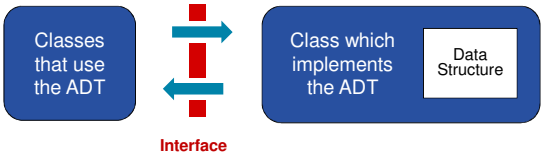
Abstract Data Types

- An *abstract data type* (ADT) is an abstraction of a data structure
- It is not a data structure
- ADT specifies 3 things:
  - the data that will be stored
  - different operations on the data
  - what errors will occur during an operation

More Terminology

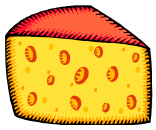
- An ADT is implementation independent
  - can implemented with different data structures – array, linked list, etc…
  - depending how the ADT works, some are better than others
- So, an ADT basically defines an *interface* for a type of data, not how it is actually stored

Data Structures



Example ADT: Cheese Trader

- Data stores orders of cheese
- The operations supported are
  - Order buy (cheese, total, price)
  - Order sell (cheese, total, price)
  - cancel (Order)
- Error conditions:
  - nonexistent cheese
  - price is a negative value
  - total is not greater than 0



Bags

Just toss it in

Bags

- A bag is one of the most simplistic ADT that stores multiple objects
- It can only add items
- Order doesn't matter – nor is it expected to be maintained

Bags

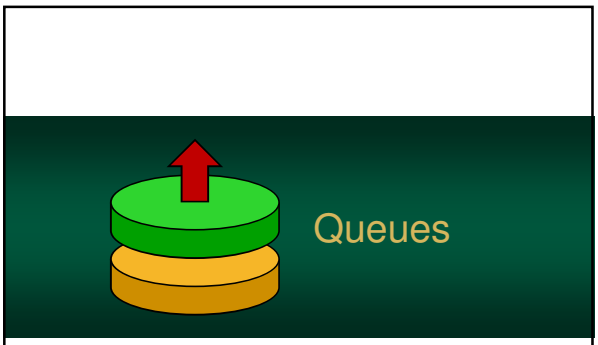
- At is core, the class only requires one method (add)
- Other attributes, such as size, count, etc… and be inferred from return types (i.e. null)

Bag ADT (Typical)

- void Add(object)
- boolean IsEmpty()
- int Size()

Bag Summary

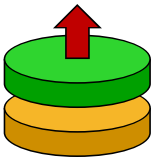
Operation	Fixed Array	Resizable Array	Resizable Array (doubling)	Singly-Linked List
Add()	O(1)	O(n)	O(n) Worst O(1) Best O(1) Average	O(1)



Conga-line of Data!

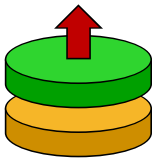
Queues

- The Queue ADT stores arbitrary objects
- Based on the concept of a line – like what you do when you buy groceries
- Objects enter the back of the line, and have to wait for prior items to leave before they do



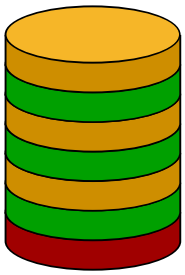
Queues

- In most parts of the World, they call a "line" a "queue"
- Main queue operations:
  - enqueue (object): place on item on the queue
  - dequeue: removes and returns the first inserted object



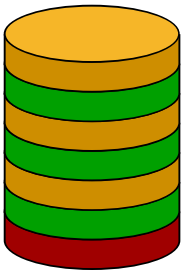
Queue Operation: Enqueue

- When an object is "enqueued", it is put on to the end of the queue
- The items on the top of the queue are not covered



Queue Operation: Dequeue

- Dequeue removes the item from the front of the queue
- Second item becomes the new first item
- This gives a first-in-first-out logic (aka FIFO)



Auxiliary Queue Operations

- Queues also tend to have some operations defined
- These are not necessary, but they are useful
- Auxiliary operations:
  - peek: return the next object without removing it. This is also sometimes called "front"
  - size: returns the number of objects on the queue
  - isEmpty: indicates whether the queue contains no objects. This is an alterative to size()

Queue ADT (Typical)

- void Enqueue(object)
- object Dequeue()
- boolean isEmpty()
- int Size()

Examples of Queues

- Data being read from the keyboard or a file
- People waiting in line to go on Space Mountain
- Instructions on how to perform a task

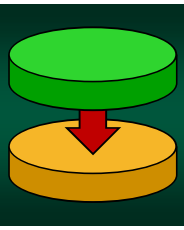
20

Array-Based Queues

- Array-based queues have the same attributes as array-based stacks
- Resizing for every enqueue/dequeue is extremely time expensive
- So, the queue can use doubling or resizing by a specific # of cells

Queue Summary

Operation	Fixed Array	Resizable Array	Resizable Array (doubling)	Singly-Linked List
Enqueue()	O(1)	O(n)	O(1)	O(1)
Dequeue()	O(1)	O(n)	O(n) Worst O(1) Best O(1) Average	O(1)
Peek()	O(1)	O(1)	O(1)	O(1)

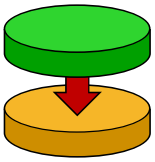


Stacks

Piles of… Data

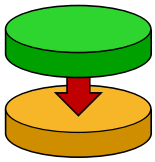
Stack

- A stack is an abstract data structure that stores objects
- Based on the concept of a stack of items – like a stack of dishes
- Data can only be added to or removed from the top of the stack



Stack

- This gives a first-in-last-out logic (aka FILO)
- Same concept is also called last-in-first-out (LIFO)



Stack ADT (Typical)

- void Push(object)
- object Pop()
- boolean isEmpty()
- int Size()

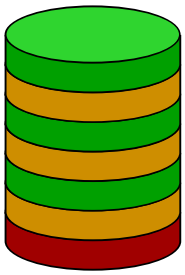
Examples of Stacks

- Page-visited "back button" history in a web browser
- Undo sequence in a text editor
- Deck of cards in Windows Solitaire

27

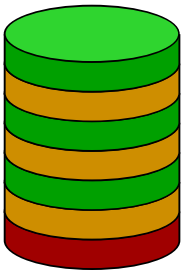
Stack Operation: Push

- A value is added to the stack
- It is placed on the top location
- Rest of the items are "covered"



Stack Operation: Pop

- Removes an item from the stack
- Last item added is removed
- 2nd item becomes the top



Auxiliary Stack Operations

- Stacks also tend to have some operations defined
- Although these are not necessary, they are useful
- Auxiliary operations:
  - top: return the last pushed object without removing it
  - size: returns the number of objects on the stack
  - isEmpty: indicates whether the stack contains no objects. This is an alterative to size()

Stacks: Error Conditions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty

Array-Based Stack

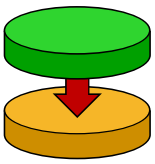
- An array is a simple way of implementing the Stack ADT
- Add elements from left to right
- A variable keeps track of the index of the top
- The stack may become full creating an exception

Resizing an Array-Based Stack

- Resizing the stack is expensive
  - looking through all the elements of a stack size *n* is O(n)
  - So, every time a stack is resize in memory, it costs O(n) to copy the old buffer to the new
- Approach:
  - grow/shrink by a specific # of elements
  - doubling: double the array on resize

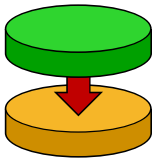
Fixed-Capacity Stacks

- Memory is finite and, often, stacks need to be finite
- A *fixed-capacity stack* has a finite number of items that it can store
- This is its *capacity*



Fixed-Capacity Stacks

- It can be implemented by keep track of a capacity value (usually an int)
- The stack would behave as normal until the capacity is reached
- In this case, one of two things will happen…



Full Fixed-Capacity Stack…

1. Stack throws an *Overflow Error*
2. Stack discards an object
   - the bottom of the stack is typically removed
   - this gives the space needed for the newly pushed object
   - e.g. the history feature of your web browser

Array-Based Fixed-Capacity Stack

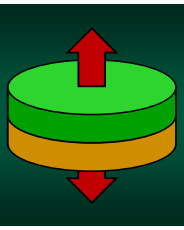
- While using an array for a normal stack (no fixed capacity) has a number of drawbacks
- … for fixed-capacity, an array is an excellent choice

Array-Based Fixed-Capacity Stack

- Once the end of the array is reached, the top moves to the start of the array
- And the bottom moves as well
- The array is never resized or copied

Stack Summary

Operation	Fixed Array	Resizable Array	Resizable Array (doubling)	Singly-Linked List
Pop()	O(1)	O(n)	O(1)	O(1)
Push()	O(1)	O(n)	O(n) Worst O(1) Best O(1) Average	O(1)
Top()	O(1)	O(1)	O(1)	O(1)

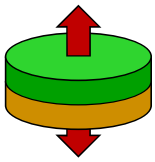


The Deque ADT

Time to shuffle the "deck"

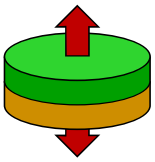
Deque ADT

- There is a variant of the queue called a deque (pronounced "deck")
- The name is derived from double-ended queue (sometimes it is shorted more to DQ)



Deque ADT

- As the name implies, its queue allows insertions and removals from both ends
- It is a merging of a stack and queue data ADT and the basic operations are union of the two



Deque ADT (Typical)

- addFirst (object)
  - place an object on the front of the deque
  - this is same as stack "push"
  - sometimes this is called "offerFirst"
- addLast (object)
  - place an object on the end of the deque
  - this is the same as queue "enqueue"
  - sometimes this is called "offerLast"

Deque ADT (Typical)

- removeFirst ()
  - remove an object from the front of the deque
  - this is the same as queue "dequeue"
  - sometimes this is called "pollFirst"
- removeLast ()
  - this is unique – and not found in either a stack or queue ADT
  - sometimes this is called "pollLast"

Deque Example

```
1. addFirst('N')
2. addLast('E')
3. addFirst('W')
4. addLast('D')
5. addFirst('P')
```

Deque Auxiliary Operations

- size()
- isEmpty()
- peekFirst()
  - return the first object in the deque without removing it
  - this is also sometimes called "front"
- peekLast()
  - return the last object in the deque without removing it
  - this is also sometimes called "last"

Advantages & Disadvantages

- Advantages
  - can function and either a stack or queue
  - "Add First" operation can be used to "redo" or "undo" a queue removal – put it back in line
- Disadvantages
  - Stacks/Queues can be created with a single-linked-list, *a Deque requires a double-linked-list*
  - otherwise, removing items from the end of the list would require O(n) – even with an end pointer

Stack & Queues in Practice

1001 Uses! (I meant 1,001 – not 9)

HTML Tag Matching

- HTML is a hierarchical structure
- HTML consists of tags
  - each tag can also embed other tags
  - allows text to be aligned, made bold, etc…

HTML Tag Matching

- Web browsers read the text and apply a tag depending if it is active
- They maintain a stack…
  - push a start tag, pop and end tag
  - if the HTML is correct, they should match
  - *… with the exception of the unary tags*

HTML Tag Matching

```
<html>
<body>
<center>
<h1>Banks of Sacramento</h1>
</center>
Sing and heave, and heave and sing,<br>
<i>To me hoodah! To my hoodah!</i><br>
Heave and make the handspikes spring.<br>
<i>To me hoodah! To me hoodah!</i>
<br>
<i>And it's blow, boys, blow,<br>
For Californi-o.<br>
For there's plenty of gold, <br>
So I've been told,<br>
On the banks of the Sacramento.</i><br>
</body>
</html>
```

Banks of Sacramento

Sing and heave, and heave and sing,
To me hoodah! To my hoodah!
Heave and make the handspikes spring.
To me hoodah! To me hoodah!

And it's blow, boys, blow,
For Californi-o.
For there's plenty of gold,
So I've been told,
On the banks of the Sacramento.

Balanced Parentheses

- When analyzing arithmetic expressions…
  - it is important to determine whether it is balanced with respect to parentheses
  - otherwise, the expression is incorrect
- A great solution is a stack
  - push each `(` and pop each `)`
  - at the end, the stack should be empty
  - also, if you attempt to pop on an empty stack, the expression is invalid

Balanced Parenthesis Examples

`(a + b)`	Balanced
`(a + b))`	Pop empty stack
`) a + b (`	Pop empty stack
`(a + (b + 1) * c) / e`	Balanced
`(a * (b + ((d + e) * f))`	Stack has 1 left

Balanced Parentheses

- *But wait…*
  - can't we just keep a "parenthesis level" counter?
  - if it is `>= 1` at the end or it ever becomes **0**, the expression is invalid
- However…
  - some expressions allow curly and square brackets
  - a simple counter is insufficient
  - stack can check if the pop'd item matches

Balanced Parenthesis Examples

`[a + b]`	Balanced
`(a + b}`	Mismatch
`{[ a + b })`	Mismatch
`(a + (b + 1) * c / e`	Unbalanced
`(a * [b + {c + d} * e])`	Balanced

Evaluating Expressions

A Stack and Queue working together!

Evaluating Expressions

- It is a common task in programs to evaluate mathematical expressions and get a result
- Computers can perform this task using an algorithm created by Dijkstra, but we will get into that later

Evaluating Expressions

- First, we need to look at mathematical expressions
  - we commonly using infix notation which is not stack or queue "friendly"
  - there are, however, two alternative notations
  - one of which is stack friendly

Infix Notation

- Using *infix notation*, we put the operating in between the two operators
- This is the standard format used today

To add the numbers *a* and *b*, we type:	`a + b`
To divide *a* by *b*, we type:	`a / b`

Prefix Notation

- *Prefix notation*, rather than putting the operator between the operands, puts it first
- It is also called *"Polish Notation"*
- Used by the LISP programming language

To add the numbers *a* and *b*, we type:	`+ a b`
To divide *a* by *b*, we type:	`/ a b`

Postfix Notation

- *Postfix notation* puts the operator at the end
- Also called *"Reverse Polish Notation" (RPN)*
- Since the operator is last, we can also use it as a "trigger" to perform math

To add the numbers *a* and *b*, we type:	`a b +`
To divide *a* by *b*, we type:	`a b /`

Where are My Parenthesis?

Infix	Prefix	Postfix
`a + b * c`	`+ a * b c`	`a b c + *`
`(a - b) * c`	`- a b * c`	`a b - c *`
`(a / (b - c) + d)`	`+ / a - b c d`	`a b c - / d +`
`(a + b / (c - d))`	`+ a / b - c d`	`a b c d - / +`

Where are My Parenthesis?

- Infix is the only notation that needs parentheses to change precedence
- The order of operators handles precedence in prefix and postfix

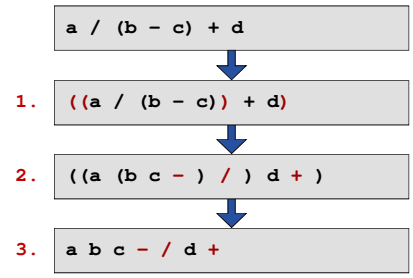
Converting to Prefix or Postfix

- Why are learning this... *be patient!*
- Converting from infix to postfix or prefix notation is easy to do by hand
- Did you notice that the operands did not change order? They were always *a, b, c…*
- We just need to rearrange the operators

Convert Infix to Prefix / Postfix

1. Make it a *fully parenthesized expression (FPE)* - one pair of parentheses enclosing each operator and its operands
2. Move the operators to the start (prefix) or end (postfix) of each sub-expression
3. Finally, remove all the parenthesis

Infix to Postfix



Compute Postfix Algorithm

- Computing a postfix expression is *unbelievably* easy using a stack
- All you need is a one queue of symbols (numbers, operators) and one stack
- In fact, on classic Hewlett Packard calculators, all operations are stack based

Compute Postfix Pseudo-code

```
while there is data in the input queue
   read a token (number or operator) from queue
   if it's a number, push it on the stack
   if it's an operator
      pop two numbers from the stack
      compute the result (using the operator)
      push the result on the stack
   end if
end while

//Afterwards, the final result is on the stack
```


Compute Postfix Demo

Input Queue: 24 10 7 – / 34 +

Stack:

Compute Postfix Demo

Input Queue: 10 7 – / 34 +

Stack: 24

Compute Postfix Demo

Input Queue: 7 – / 34 +

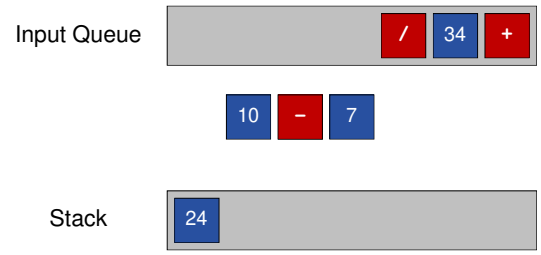
Stack: 24 10

Compute Postfix Demo

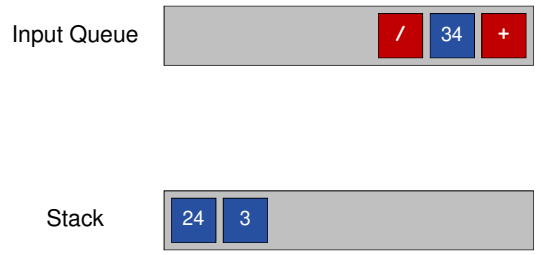
Input Queue: – / 34 +

Stack: 24 10 7

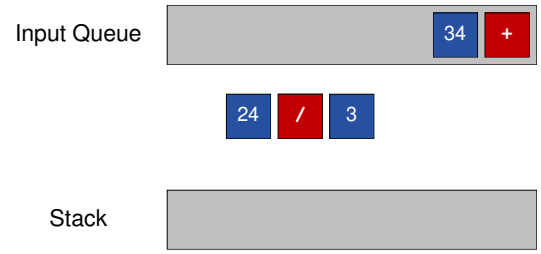
Compute Postfix Demo



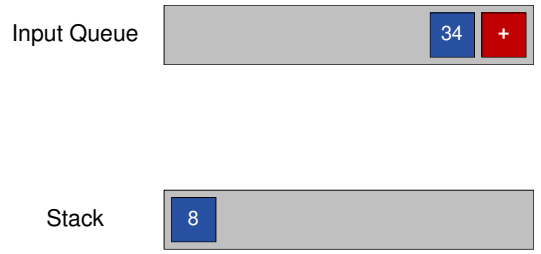
Compute Postfix Demo



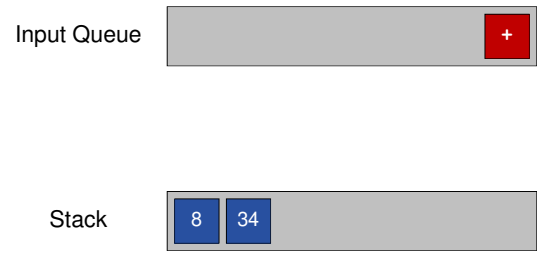
Compute Postfix Demo



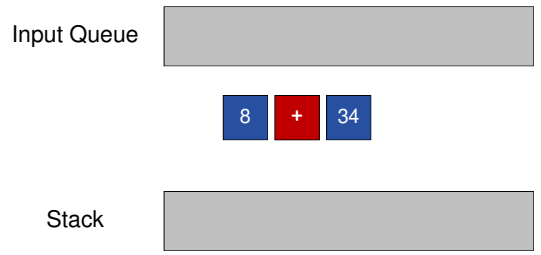
Compute Postfix Demo



Compute Postfix Demo



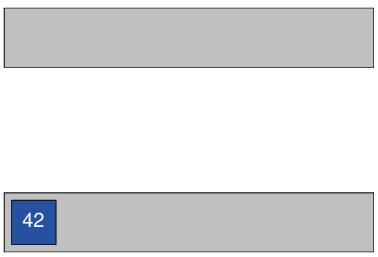
Compute Postfix Demo

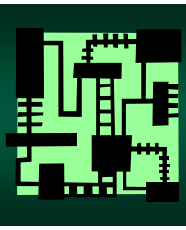


Compute Postfix Demo

Input Queue

Stack





Infix to Postfix Algorithm

Let the computer do the work…

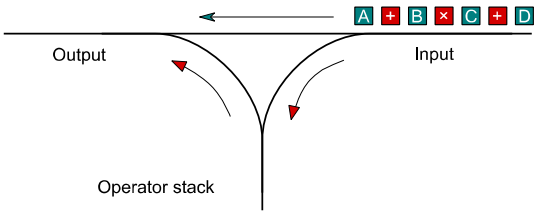
Infix to Postfix Algorithm

- Infix expressions need to be converted to postfix to be evaluated
- Dijkstra's *Shunting-yard algorithm* performs this task

Shunting-yard algorithm

- Named after railroad shunting yards – which move trains onto different tracks
- Dijkstra's solution uses an input queue, operator stack, and output queue

Shunting-yard Algorithm



Shunting-yard Algorithm

- The most basic version of this algorithm requires *Fully-Parenthesized Expression (FPE)*
- This means, there is no precedence and parenthesis are put around every operator

FPE Shunting-yard Algorithm

```
while the input queue has tokens
   read a token from the input queue
   if the token is a…
      operand : add it to output queue
      operator : push it on the stack
      '(' : push it onto the stack
      ')' :
         while the top of stack isn't a '('
            pop an operator
            add it to the output queue
         end while
         pop and discard the extra '('
   end if
end while
```


FPE Shunting-yard Algorithm

Input Queue	((a * (b + c)) / d)
Operator Stack	
Output Queue	

FPE Shunting-yard Algorithm

Input Queue	(a * (b + c)) / d)
Operator Stack	(
Output Queue	

FPE Shunting-yard Algorithm

Input Queue	a * (b + c)) / d)
Operator Stack	((
Output Queue	

FPE Shunting-yard Algorithm

Input Queue	* (b + c)) / d)
Operator Stack	((
Output Queue	a

FPE Shunting-yard Algorithm

Input Queue	(b + c)) / d)
Operator Stack	((*
Output Queue	a

FPE Shunting-yard Algorithm

Input Queue	`b + c ) ) / d )`
Operator Stack	`( ( * (`
Output Queue	`a`

FPE Shunting-yard Algorithm

Input Queue	`+ c ) ) / d )`
Operator Stack	`( ( * (`
Output Queue	`a b`

FPE Shunting-yard Algorithm

Input Queue	`c ) ) / d )`
Operator Stack	`( ( * ( +`
Output Queue	`a b`

FPE Shunting-yard Algorithm

Input Queue	`) ) / d )`
Operator Stack	`( ( * ( +`
Output Queue	`a b c`

FPE Shunting-yard Algorithm

Input Queue	`) / d )`
Operator Stack	`( ( *`
Output Queue	`a b c +`

FPE Shunting-yard Algorithm

Input Queue	`/ d )`
Operator Stack	`(`
Output Queue	`a b c + *`

FPE Shunting-yard Algorithm

Input Queue	`d )`
Operator Stack	`( /`
Output Queue	`a b c + *`

FPE Shunting-yard Algorithm

Input Queue	`)`
Operator Stack	`( /`
Output Queue	`a b c + * d`

FPE Shunting-yard Algorithm

Input Queue	
Operator Stack	
Output Queue	`a b c + * d /`

Too Many Paranthesis!

- FPE's are rarely used in real-World examples
- In fact, we use precedence rules to simplify expressions
- Fortunately, the algorithm can be modified, very easily, to handle precedence!

FPE Shunting-yard Algorithm

```
while the input queue has tokens
   read a token from the input queue
   if the token is a…
      operand : add it to output queue
      operator : new rules – see next slide
      '(' : push it onto the stack
      ')' :
         while the top of stack isn't a '('
            pop an operator
            add it to the output queue
         end while
         pop and discard the '('
   end if
end while
```


Operator: New Rules

- When you read an operator from the input queue….
- … go into a loop that looks at the top of the stack and compares its precedence to the current operator
- If the current operator is …
  - left-associative, pop while the top is **>=**
  - right-associative, pop while the top is **>**

Shunting-yard Algorithm Operators

- Stop if you hit a '('
- Each pop'd operator is put directly on the output queue
- Finally, push the current operator onto the stack

Operator Associatively

Operator	Associatively
`+ - * /`	Left
`^` (exponent)	Right

Shunting-yard Algorithm Example 1

Input Queue	`a - b * c + d`
Operator Stack	
Output Queue	

Shunting-yard Algorithm Example 1

Input Queue	`- b * c + d`
Operator Stack	
Output Queue	`a`

Shunting-yard Algorithm Example 1

Input Queue	`b * c + d`
Operator Stack	`-`
Output Queue	`a`

Shunting-yard Algorithm Example 1

Input Queue	`* c + d`
Operator Stack	`-`
Output Queue	`a b`

Shunting-yard Algorithm Example 1

Input Queue	c + d
Operator Stack	– *
Output Queue	a b

Shunting-yard Algorithm Example 1

Input Queue	+ d
Operator Stack	– *
Output Queue	a b c

Shunting-yard Algorithm Example 1

Input Queue	d
Operator Stack	– * +
Output Queue	a b c

The precedence of * – are both >= than +

Shunting-yard Algorithm Example 1

Input Queue	d
Operator Stack	+
Output Queue	a b c * –

The precedence of * – are both >= than +

Shunting-yard Algorithm Example 1

Input Queue	
Operator Stack	+
Output Queue	a b c * – d

Shunting-yard Algorithm Example 1

Input Queue	
Operator Stack	
Output Queue	a b c * – d +

Remaining stack items pop'd

Shunting-yard Algorithm Example 2

Input Queue	`a + (b – c * d) / e – f`
Operator Stack	
Output Queue	

Shunting-yard Algorithm Example 2

Input Queue	`+ (b – c * d) / e – f`
Operator Stack	
Output Queue	`a`

Shunting-yard Algorithm Example 2

Input Queue	`(b – c * d) / e – f`
Operator Stack	`+`
Output Queue	`a`

Shunting-yard Algorithm Example 2

Input Queue	`b – c * d) / e – f`
Operator Stack	`+ (`
Output Queue	`a`

Shunting-yard Algorithm Example 2

Input Queue	`– c * d) / e – f`
Operator Stack	`+ (`
Output Queue	`a b`

Shunting-yard Algorithm Example 2

Input Queue	`c * d) / e – f`
Operator Stack	`+ ( –`
Output Queue	`a b`

Shunting-yard Algorithm Example 2

Input Queue	`* d) / e – f`
Operator Stack	`+ ( –`
Output Queue	`a b c`

Shunting-yard Algorithm Example 2

Input Queue	`d) / e – f`
Operator Stack	`+ ( – *`
Output Queue	`a b c`

`–` has a lower precedence than `*`

Shunting-yard Algorithm Example 2

Input Queue	`) / e – f`
Operator Stack	`+ ( – *`
Output Queue	`a b c d`

Shunting-yard Algorithm Example 2

Input Queue	`/ e – f`
Operator Stack	`+`
Output Queue	`a b c d * –`

`)` was read. All items are pop'd until matching `(` found

Shunting-yard Algorithm Example 2

Input Queue	`e – f`
Operator Stack	`+ /`
Output Queue	`a b c d * –`

Shunting-yard Algorithm Example 2

Input Queue	`– f`
Operator Stack	`+ /`
Output Queue	`a b c d * – e`

Shunting-yard Algorithm Example 2

Input Queue: `f`

Operator Stack: `+ /` `-`

The precedence of both / and + are >= than -

Output Queue: `a b c d * - e`

Shunting-yard Algorithm Example 2

Input Queue: `f`

Operator Stack: `-`

Output Queue: `a b c d * - e / +`

Shunting-yard Algorithm Example 2

Input Queue:

Operator Stack: `-`

Output Queue: `a b c d * - e / + f`

Shunting-yard Algorithm Example 2

Input Queue:

Operator Stack:

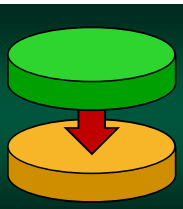
Remaining stack items pop'd

Output Queue: `a b c d * - e / + f -`

Testing Our Result

`a + (b - c * d) / e - f`

1. `((a + ((b - (c * d)) / e)) - f)`
2. `((a ((b (c d *) -)) e /) +) f -)`
3. `a b c d * - e / + f -`

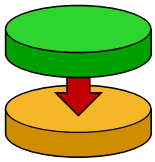


System Stack

How it works? …um, I forgot

The Stack and Heap

- Your computer maintains two distinct types of memory for running programs
- The Stack is used to …
  - store function states
  - this includes local variables
  - you cannot modify the stack – it is hidden

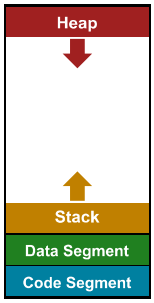


The Stack and Heap

- The Heap is used to …
  - store *dynamic* allocation
  - when you create objects using "new", the heap is used to allocate storage
  - unlike the stack, data persists regardless of function calls
  - system performs garbage cleanup after the memory is no longer needed

How it All Fits Together

- Programs can be seen as 4 different segments
  - Code Segment contains the program instructions
  - Data Segment contains global data
  - Stack grows upwards
  - Heap grows downwards
- The heap will hopefully never run into the stack



Cursors

Okay, now its getting weird

Cursors

- *Cursors* are a melding of the idea of arrays and linked lists
- Cursors want to minimize the constant creation and deletion of new nodes
- So, it maintains an array of unused nodes

Cursors

- Multiple nodes are allocated early - called a *pool*
- If a node is needed, one is removed from the pool
- If a node is removed, and the array has room, it is placed back in the array

The Reason

- Arrays can be wasteful …
  - in space – when there are partially filled arrays
  - in time – created and destroyed frequently
- Linked lists can be wasteful…
  - require memory to be allocated each time a node is created
  - puts a lot of work on the heap

Even more approaches

- You can also use another "pool" linked list
- So, your Linked List class
  - would have a linked list of valid nodes
  - and another list of unused notes
  - the danger here is that you don't limit the size of the pool – and it grows ***forever***
  - so, if you use two linked lists, keep a pool member count