Group Practice — make this language LL(1)...

#	Rules
1	$S \rightarrow A\ B\ \$$
2	$S \rightarrow B\ C\ \$$
3	$A \rightarrow A\ x$
4	$A \rightarrow x$
5	$B \rightarrow y\ A\ B$
6	$B \rightarrow h$
7	$C \rightarrow x\ C\ y$
8	$C \rightarrow p$

#	$p \in P$	Computed By	Predict Set
1	$S \rightarrow A\ B\ \$$	$FirstSet(RHS)$	x
2	$S \rightarrow B\ C\ \$$	$FirstSet(RHS)$	h, y
3	$A \rightarrow A\ x$	$FirstSet(RHS)$	x
4	$A \rightarrow x$	$FirstSet(RHS)$	x
5	$B \rightarrow y\ A\ B$	$FirstSet(RHS)$	y
6	$B \rightarrow h$	$FirstSet(RHS)$	h
7	$C \rightarrow x\ C\ y$	$FirstSet(RHS)$	x
8	$C \rightarrow p$	$FirstSet(RHS)$	p

	h	p	x	y	$
S	2		1	2	
A			$\star$		
B	6			5	
C		8	7		

Group Practice — make this language LL(1)…

#	Rules
1	$S \rightarrow A\,B\,\$$
2	$S \rightarrow B\,C\,\$$
3	$A \rightarrow A\,x$
4	$A \rightarrow x$
5	$B \rightarrow y\,A\,B$
6	$B \rightarrow h$
7	$C \rightarrow x\,C\,y$
8	$C \rightarrow p$

The language is **not LL(1)** due to the left recursion rule

$$A \rightarrow A\,x$$

You might recall the reformatting equations from a previous lecture:

$$\begin{array}{lcl} A & \rightarrow & A\gamma\beta \\ A & \rightarrow & \beta \end{array} \quad \Rightarrow \quad \begin{array}{lcl} A & \rightarrow & \beta R \\ R & \rightarrow & \gamma\beta R \\ & | & \lambda \end{array}$$

(γ may be “empty,” recall lower Greek letters are $(\Sigma + N)*$)

In this case $\gamma = \lambda$, since we must have a symbol for β.

The following refactoring of A will make this an LL(1) language:

$$\begin{array}{lcl} A & \rightarrow & x\ R \\ R & \rightarrow & x\,R \\ & | & \lambda \end{array}$$

Left Recursion Blemishes on LL(1) Parsing

#	Rules
1	$S \rightarrow A\,B\,\$$
2	$S \rightarrow B\,C\,\$$
3	$A \rightarrow A\,x$
4	$A \rightarrow x$
5	$B \rightarrow y\,A\,B$
6	$B \rightarrow h$
7	$C \rightarrow x\,C\,y$
8	$C \rightarrow p$

Having to avoid left-recursion is a considerable blemish on recursive descent parsing — we want languages to be **expressive**: permitting an idea to be communicated with a minimal syntax and without "structure obfuscation."

Imagine an LL(1) grammar for left associative arithmetic operations! Yuck.

LR(0) Parsing

#	Rules
1	$S \rightarrow A\,B\,\$$
2	$S \rightarrow B\,C\,\$$
3	$A \rightarrow A\,x$
4	$A \rightarrow x$
5	$B \rightarrow y\,A\,B$
6	$B \rightarrow h$
7	$C \rightarrow x\,C\,y$
8	$C \rightarrow p$

	h	p	x	y	\$	A	B	C
0	sh-1		sh-2	sh-3		sh-4	sh-5	
1	Reduce 6							
2	Reduce 4							
3			sh-2			sh-6		
4	sh-1		sh-7	sh-3			sh-8	
5		sh-9	sh-10					sh-11
6	sh-1		sh-7	sh-3			sh-12	
7	Reduce 3							
8					sh-13			
9	Reduce 8							
10		sh-9	sh-10					sh-14
11					sh-15			
12	Reduce 5							
13	Reduce 1							
14				sh-16				
15	Reduce 2							
16	Reduce 7							

Operation: begin

TOP OF STACK	**FRONT OF DEQUE**							
0	y	x	x	h	x	p	y	$

The **DEQUE** is initialized with the input sequence of *tokens*;
the first token at the front (top) of the deque.
State $\boxed{0}$ is pushed onto the **STACK**.

Operation: begin

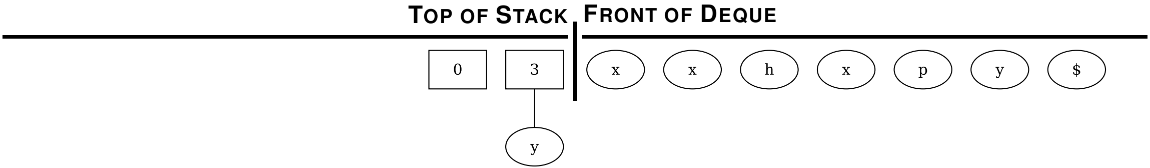
TOP OF STACK	**FRONT OF DEQUE**							
0	y	x	x	h	x	p	y	$

The stack's top is state $\boxed{0}$ and the front of the deque is token `y`,

Using the LR(0) table we look up the **`sh-3`** action

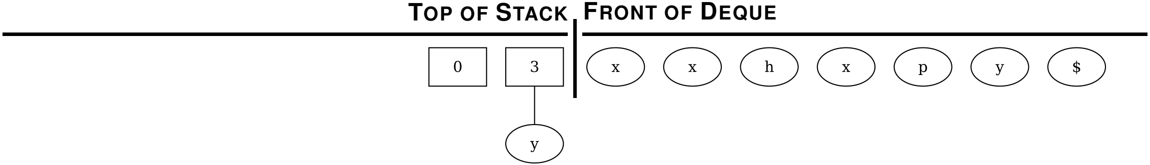
	h	*p*	*x*	*y*	$	*A*	*B*	*C*
0	sh-1		sh-2	sh-3		sh-4	sh-5	
1	Reduce 6							
2	Reduce 4							
⋮								

Operation: shift y to stack, goto state 3



`sh-3` action: push state $\boxed{3}$ onto the stack,
labeled with the token `y` from the front of the deque.

Operation: shift y to stack, goto state 3

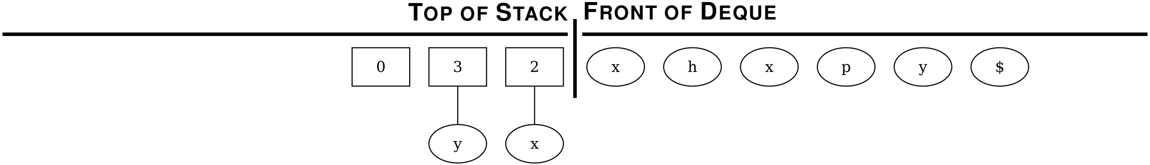


The stack's top is state $\boxed{3}$ and the front of the deque is token `x`,

Using the LR(0) table we look up the **`sh-2`** action

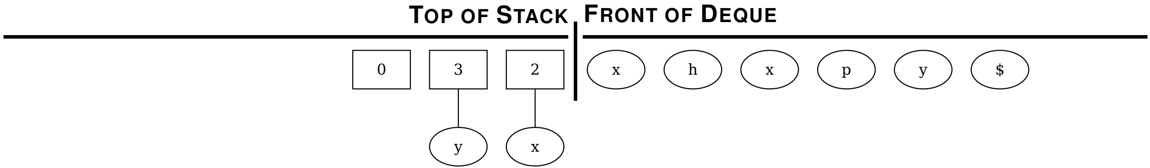
	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
3			sh-2			sh-6		
4	sh-1		sh-7	sh-3			sh-8	
⋮								

Operation: shift x to stack, goto state 2



`sh−2` action: push state $\boxed{2}$ onto the stack, labeled with the token `x` from the front of the deque.

Operation: shift x to stack, goto state 2

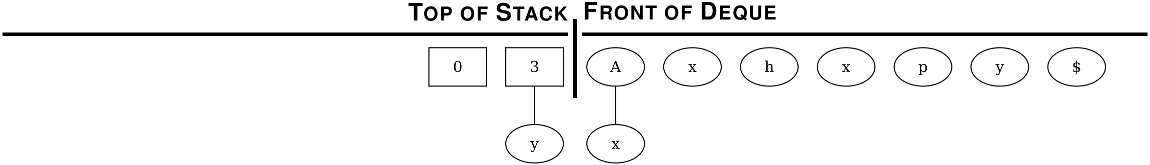


The stack's top is state $\boxed{2}$,

the LR(0) table says we should ***reduce with rule 4***.

	h	p	x	y	$	A	B	C
⋮								
2	Reduce 4							
3			sh-2			sh-6		
⋮								

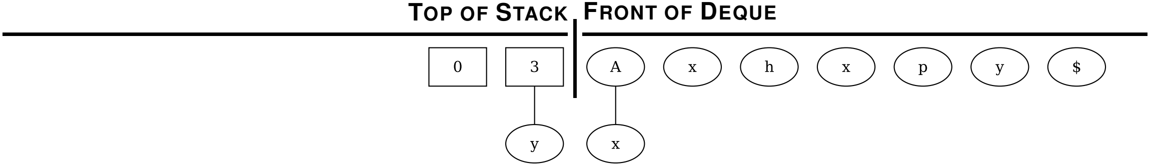
Operation: reduce by rule 4 $A \rightarrow x$



Reduce 4 action: reduce the top-most elements of the stack to be children of **rule 4**'s RHS non-terminal.

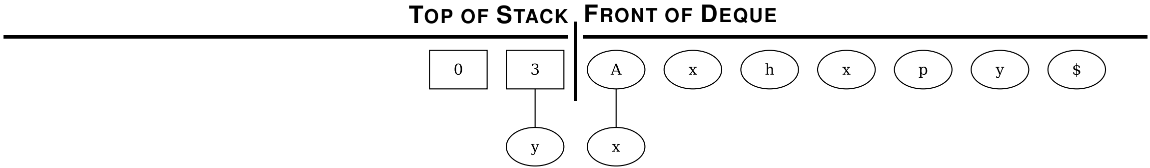
Push this tree back onto the front of the deque.

Operation: reduce by rule 4 $A \rightarrow x$



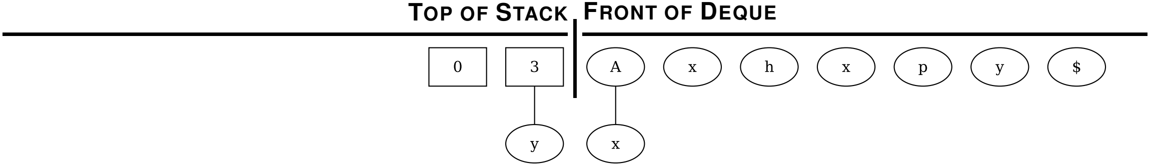
How many elements came off the stack? It depends on the RHS of the reduction rule.

Operation: reduce by rule 4 $A \to x$



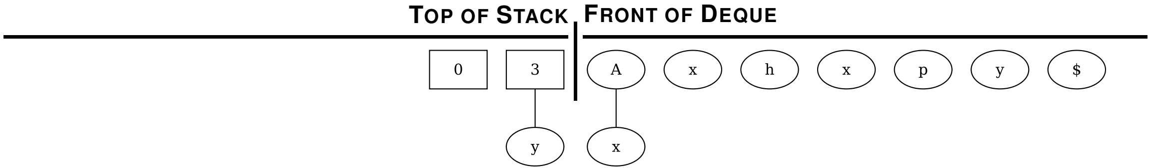
The deque has either **tokens** or **tree roots** as its elements;
depending on the implementation language this may be easy or tedious to accomplish.

Operation: reduce by rule 4 $A \rightarrow x$



Would anyone like to hazard a guess at what we do next?

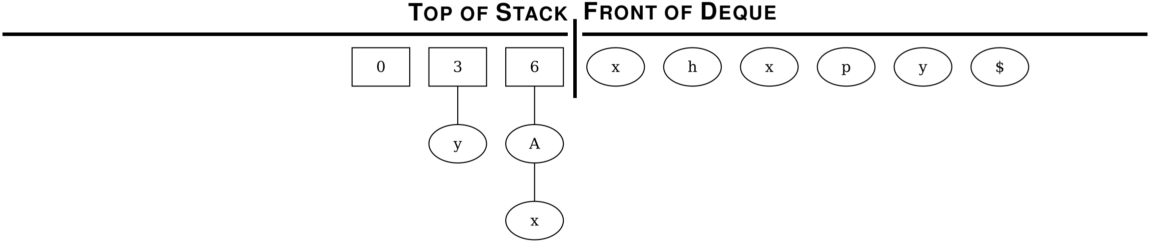
Operation: reduce by rule 4 $A \rightarrow x$



The stack's top is state $\boxed{3}$ and the front of the deque is **non-terminal** A,
Using the LR(0) table we look up the **`sh-6`** action

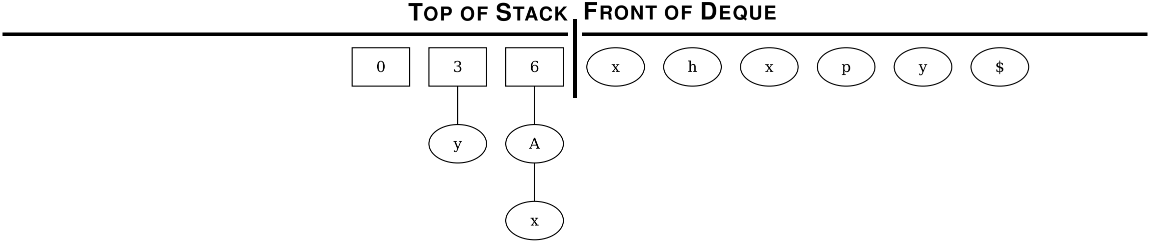
	h	p	x	y	$\$$	A	B	C
⋮								
3			sh-2			sh-6		
4	sh-1		sh-7	sh-3			sh-8	
⋮								

Operation: shift A to stack, goto state 6



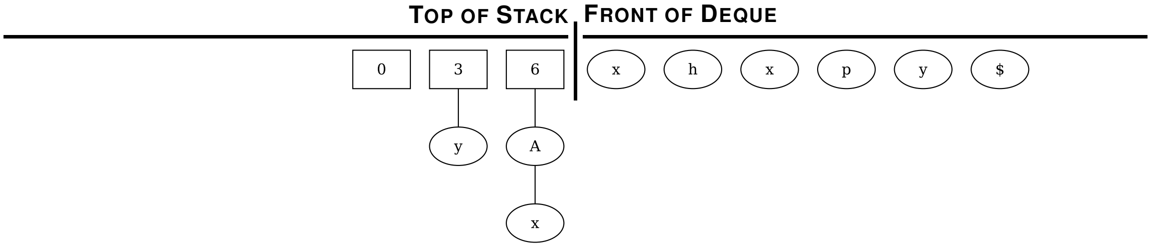
`sh-6` action: push state $\boxed{6}$ onto the stack, labeled with the A **tree** from the front of the deque.

Operation: shift A to stack, goto state 6



The stack always has “state” items in it,
these state items **may have** connected to them *tokens* or *trees*.

Operation: shift A to stack, goto state 6

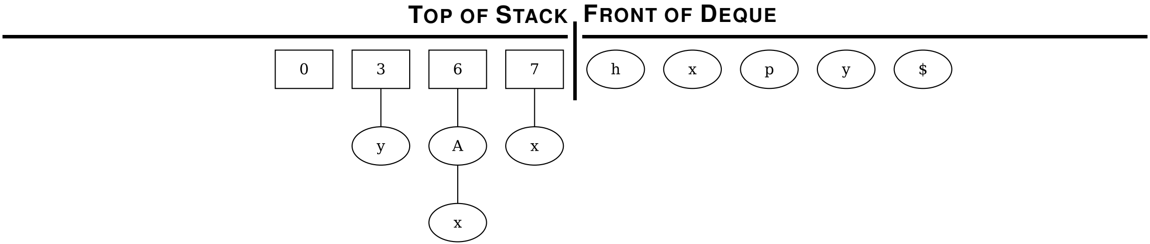


The stack's top is state $\boxed{6}$ and the front of the deque is token `x`,

Using the LR(0) table we look up the **`sh-7`** action

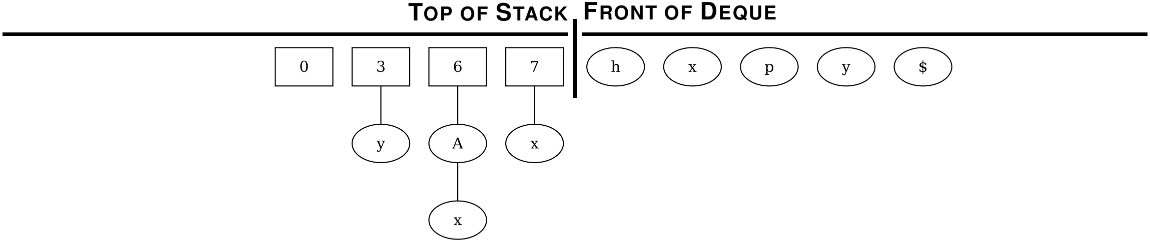
	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
6	sh-1		sh-7	sh-3			sh-12	
7	Reduce 3							
⋮								

Operation: shift x to stack, goto state 7



`sh-7` action: push state $\boxed{7}$ onto the stack,
labeled with the element from the front of the deque.

Operation: shift x to stack, goto state 7

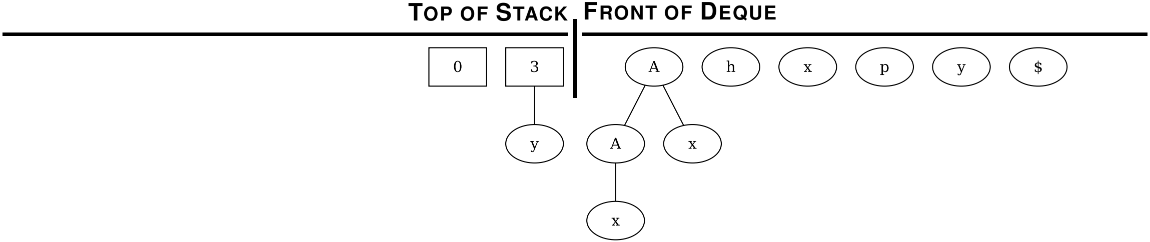


The stack's top is state $\boxed{7}$,

the LR(0) table action is **Reduce 3**

	h	*p*	*x*	*y*	\$	*A*	*B*	*C*
⋮								
7	Reduce 3							
8					sh-13			
⋮								

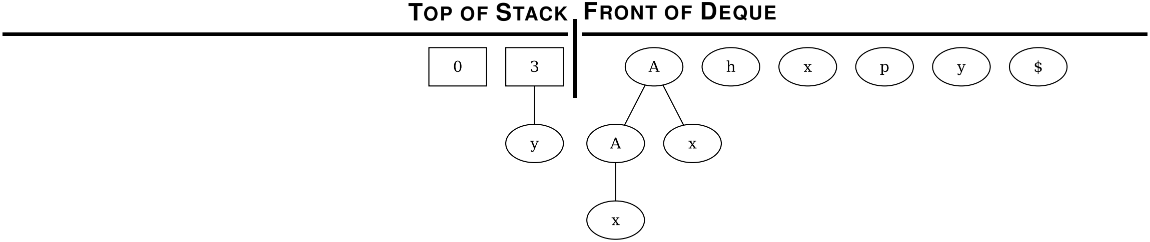
Operation: reduce by rule 3 $A \rightarrow A\ x$



Reduce 3 action: reduce the top-most elements of the stack to be children of **rule 3**'s RHS non-terminal.

Push this tree back onto the front of the deque.

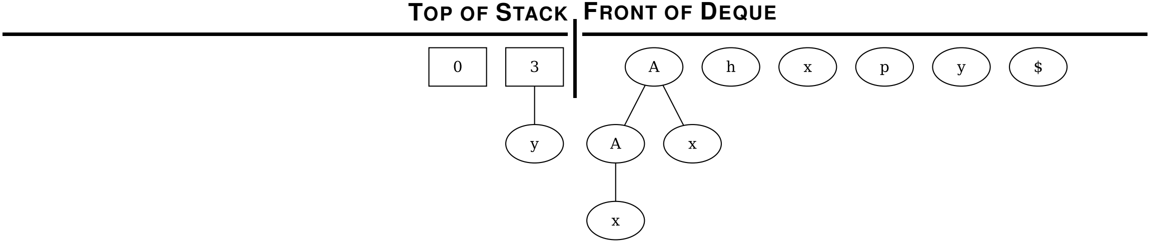
Operation: reduce by rule 3 $A \to A\,x$



The stack's top is state $\boxed{3}$ and the front of the deque is **non-terminal** A,
Using the LR(0) table we look up the **`sh-6`** action

	h	p	x	y	$	A	B	C
⋮								
3			sh-2			sh-6		
4	sh-1		sh-7	sh-3			sh-8	
⋮								

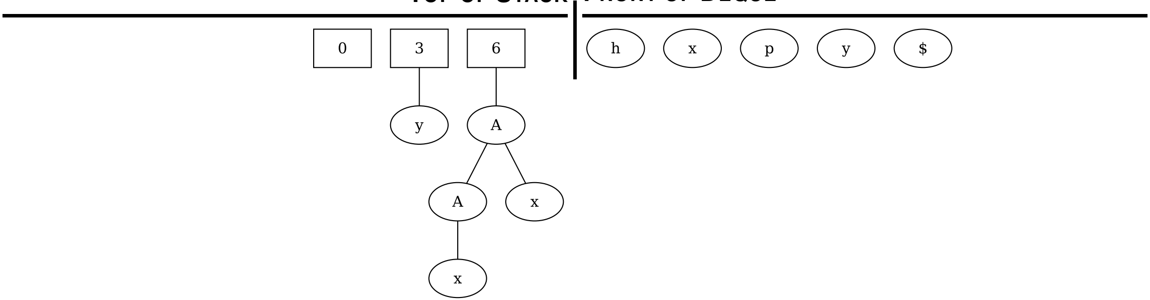
Operation: reduce by rule 3 $A \rightarrow A\, x$



How will **`sh-6`** change the data structures?

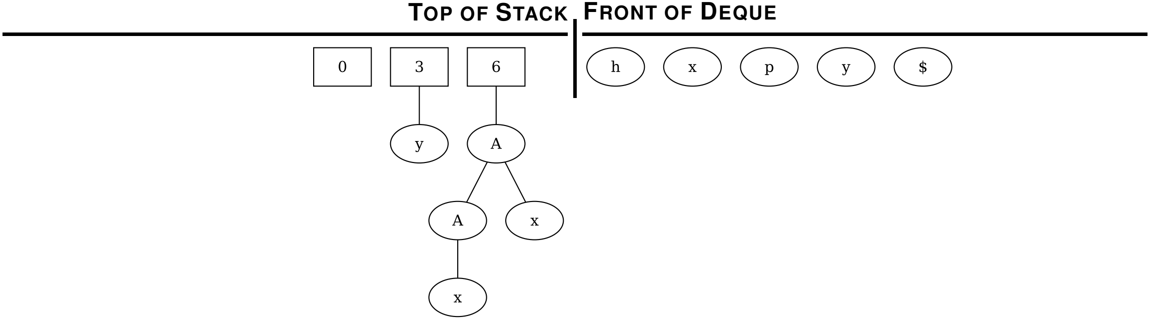
Operation: shift A to stack, goto state 6

TOP OF STACK **FRONT OF DEQUE**



sh−6 action: push state $\boxed{6}$ onto the stack, labeled with the element from the front of the deque.

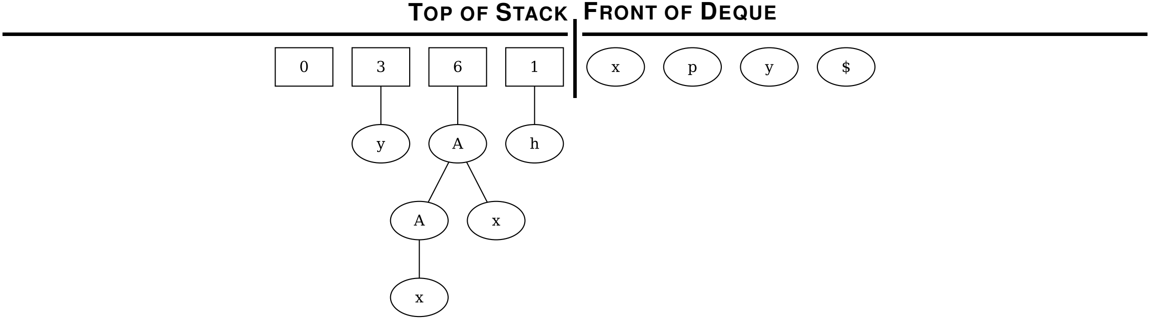
Operation: shift A to stack, goto state 6



The stack's top is state $\boxed{6}$ and the front of the deque is token `h`,
Using the LR(0) table we look up the **`sh-1`** action

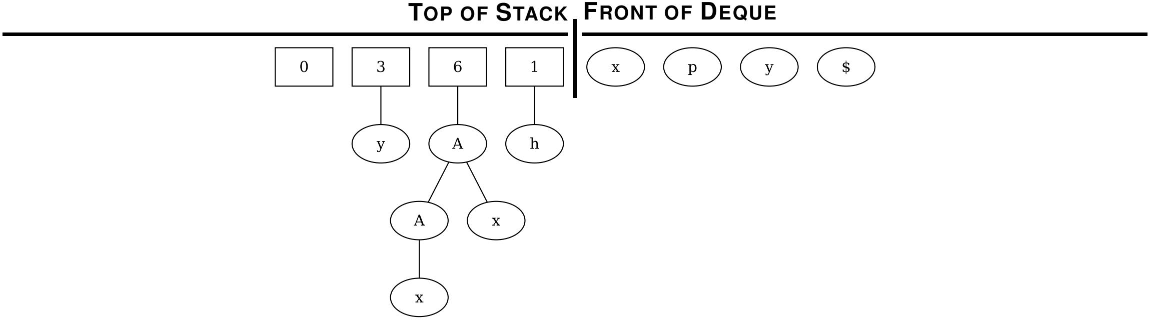
	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
6	sh-1		sh-7	sh-3			sh-12	
7	Reduce 3							
⋮								

Operation: shift h to stack, goto state 1



sh−1 action: push state $\boxed{1}$ onto the stack, labeled with the element from the front of the deque.

Operation: shift h to stack, goto state 1

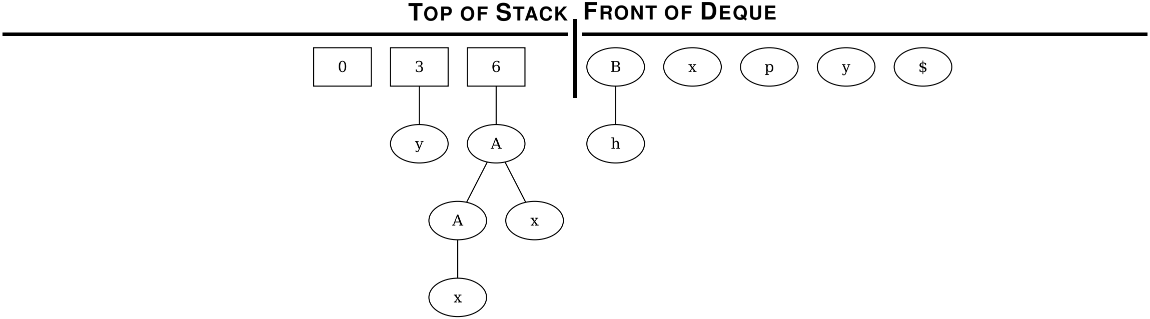


The stack's top is state $\boxed{1}$,

the LR(0) table action is **Reduce 6**

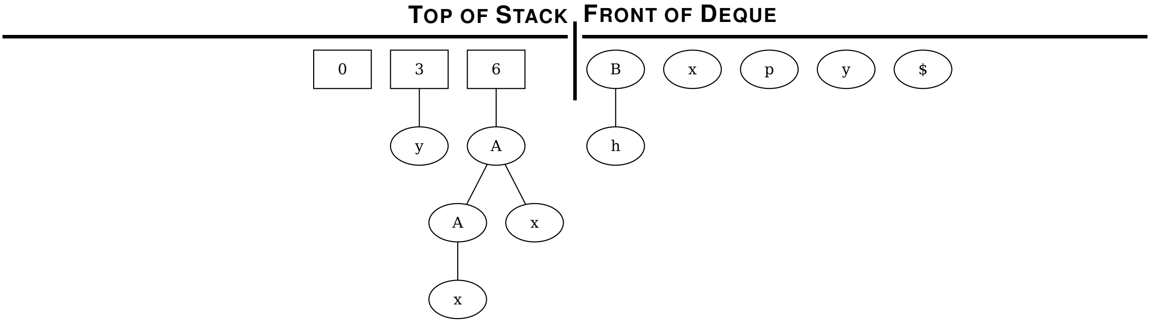
	h	p	x	y	$	A	B	C
0	sh-1		sh-2	sh-3		sh-4	sh-5	
1	Reduce 6							
2	Reduce 4							
⋮								

Operation: reduce by rule 6 $B \rightarrow h$



Be careful not to confuse the enumerated **states on the stack** with **reduction rule numbers** stored in the LR(0) parsing table! Reducing by rule 6 and ending up in state 6 was a coincidence!

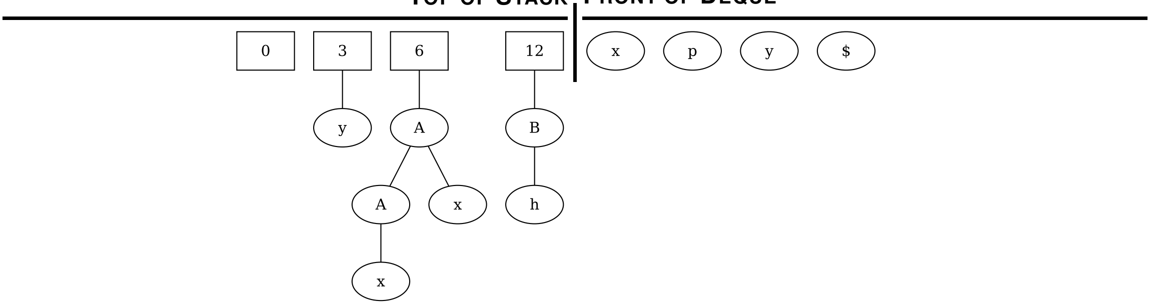
Operation: reduce by rule 6 $B \rightarrow h$



	h	p	x	y	$	A	B	C
⋮								
6	sh-1		sh-7	sh-3			sh-12	
7	Reduce 3							
⋮								

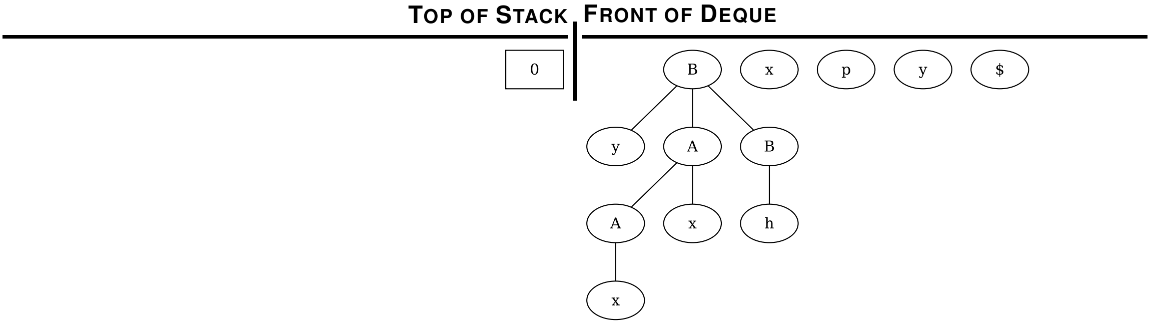
Operation: shift B to stack, goto state 12

TOP OF STACK **FRONT OF DEQUE**



	h	p	x	y	$	A	B	C
⋮								
12	Reduce 5							
13	Reduce 1							
⋮								

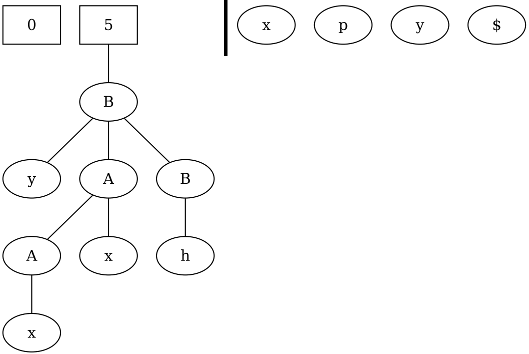
Operation: reduce by rule 5 $B \rightarrow y\,A\,B$



	h	p	x	y	\$	A	B	C
0	sh-1		sh-2	sh-3		sh-4	sh-5	
1	Reduce 6							
2	Reduce 4							
⋮								

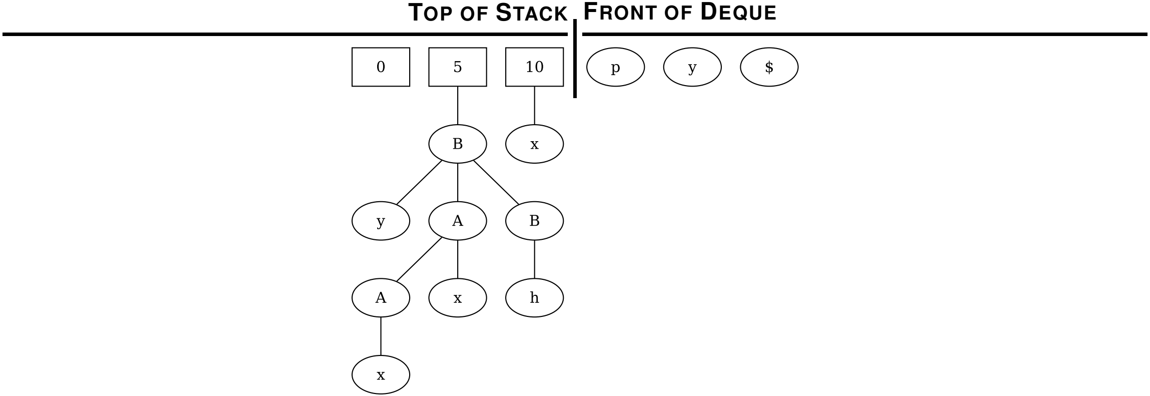
Operation: shift B to stack, goto state 5

TOP OF STACK **FRONT OF DEQUE**



	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
5		sh-9	sh-10					sh-11
6	sh-1		sh-7	sh-3			sh-12	
⋮								

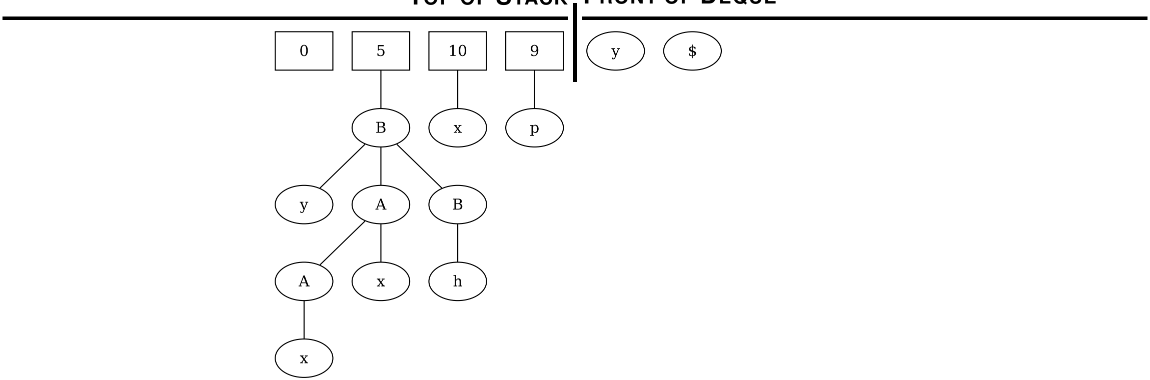
Operation: shift x to stack, goto state 10



	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
10		sh-9	sh-10					sh-14
11					sh-15			
⋮								

Operation: shift p to stack, goto state 9

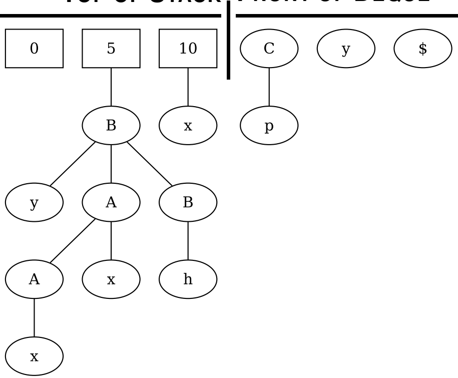
TOP OF STACK **FRONT OF DEQUE**



	h	p	x	y	$\$$	A	B	C
⋮								
9	Reduce 8							
10		sh-9	sh-10					sh-14
⋮								

Operation: reduce by rule 8 $C \rightarrow p$

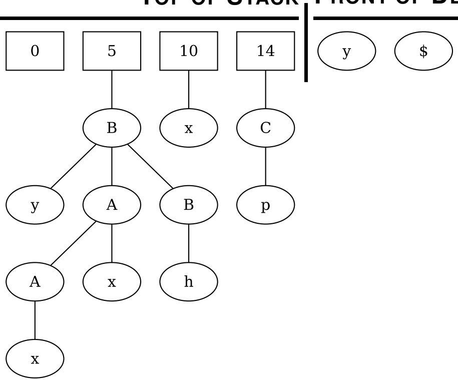
TOP OF STACK **FRONT OF DEQUE**



	h	p	x	y	$\$$	A	B	C
⋮								
10		sh-9	sh-10					sh-14
11					sh-15			
⋮								

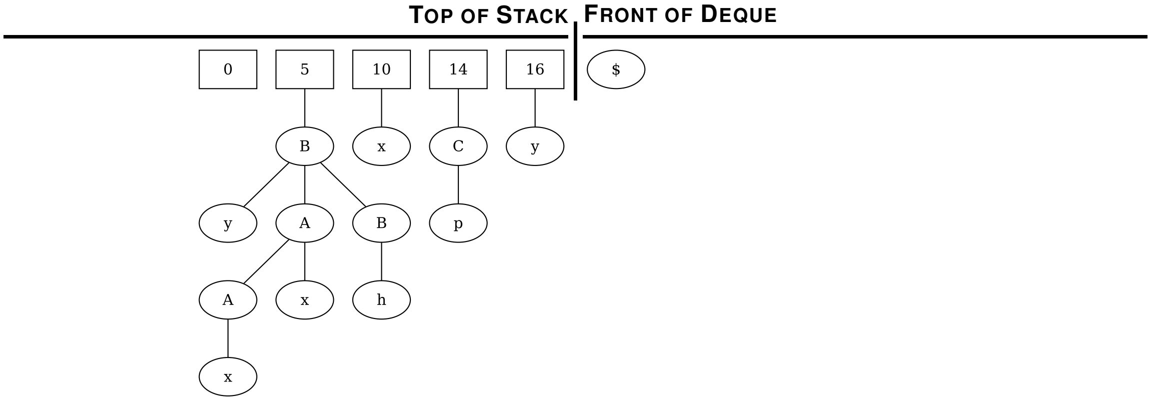
Operation: shift C to stack, goto state 14

TOP OF STACK | **FRONT OF DEQUE**



	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
14				sh-16				
15	Reduce 2							
16	Reduce 7							

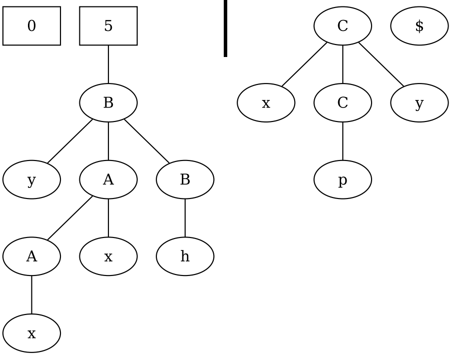
Operation: shift y to stack, goto state 16



	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
14				sh-16				
15	Reduce 2							
16	Reduce 7							

Operation: reduce by rule 7 $C \rightarrow x\,C\,y$

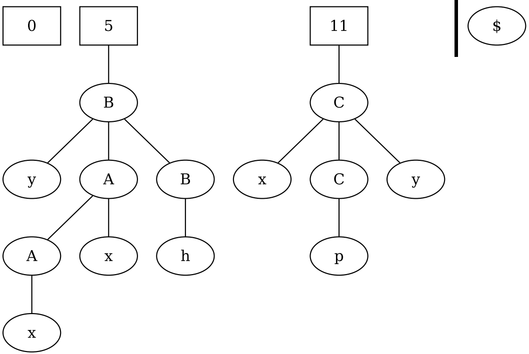
TOP OF STACK | **FRONT OF DEQUE**



	h	p	x	y	$\$$	A	B	C
⋮								
5		sh-9	sh-10					sh-11
6	sh-1		sh-7	sh-3			sh-12	
⋮								

Operation: shift C to stack, goto state 11

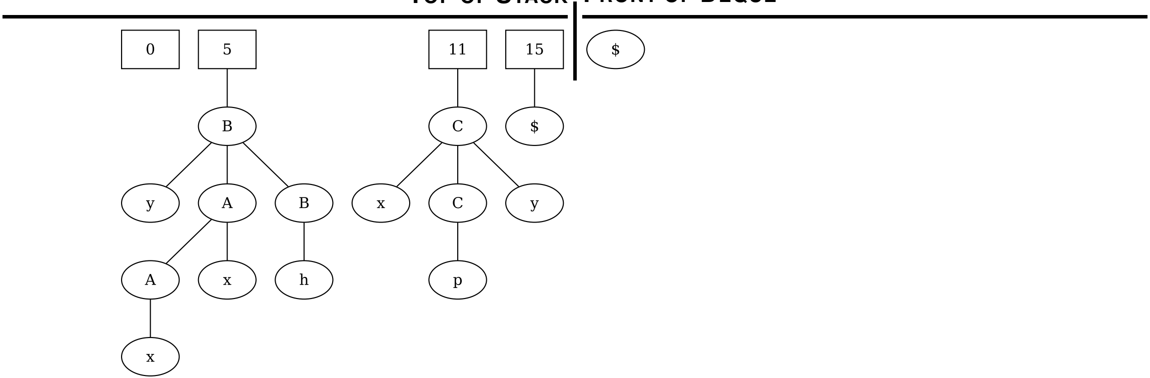
TOP OF STACK **FRONT OF DEQUE**



	h	*p*	*x*	*y*	$	*A*	*B*	*C*
⋮								
11					sh-15			
12	Reduce 5							
⋮								

Operation: shift $ to stack, goto state 15

TOP OF STACK **FRONT OF DEQUE**



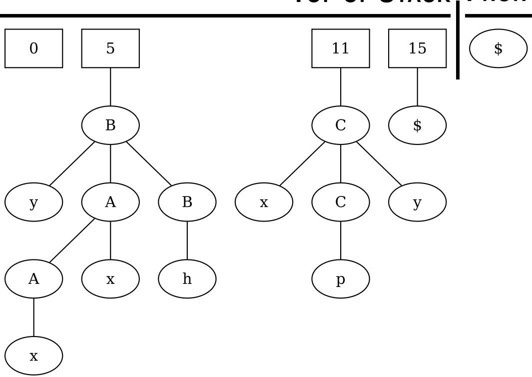
Wait a tick! How can there be TWO end-of-input markers?

This is a common trick in LR parsing, sometimes mentioned in texts as an input queue "back-padded with ∞ \$ markers"

The reason is that it makes the conditional logic of the LR algorithm easier to write and read, and it has no deliterious effects on the outcome. It's just a marker. **:**)

Operation: shift $ to stack, goto state 15

TOP OF STACK | **FRONT OF DEQUE**

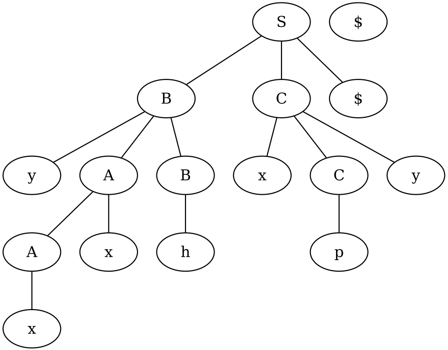


	h	p	x	y	\$	A	B	C
⋮								
14				sh-16				
15	Reduce 2							
16	Reduce 7							

Operation: reduce by rule 2 $S \to B\,C\,\$$

TOP OF STACK | **FRONT OF DEQUE**

0

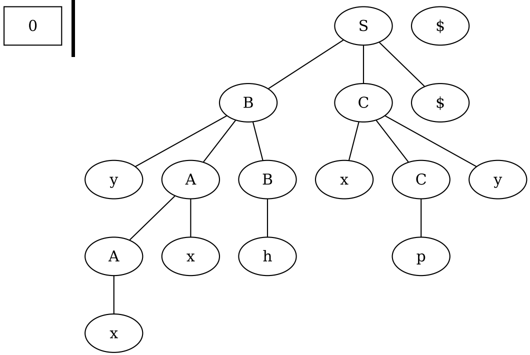


SYNTAX ERROR?

	h	p	x	y	$\$$	A	B	C
0	sh-1		sh-2	sh-3		sh-4	sh-5	
1	Reduce 6							
2	Reduce 4							
⋮								

Operation: reduce by rule 2 $S \to B\,C\,\$$

TOP OF STACK **FRONT OF DEQUE**



No, not in this special case: We are in state $\boxed{0}$ with S at the front of the deque.

The LR(0) table **doesn't have a column for** S!

We must notice that the front of the deque is the **starting goal** of the grammar!

A **raw parse tree of a valid language sentence is at the front of the deque**.

LR Parsing Verifies Input with Rightmost Derivations

Pseudo code for the LR "knitting" (parsing) algorithm `is here` and linked to from the schedule page as well.

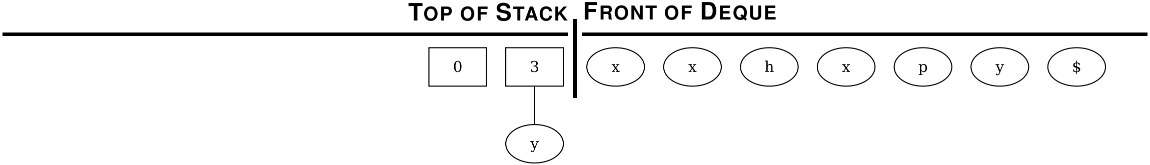
Watch the same input being parsed, but his time we will keep track of the derivational steps being performed.

Operation: begin

TOP OF STACK	FRONT OF DEQUE							
0	y	x	x	h	x	p	y	$

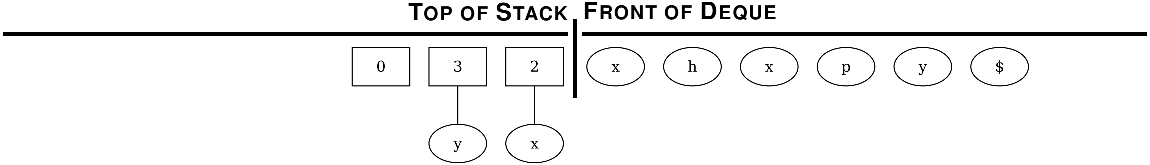
$S \Rightarrow \mid y\, x\, x\, h\, x\, p\, y\, \$$

Operation: shift y to stack, goto state 3



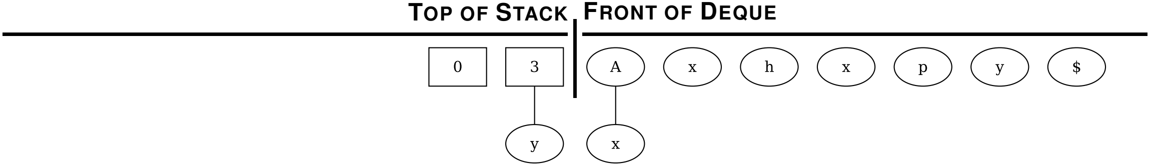
$S \Rightarrow y \,\rule{0.3em}{1em}\, x\, x\, h\, x\, p\, y\, \$$

Operation: shift x to stack, goto state 2



$S \Rightarrow y\, x \,\rule{1ex}{1em}\, x\, h\, x\, p\, y\, \$$

Operation: reduce by rule 4 $A \rightarrow x$

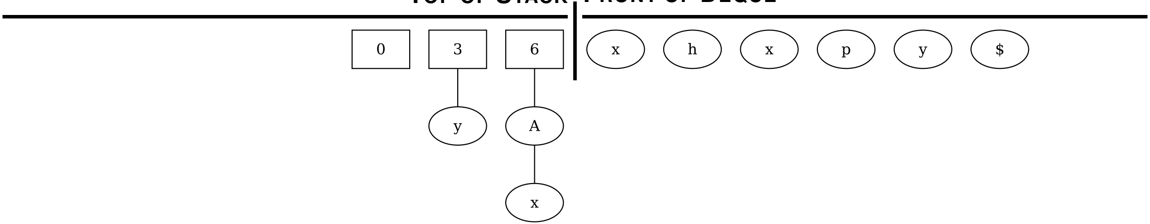


$S \Rightarrow y \,\rule{0.5em}{1em}\, A\, x\, h\, x\, p\, y\, \$$

$S \Rightarrow y\, x\, x\, h\, x\, p\, y\, \$$

Operation: shift A to stack, goto state 6

TOP OF STACK **FRONT OF DEQUE**

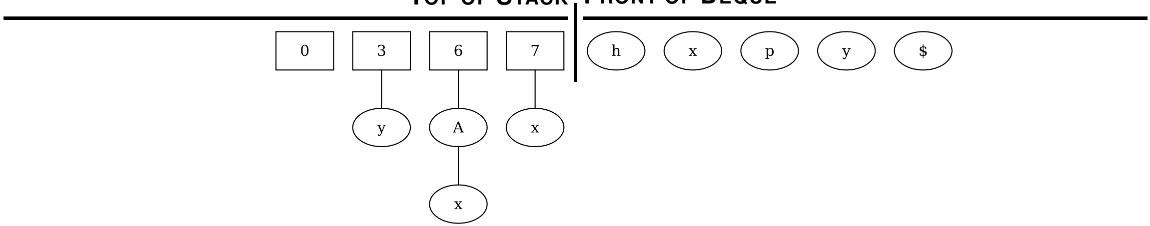


$$S \Rightarrow y\,A \,\big|\, x\,h\,x\,p\,y\,\$$$

$$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$$

Operation: shift x to stack, goto state 7

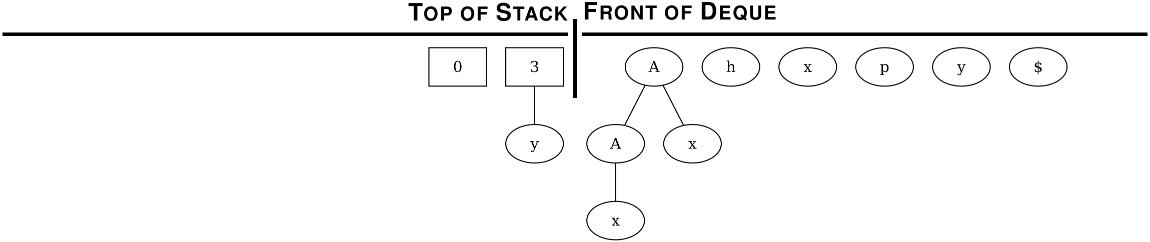
TOP OF STACK **FRONT OF DEQUE**



$$S \Rightarrow y\,A\,x \,\mathbf{|}\, h\,x\,p\,y\,\$$$

$$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$$

Operation: reduce by rule 3 $A \rightarrow A\ x$

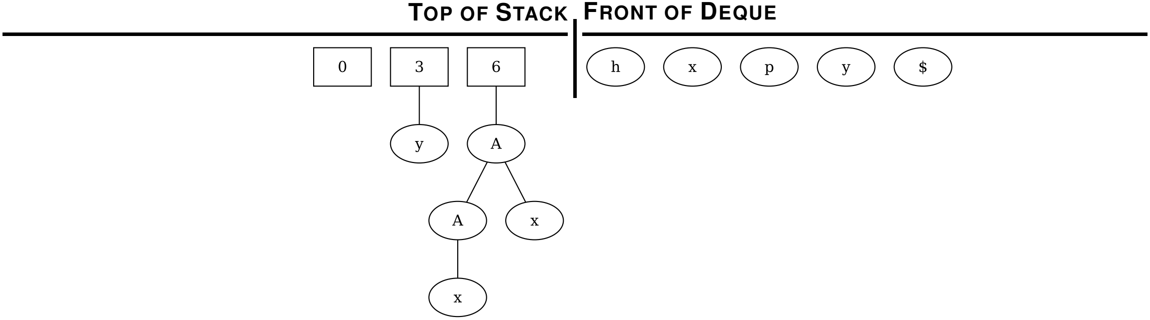


$S \Rightarrow y \,\rule{0.15em}{0.8em}\, A\ h\ x\ p\ y\ \$$

$S \Rightarrow y\ A\ x\ h\ x\ p\ y\ \$$

$S \Rightarrow y\ x\ x\ h\ x\ p\ y\ \$$

Operation: shift A to stack, goto state 6

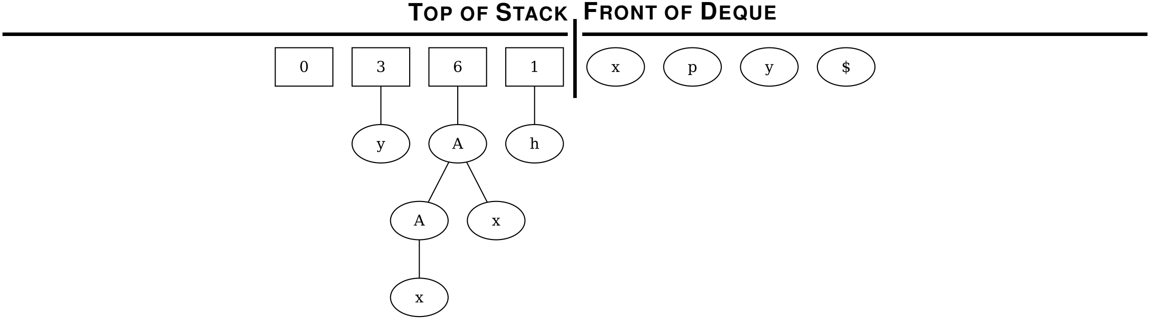


$S \Rightarrow y\,A \,|\, h\,x\,p\,y\,\$$

$S \Rightarrow y\,A\,x\,h\,x\,p\,y\,\$$

$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$

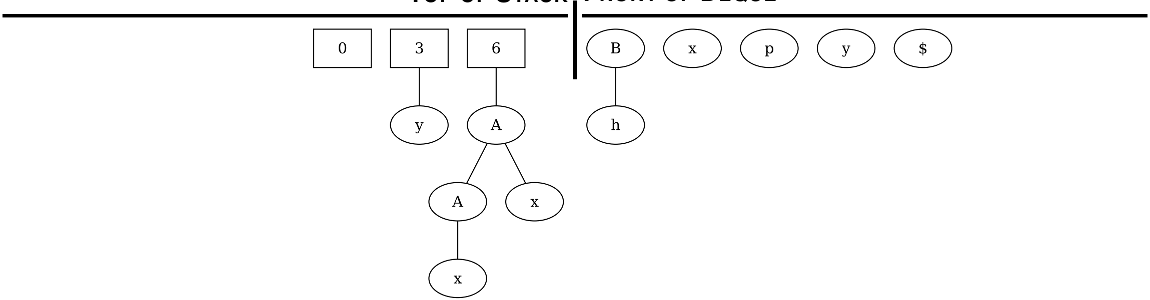
Operation: shift h to stack, goto state 1



$$S \Rightarrow y\,A\,h \,\rule{3pt}{1em}\, x\,p\,y\,\$$$
$$S \Rightarrow y\,A\,x\,h\,x\,p\,y\,\$$$
$$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$$

Operation: reduce by rule 6 $B \rightarrow h$

TOP OF STACK **FRONT OF DEQUE**



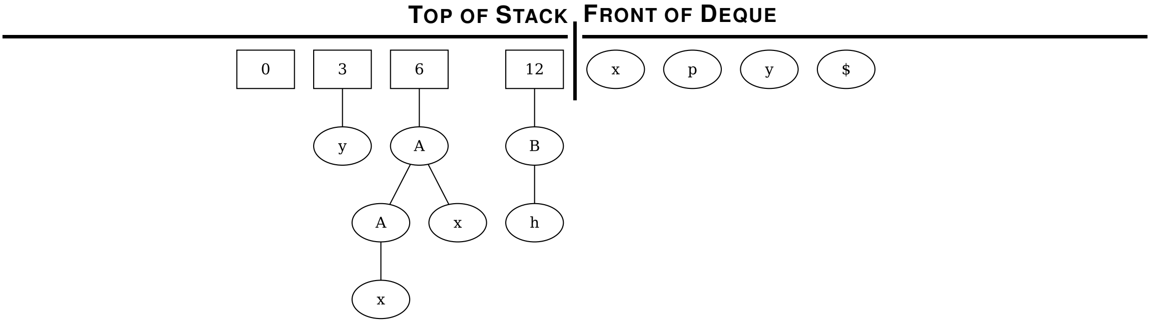
$S \Rightarrow y\, A \mid B\, x\, p\, y\, \$$

$S \Rightarrow y\, A\, h\, x\, p\, y\, \$$

$S \Rightarrow y\, A\, x\, h\, x\, p\, y\, \$$

$S \Rightarrow y\, x\, x\, h\, x\, p\, y\, \$$

Operation: shift B to stack, goto state 12



$S \Rightarrow y\,A\,B\,\rule{0.4em}{1em}\,x\,p\,y\,\$$

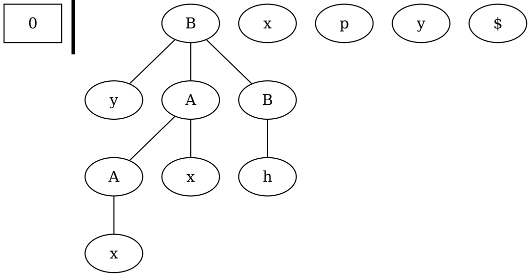
$S \Rightarrow y\,A\,h\,x\,p\,y\,\$$

$S \Rightarrow y\,A\,x\,h\,x\,p\,y\,\$$

$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$

Operation: reduce by rule 5 $B \rightarrow y\,A\,B$

TOP OF STACK | **FRONT OF DEQUE**



$S \Rightarrow \rule{0.3em}{1em}\, B\ x\ p\ y\ \$$

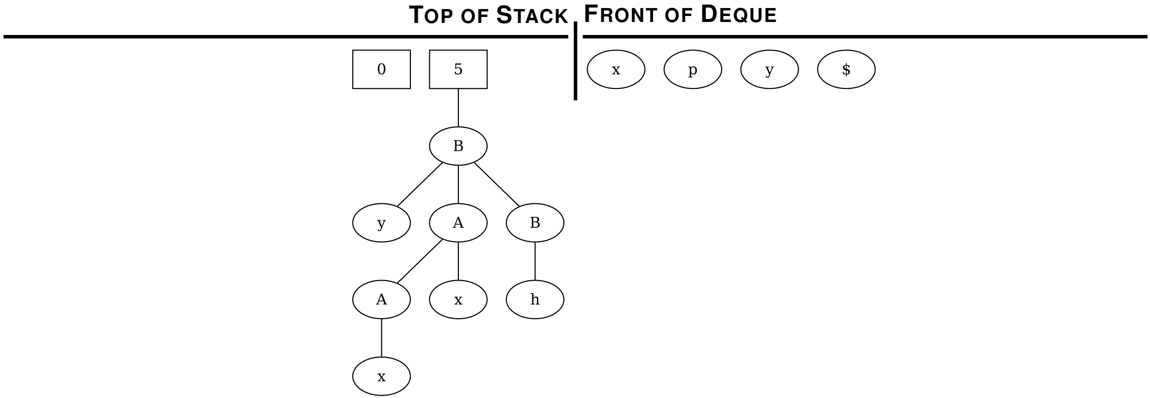
$S \Rightarrow y\ A\ B\ x\ p\ y\ \$$

$S \Rightarrow y\ A\ h\ x\ p\ y\ \$$

$S \Rightarrow y\ A\ x\ h\ x\ p\ y\ \$$

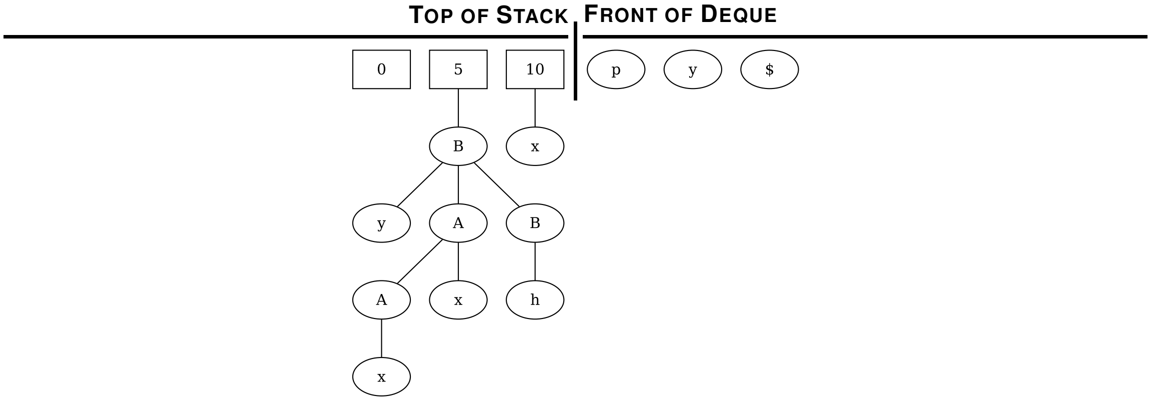
$S \Rightarrow y\ x\ x\ h\ x\ p\ y\ \$$

Operation: shift B to stack, goto state 5



$$S \Rightarrow B \mid x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ B\ x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ h\ x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ x\ h\ x\ p\ y\ \$$$
$$S \Rightarrow y\ x\ x\ h\ x\ p\ y\ \$$$

Operation: shift x to stack, goto state 10



$$S \Rightarrow B\ x \mid p\ y\ \$$$

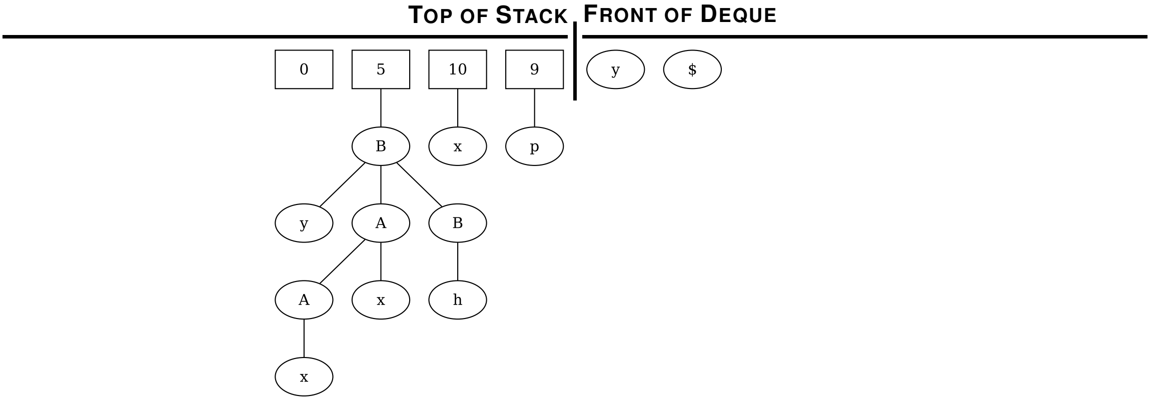
$$S \Rightarrow y\ A\ B\ x\ p\ y\ \$$$

$$S \Rightarrow y\ A\ h\ x\ p\ y\ \$$$

$$S \Rightarrow y\ A\ x\ h\ x\ p\ y\ \$$$

$$S \Rightarrow y\ x\ x\ h\ x\ p\ y\ \$$$

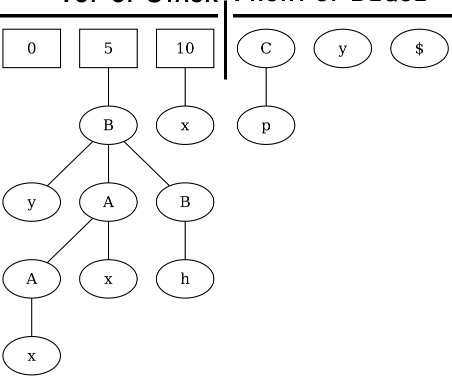
Operation: shift p to stack, goto state 9



$$S \Rightarrow B\ x\ p \mid y\ \$$$
$$S \Rightarrow y\ A\ B\ x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ h\ x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ x\ h\ x\ p\ y\ \$$$
$$S \Rightarrow y\ x\ x\ h\ x\ p\ y\ \$$$

Operation: reduce by rule 8 $C \rightarrow p$

TOP OF STACK | **FRONT OF DEQUE**



$S \Rightarrow B\,x\,|\,C\,y\,\$$

$S \Rightarrow B\,x\,p\,y\,\$$

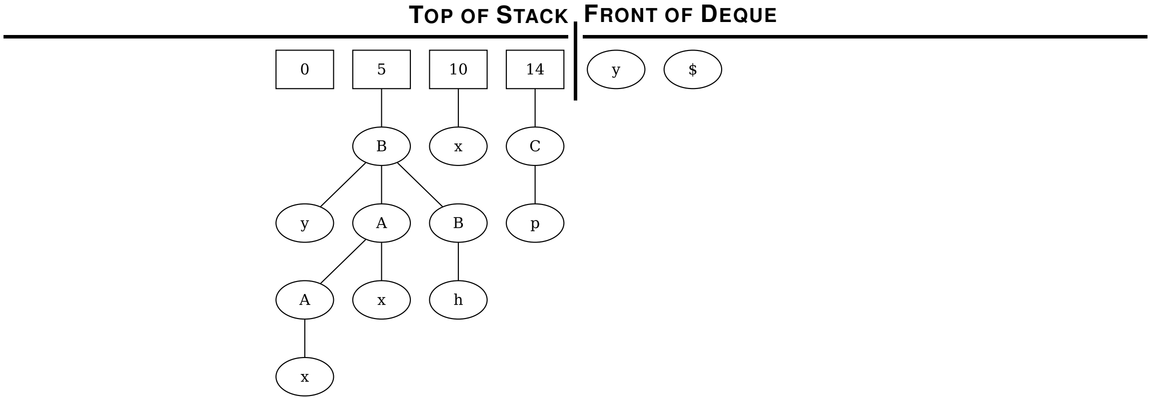
$S \Rightarrow y\,A\,B\,x\,p\,y\,\$$

$S \Rightarrow y\,A\,h\,x\,p\,y\,\$$

$S \Rightarrow y\,A\,x\,h\,x\,p\,y\,\$$

$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$

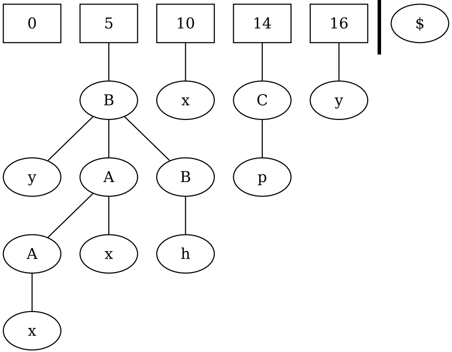
Operation: shift C to stack, goto state 14



$$S \Rightarrow B\ x\ C\ \vert\ y\ \$$$
$$S \Rightarrow B\ x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ B\ x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ h\ x\ p\ y\ \$$$
$$S \Rightarrow y\ A\ x\ h\ x\ p\ y\ \$$$
$$S \Rightarrow y\ x\ x\ h\ x\ p\ y\ \$$$

Operation: shift y to stack, goto state 16

TOP OF STACK | **FRONT OF DEQUE**



$$S \Rightarrow B\,x\,C\,y \,\rule{0.3em}{1em}\, \$$$

$$S \Rightarrow B\,x\,p\,y\,\$$$

$$S \Rightarrow y\,A\,B\,x\,p\,y\,\$$$

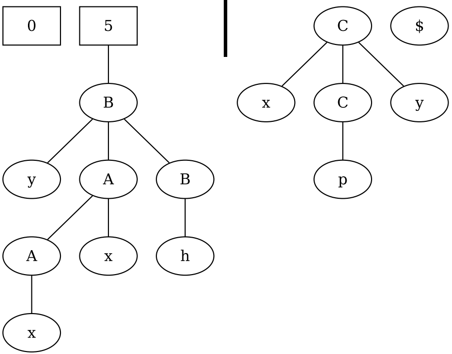
$$S \Rightarrow y\,A\,h\,x\,p\,y\,\$$$

$$S \Rightarrow y\,A\,x\,h\,x\,p\,y\,\$$$

$$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$$

Operation: reduce by rule 7 $C \rightarrow x\,C\,y$

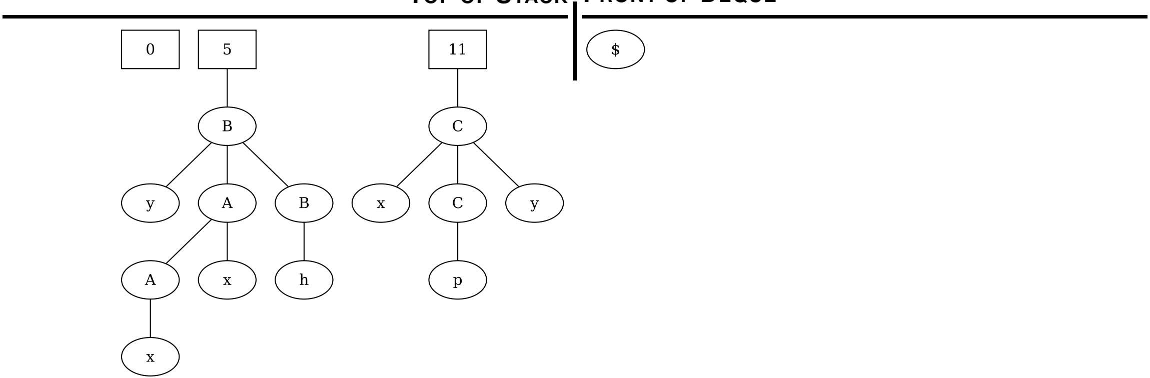
TOP OF STACK | **FRONT OF DEQUE**



$$S \Rightarrow B \,|\, C\,\$$$
$$S \Rightarrow B\,x\,C\,y\,\$$$
$$S \Rightarrow B\,x\,p\,y\,\$$$
$$S \Rightarrow y\,A\,B\,x\,p\,y\,\$$$
$$S \Rightarrow y\,A\,h\,x\,p\,y\,\$$$
$$S \Rightarrow y\,A\,x\,h\,x\,p\,y\,\$$$
$$S \Rightarrow y\,x\,x\,h\,x\,p\,y\,\$$$

Operation: shift C to stack, goto state 11

TOP OF STACK | **FRONT OF DEQUE**



$$S \Rightarrow B\,C \mid \$$$

$$S \Rightarrow B\,x\,C\,y\,\$$$

$$S \Rightarrow B\,x\,p\,y\,\$$$

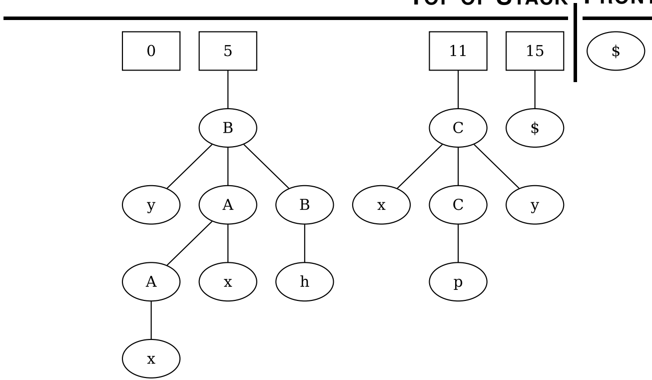
$$S \Rightarrow y\,A\,B\,x\,p\,y\,\$$$

$$S \Rightarrow y\,A\,h\,x\,p\,y\,\$$$

$$\vdots$$

Operation: shift \$ to stack, goto state 15

TOP OF STACK | **FRONT OF DEQUE**



$$S \Rightarrow B\,C\,\$\,\blacksquare$$

$$S \Rightarrow B\,x\,C\,y\,\$$$

$$S \Rightarrow B\,x\,p\,y\,\$$$

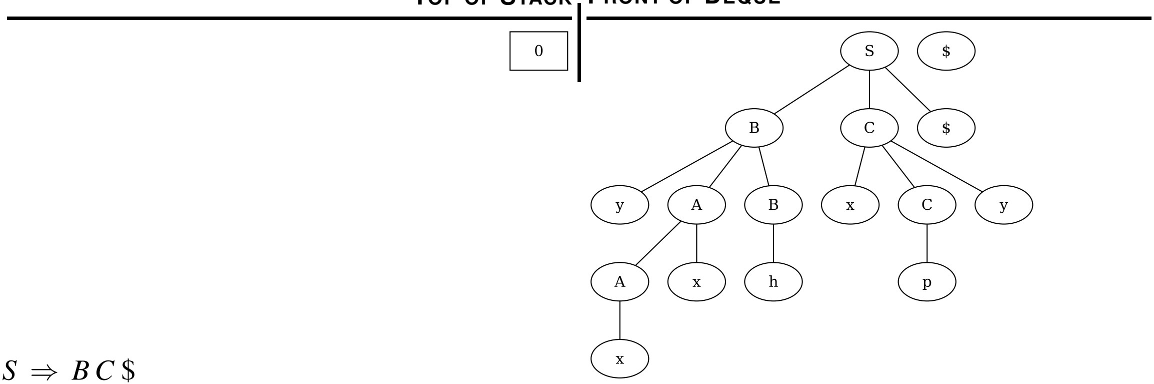
$$S \Rightarrow y\,A\,B\,x\,p\,y\,\$$$

$$S \Rightarrow y\,A\,h\,x\,p\,y\,\$$$

$$\vdots$$

Operation: reduce by rule 2 $S \to B\,C\,\$$

TOP OF STACK | **FRONT OF DEQUE**



$S \Rightarrow B\,C\,\$$

$S \Rightarrow_{rm} B\,x\,C\,y\,\$$

$S \Rightarrow_{rm} B\,x\,p\,y\,\$$

$S \Rightarrow_{rm} y\,A\,B\,x\,p\,y\,\$$

$S \Rightarrow_{rm} y\,A\,h\,x\,p\,y\,\$$

$S \Rightarrow_{rm} y\,A\,x\,h\,x\,p\,y\,\$$

$S \Rightarrow_{rm} y\,x\,x\,h\,x\,p\,y\,\$$

From the initial goal rule downward, the rightmost non-terminal was always reduced. **So this was a rightmost derivation.** The "parse time ordering" of these operations are left to right **but from the bottom of the derivation up** — from the **top down** this is a rightmost parse!

LR, the "Canonical" Way to Parse

LR parsing[1] is often referred to as "canonical" parsing. Why?

[1] Technically, it's LR(1) that is considered the canonical form.

LR, the "Canonical" Way to Parse

LR parsing[1] is often referred to as "canonical" parsing. Why?

canonical kə-nŏn'ĭ-kəl
adj. Of, relating to, or required by canon law. adj. Of or appearing in the biblical canon. adj. ***Conforming to orthodox*** *or well-established rules or patterns, as of procedure.*

orthodox 'ór-thə-däks
1a: conforming to ***established doctrine*** *especially in religion orthodox principles the orthodox interpretation 1b: conventional*

IOW: this is "the way to parse."

LR(k) ("shift-reduce parsing") was shown by Knuth (1965) to be capable of parsing **any deterministic** context free grammar. Knuth's result was more academic than practical at the time because it required **huge data structures in memory** to form the parsing table.[2]

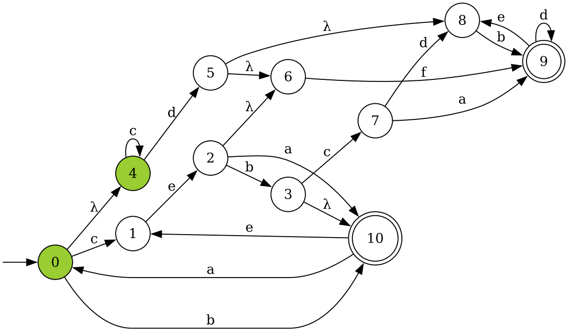
Subsequent research by others produced more memory-practical algorithms such as SLR (what we'll focus on in this course) and LALR.

[1] Technically, it's LR(1) that is considered the canonical form.
[2] Historical tidbit: you will notice languages developed before this result use `endif` markers — why? They were using LL (recursive descent) parsing and needed to resolve the "dangling brackets problem" of `if-then-else` structures.

LR, the "Canonical" Way to Parse

Deterministic Context Free Grammar?

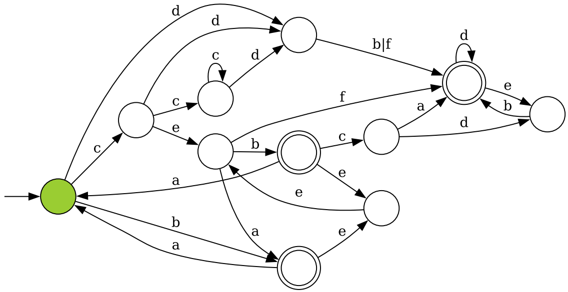
Similar to the difference between **NFAs and DFAs**: to match a string with an NFA you'll have to remember multiple states at one time because NFAs have λ-edges and permit multiple same-character transitions away from a node (state).



LR, the “Canonical” Way to Parse

Deterministic Context Free Grammar?

Deterministic FAs don’t have λ-edges and permit only one transition per character from a state. The “matching state” of DFAs can be expressed in a simple table and can be stored as a single value in an algorithm.



State	a	b	c	d	e	f
0		8	1	2		
1			4	2	5	
2		9				9
3		9				
4			4	2		
5	8	10				9
6	9			3		
7					5	
+ 8	0				7	
+ 9				9	3	
+ 10	0		6		7	

LR, the "Canonical" Way to Parse

Deterministic Context Free Grammar?

Analogously, **deterministic context free grammars** can be parsed by remembering only one state throughout the parsing algorithm.

Our stack in the shift-reduce algorithm remembers a **history of states** we will return to, but the algorithm itself is in only one state at a time.

It is the state at the **top of the stack**.

	h	p	x	y	\$	A	B	C
0	sh-1		sh-2	sh-3		sh-4	sh-5	
1	Reduce 6							
2	Reduce 4							
3			sh-2			sh-6		
4	sh-1		sh-7	sh-3			sh-8	
5		sh-9	sh-10					sh-11
6	sh-1		sh-7	sh-3			sh-12	
7	Reduce 3							
8					sh-13			
9	Reduce 8							
10		sh-9	sh-10					sh-14
11					sh-15			
12	Reduce 5							
13	Reduce 1							
14				sh-16				
15	Reduce 2							
16	Reduce 7							

LR, the "Canonical" Way to Parse

Deterministic Context Free Grammar?

Notice how each cell of the LR parsing table has only one action to be performed?

`sh-X` shift input to stack under parse state X

`Reduce-Y` "reduce" the top elements of the stack with production rule Y, push the resulting tree back onto the input deque

None of the **cells** contain entries like `sh-2` AND `sh-8` or `sh-6` AND `Reduce 3`.

	h	p	x	y	\$	A	B	C
0	sh-1		sh-2	sh-3		sh-4	sh-5	
1	Reduce 6							
2	Reduce 4							
3			sh-2			sh-6		
4	sh-1		sh-7	sh-3			sh-8	
5		sh-9	sh-10					sh-11
6	sh-1		sh-7	sh-3			sh-12	
7	Reduce 3							
8					sh-13			
9	Reduce 8							
10		sh-9	sh-10					sh-14
11					sh-15			
12	Reduce 5							
13	Reduce 1							
14				sh-16				
15	Reduce 2							
16	Reduce 7							

LR, the "Canonical" Way to Parse

This grammar highlights how LR parsing defers the choice of production rules until all of a RHS is satisfied:

#	Rules
1	$S \rightarrow x\,A\,\$$
2	$A \rightarrow x\,A$
3	$A \rightarrow x\,B$
4	$A \rightarrow x\,C$
5	$B \rightarrow y\,y\,g$
6	$C \rightarrow y\,y\,k$

Not surprisingly, we'll see the algorithm consume **all of the input** before encountering a g or k and "deciding" which of the A production rules to use.

LR parsing **runtime** memory complexity is on the order of source input length.

Unlike LL parsing, shift-reduce (canonical, LR) parsing **defers production rule decisions** until all of the RHS has been seen.

Operation: begin

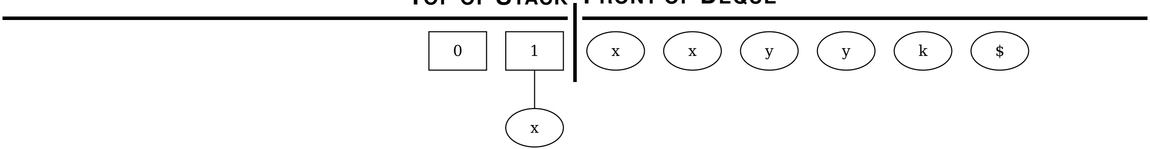
TOP OF STACK | **FRONT OF DEQUE**

| 0 | x | x | x | y | y | k | $ |

	g	*k*	*x*	*y*	$	*A*	*B*	*C*
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

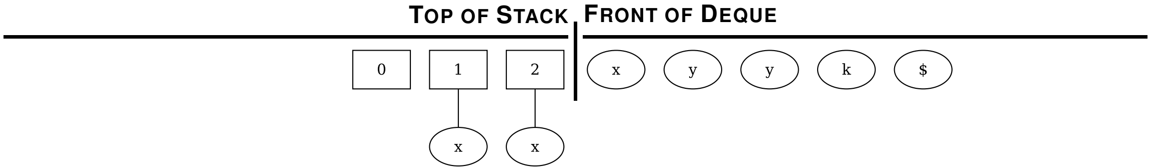
Operation: shift x to stack, goto state 1

TOP OF STACK **FRONT OF DEQUE**



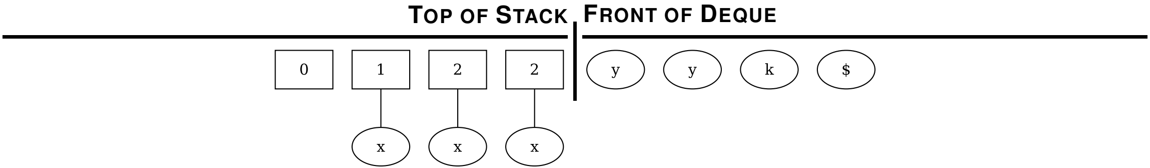
	g	*k*	*x*	*y*	$	*A*	*B*	*C*
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: shift x to stack, goto state 2



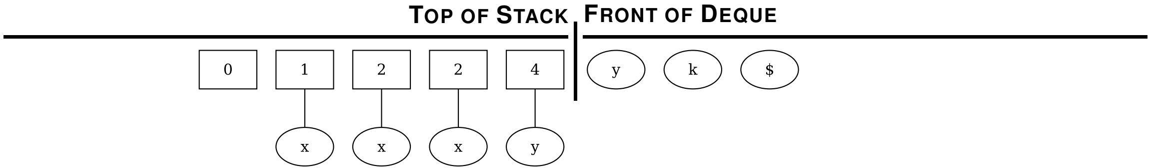
	g	*k*	*x*	*y*	$	*A*	*B*	*C*
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: shift x to stack, goto state 2



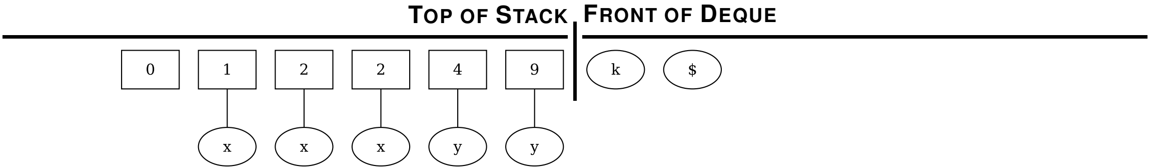
	g	k	x	y	$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: shift y to stack, goto state 4



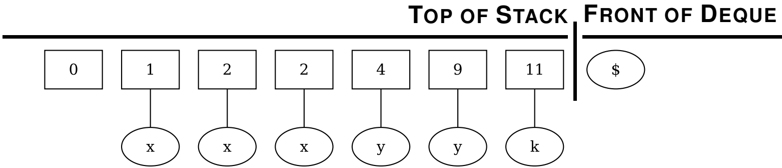
	g	k	x	y	$\$$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: shift y to stack, goto state 9



	g	k	x	y	\$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

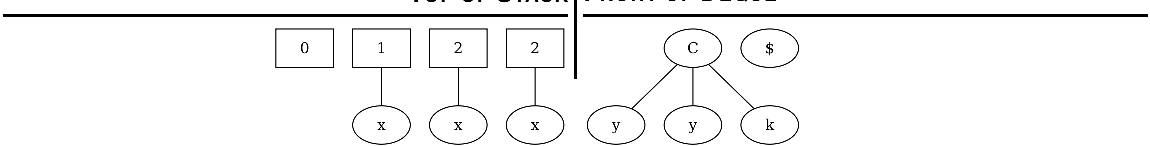
Operation: shift k to stack, goto state 11



	g	k	x	y	$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: reduce by rule 6 $C \rightarrow y\,y\,k$

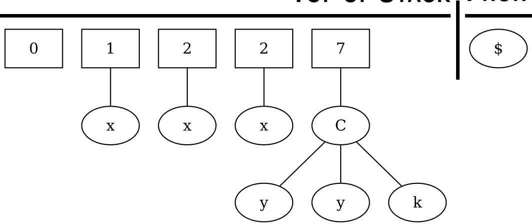
TOP OF STACK | **FRONT OF DEQUE**



	g	k	x	y	$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: shift C to stack, goto state 7

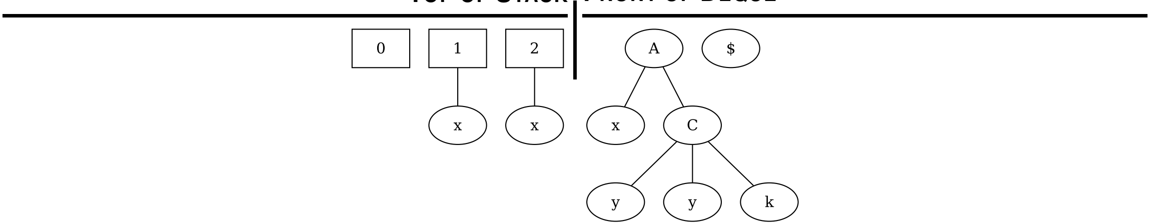
TOP OF STACK **FRONT OF DEQUE**



	g	k	x	y	$\$$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: reduce by rule 4 $A \rightarrow x\,C$

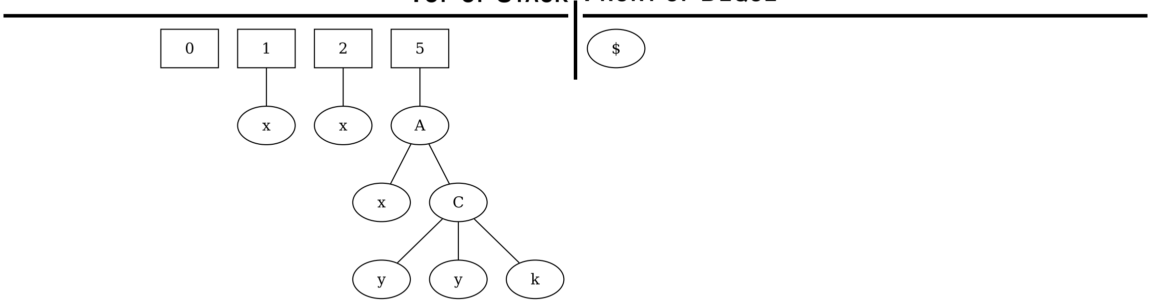
TOP OF STACK | **FRONT OF DEQUE**



	g	k	x	y	\$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: shift A to stack, goto state 5

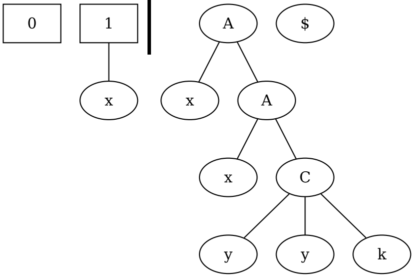
TOP OF STACK **FRONT OF DEQUE**



	g	k	x	y	$	A	B	C
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

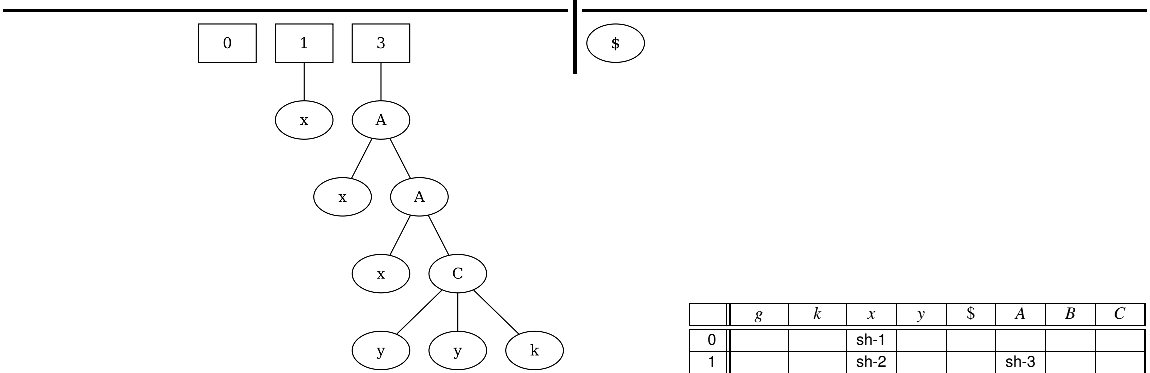
Operation: reduce by rule 2 $A \rightarrow x\,A$

TOP OF STACK **FRONT OF DEQUE**



Operation: shift A to stack, goto state 3

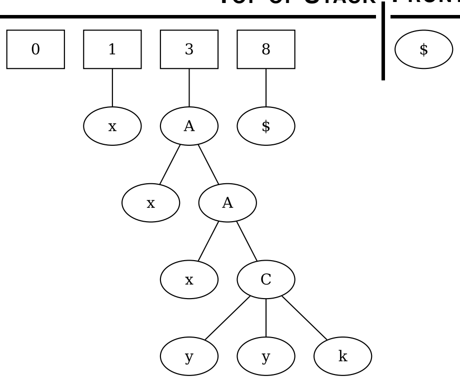
TOP OF STACK **FRONT OF DEQUE**



	g	*k*	*x*	*y*	$	*A*	*B*	*C*
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: shift $ to stack, goto state 8

TOP OF STACK | **FRONT OF DEQUE**



	g	*k*	*x*	*y*	$	*A*	*B*	*C*
0			sh-1					
1			sh-2			sh-3		
2			sh-2	sh-4		sh-5	sh-6	sh-7
3					sh-8			
4				sh-9				
5	Reduce 2							
6	Reduce 3							
7	Reduce 4							
8	Reduce 1							
9	sh-10	sh-11						
10	Reduce 5							
11	Reduce 6							

Operation: reduce by rule 1 $S \rightarrow x\,A\,\$$

TOP OF STACK | **FRONT OF DEQUE**