CPSC 221
Basic Algorithms and Data Structures

ADTs, Stacks, and Queues

Textbook References:
Koffman: 4.5-4.7, 5, 6.1-6.3, 6.5

Hassan Khosravi
January – April 2015

Learning goals

- Differentiate an abstraction from an implementation.
- Define and give examples of problems that can be solved using the abstract data types stacks and queues.
- Compare and contrast the implementations of these abstract data types using linked lists and circular arrays in C++.
- Manipulate data in stacks and queues(irrespective of any implementation).

What is an Abstract Data Type?

- Abstract Data Type (ADT) – a mathematical description of an object and the set of operations on the object.
  - A description of how a data structure works (could be implemented by different actual data structures).
- Example: Dictionary ADT
  - Stores pairs of strings: (word, definition)
  - Operations:
    - Insert(word, definition)
    - Delete(word)
    - Find(word)

Implemented by a data structure

Why so many data structures?

Ideal data structure:

fast, elegant, memory efficient

Trade-offs

- time vs. space
- performance vs. elegance
- generality vs. simplicity
- one operation's performance vs. another's
- serial performance vs. parallel performance

"Dictionary" or "Map" ADT

- list
- binary search tree
- AVL tree
- Splay tree
- B+ tree
- Red-Black tree
- hash table
- concurrent hash table
- …

Code Implementation

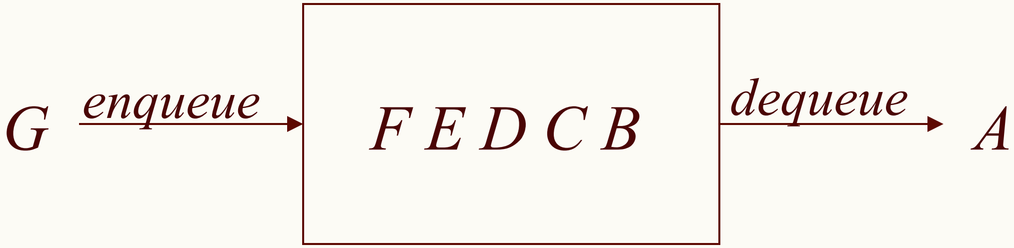
- Theoretically
  - abstract base class describes ADT
  - inherited implementations implement data structures
  - can change data structures transparently (to client code)
- Practice
  - different implementations sometimes suggest different interfaces (generality *vs*. simplicity)
  - performance of a data structure may influence form of client code (time *vs*. space, one operation *vs*. another)

ADT Presentation Algorithm

- Present an ADT
- Motivate with some applications
- Repeat until browned entirely through
  - develop a data structure for the ADT
  - analyze its properties
    - efficiency
    - correctness
    - limitations
    - ease of programming
- Contrast data structure's strengths and weaknesses
  - understand when to use each one

Queue ADT

- Queue operations
  - create
  - destroy
  - enqueue
  - dequeue
  - is_empty



- Queue property:

  if x is enqueued before y is enqueued,
  then x will be dequeued before y is dequeued.

FIFO: First In First Out

Applications of the Q

- Hold jobs for a printer
- Store packets on network routers
- Hold memory “freelists”
- Make waitlists fair
- Breadth first search

Abstract Q Example

enqueue R

enqueue O

dequeue

enqueue T

enqueue A

enqueue T

dequeue

dequeue

enqueue E

dequeue

In order, what letters are dequeued?

a. OATE
b. ROTA
c. OTAE
d. None of these, but it **can** be determined from just the ADT.
e. None of these, and it **cannot** be determined from just the ADT.

Abstract Q Example

enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue

In order, what letters are dequeued?

a. OATE
b. ROTA
c. OTAE
d. None of these, but it **can** be determined from just the ADT.
e. None of these, and it **cannot** be determined from just the ADT.

Array Representation of Queues

- Queues can be easily represented using linear arrays.
- Every queue has front and back variables that point to the position from where deletions and insertions can be done, respectively. Consider the queue shown in figure

12	9	7	18	14	36				
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

front = 0
back = 6

- If we want to add one more value in the list say with value 45, then back would be incremented by 1 and the value would be stored at the position pointed by back.

12	9	7	18	14	36	45			
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

front = 0
back= 7

Array Representation of Queues

- Now, if we want to delete an element from the queue, then the value of front will be incremented. Deletions are done from only this end of the queue

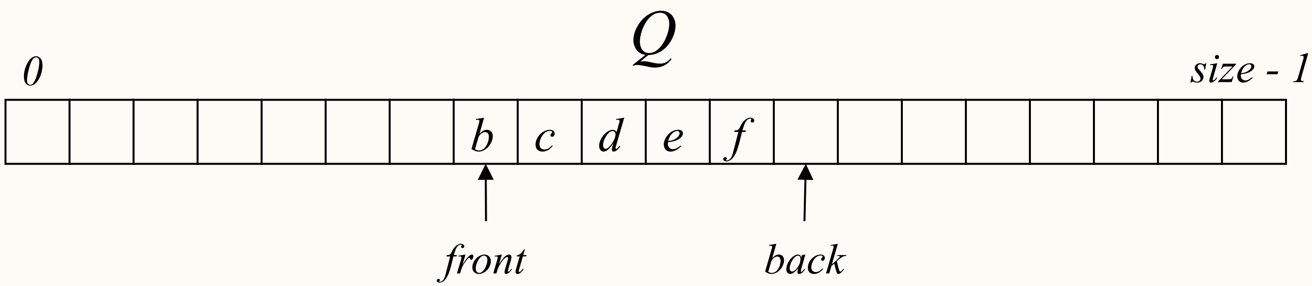
	9	7	18	14	36	45			
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

front = 1
back = 7

- What is a problem with this implementation?

		7	18	14	36	45	21	99	72
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Circular Array Q Data Structure



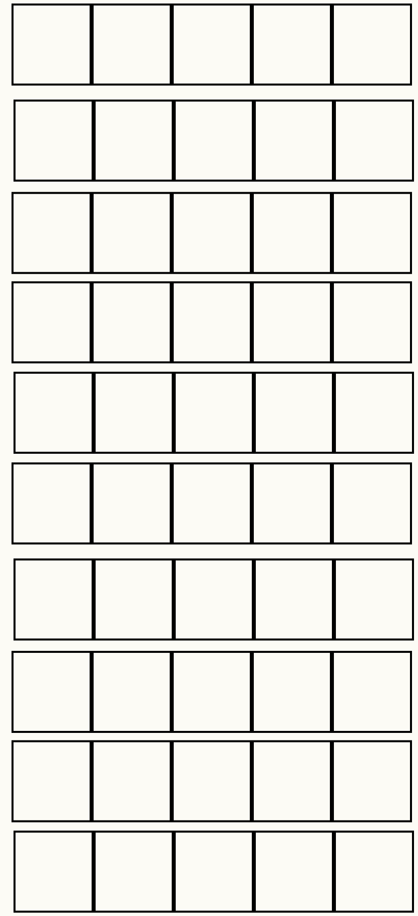

```
void enqueue(Object x) {
    Q[back] = x;
    back = (back + 1) % size;
}
Object dequeue() {
    x = Q[front];
    front = (front + 1) % size;
    return x;
}
```

```
bool is_empty() {
    return (front == back);
}


bool is_full() {
    return front ==
        (back + 1) % size;
}
```

Circular Array Q Example 1

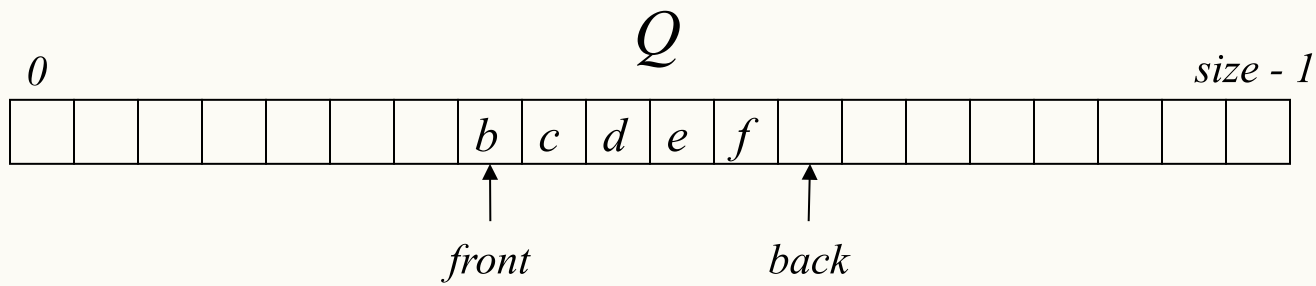
enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue



Circular Array Q Example 1

enqueue R	*R*				
enqueue O	*R*	*O*			
dequeue	~~*R*~~	*O*			
enqueue T	~~*R*~~	*O*	*T*		
enqueue A	~~*R*~~	*O*	*T*	*A*	
enqueue T	~~*R*~~	*O*	*T*	*A*	*T*
dequeue	~~*R*~~	~~*O*~~	*T*	*A*	*T*
dequeue	~~*R*~~	~~*O*~~	~~*T*~~	*A*	*T*
enqueue E	*E*	~~*O*~~	~~*T*~~	*A*	*T*
dequeue	*E*	~~*O*~~	~~*T*~~	~~*A*~~	*T*

Circular Array Q Data Structure




```
void enqueue(Object x) {
    Q[back] = x;
    back = (back + 1) % size;
}
Object dequeue() {
    x = Q[front];
    front = (front + 1) % size;
    return x;
}
```

```
bool is_empty() {
    return (front == back);
}

bool is_full() {
    return front ==
           (back + 1) % size;
}
```

What is wrong with this code?

Circular Array Q Example 2

enqueue R	*R*				
enqueue O	*R*	*O*			
enqueue T	*R*	*O*	*T*		
enqueue A	*R*	*O*	*T*	*A*	
enqueue T	*R*	*O*	*T*	*A*	*T*
enqueue E	*E*	*O*	*T*	*A*	*T*

- Before inserting
  - Check is_full()
- Before removing
  - Check is_empty()

Circular Array Q Example 3

enqueue R

enqueue O

dequeue

enqueue T

enqueue A

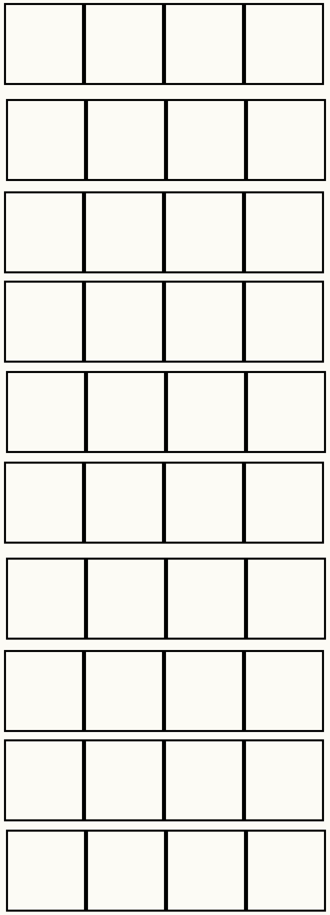
enqueue T

dequeue

dequeue

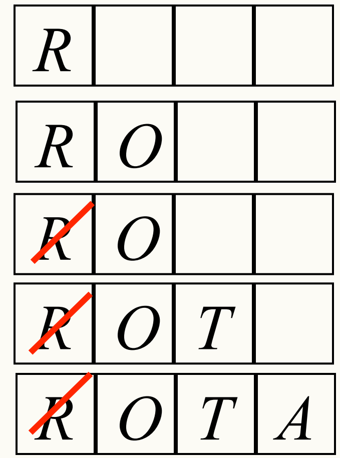
enqueue E

dequeue



Circular Array Q Example 3

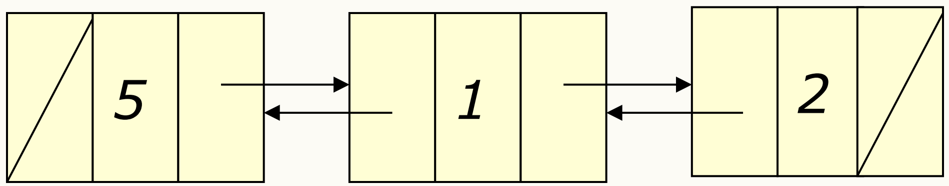
enqueue R
enqueue O
dequeue
enqueue T
enqueue A
enqueue T
dequeue
dequeue
enqueue E
dequeue



Cannot add the second T

Linked Lists

- Consider the following abstraction, picturing a short linked list:



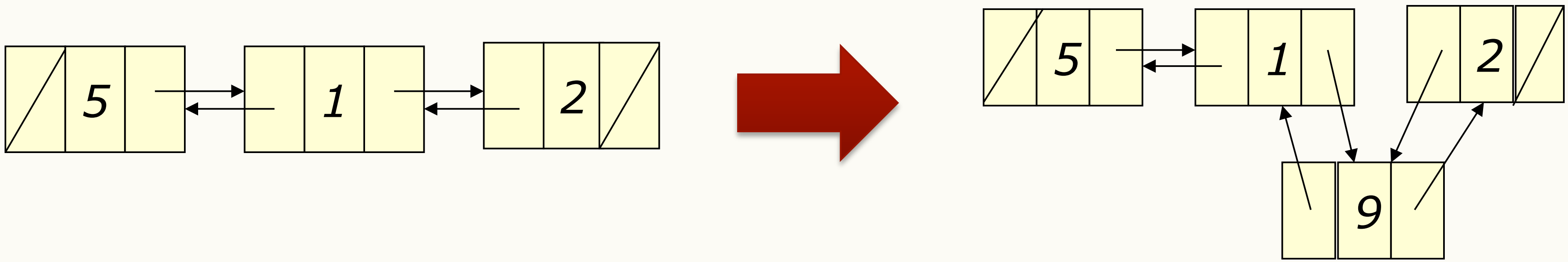
Diagonal line represents NULL

- What might it look like in memory?

```
struct Node
{
    Node *previous;
    int  data;
    Node *next;
};
```

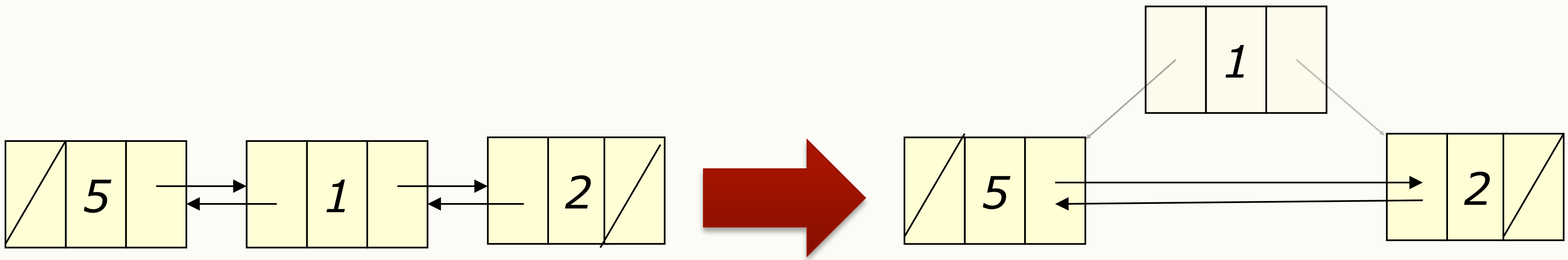
	data	*previous*	*next*
1080	2	100	Null
…			
600	5	Null	100
…			
140			
120			
100	1	600	1080

Inserting an Element to a Linked List



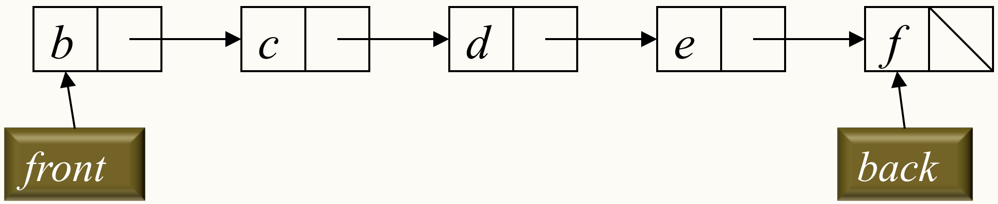
	data	*previous*	*next*
1080	2	140	Null
…			
600	5	Null	100
…			
140	9	100	1080
120			
100	1	600	140

Removing an Element from a Linked List

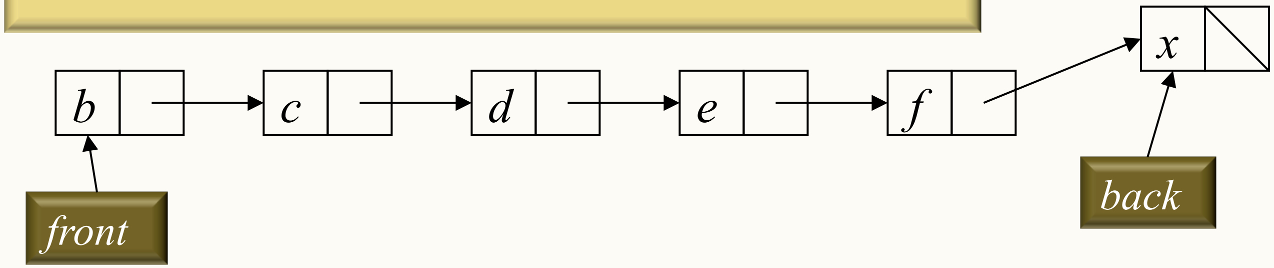


	data	*previous*	*next*	
1080	2	600	Null	
…				
600	5	Null	1080	
…				
140				
120				
100	1	600	1080	← *delete*

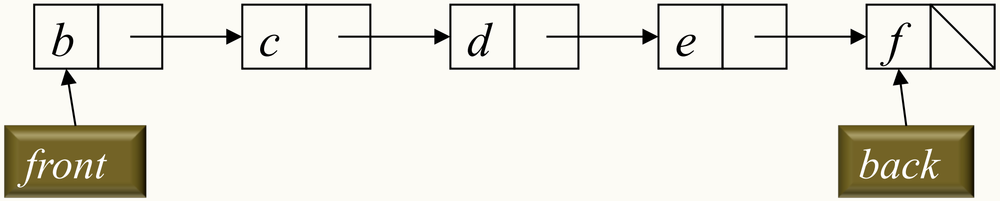
Linked List Q Data Structure




```
void enqueue(Object x) {
    if (is_empty())
        front = back = new Node(x);
    else {
        back->next = new Node(x);
        back = back->next;
    }
}
```



Linked List Q Data Structure

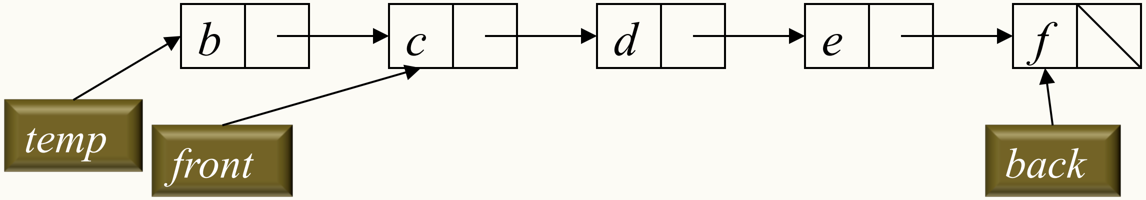



```
Object dequeue() {
    assert(!is_empty);
    char result = front->data;
    Node * temp = front;
    front = front->next;
    delete temp;
    return result;
}

bool is_empty() {
    return front == NULL;
}
```

Welcome to manual memory management!

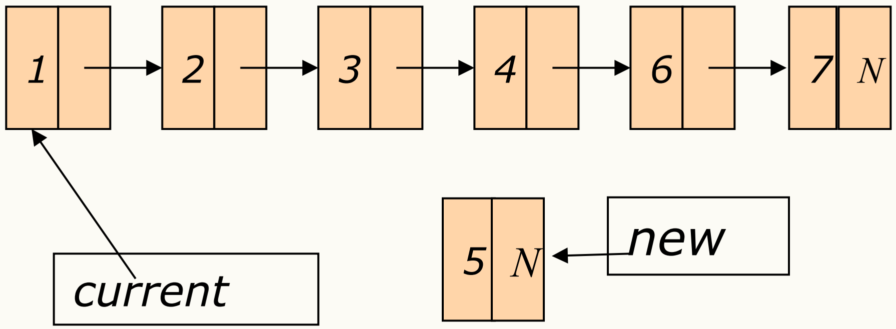
Tip: "a `delete` for every `new`"



Clicker question (Inserting into a list)

- Consider the following linked list, and possible commands

```
W: current->next = new
X: current= current->next
Y: new->next = current->next
Z: current = new
```



- Assuming that we would like to keep the list sorted, which of the following list of commands correctly inserts the new node into the list

A: X X X Y W

B: X X X X W Y

C: X X X W Y

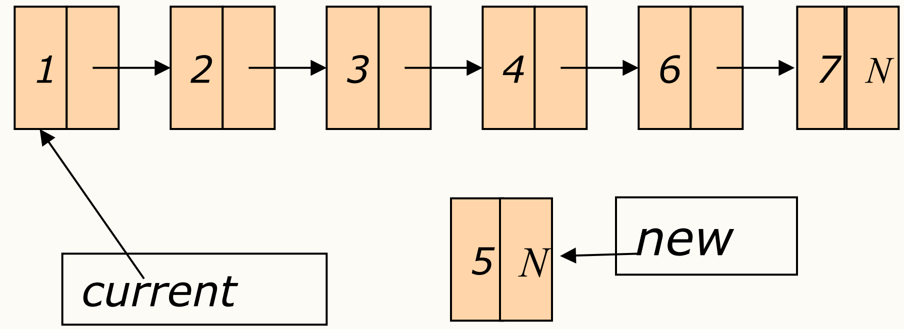
D: X X X W Z Y

E: None of the above

Clicker Question (answer)

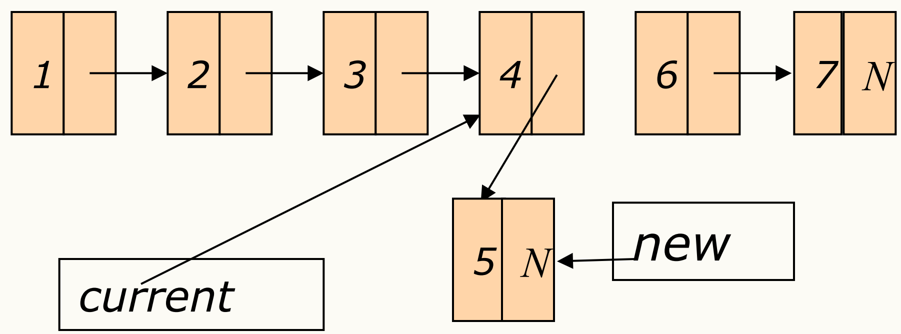
- Consider the following linked list, and possible commands

```
W: current->next = new
X: current= current->next
Y: new->next = current->next
Z: current = new
```



- Assuming that we would like to keep the list sorted, which of the following list of commands correctly inserts the new node into the list

A: X X X Y W

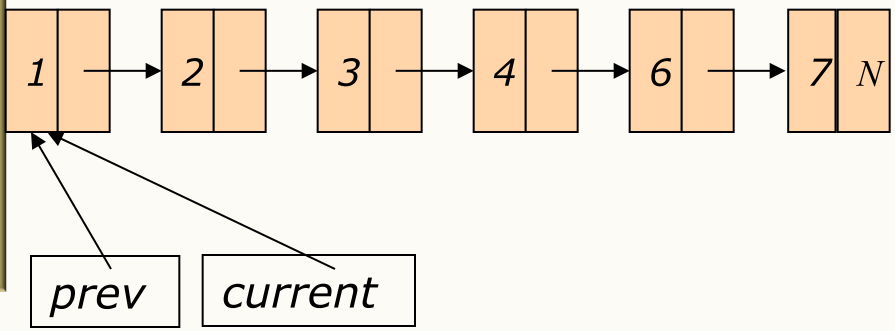


If W is performed before Y, then the second part of the list is lost

Clicker question (deleting from a list)

- Consider the following linked list, and possible commands

```
V: current= current->next
W: prev = prev->next
X: prev->next = current->next
Y: current->next = prev->next
Z: delete current; current= NULL;
```



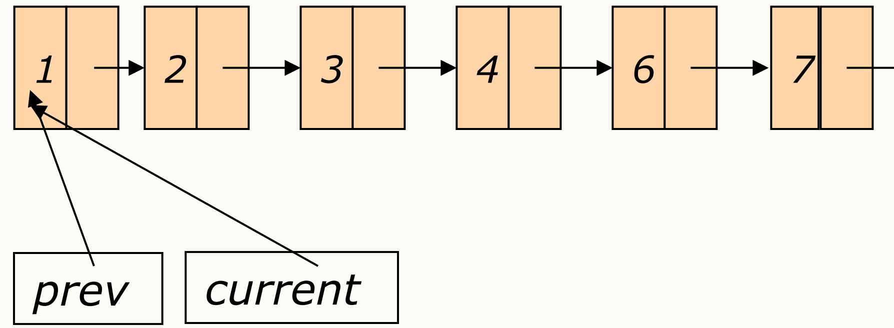
- Which one of the following list of commands correctly deletes 3 from the list

A: V W V Y Z
B: W V W X Z
C: V W V X Z
D: V V W W Y Z
E: None of the above

Clicker question (answer)

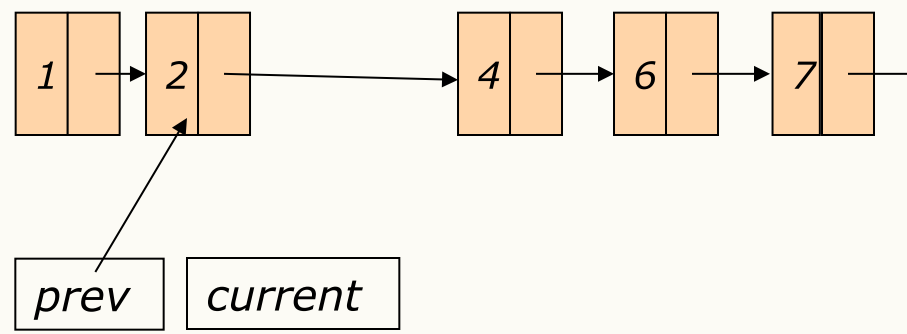
- Consider the following linked list, and possible commands

```
V: current= current->next
W: prev = prev->next
X: prev->next = current->next
Y: current->next = prev->next
Z: delete current; current= NULL;
```



- Which one of the following list of commands correctly deletes 3 from the list

C: V W V X Z



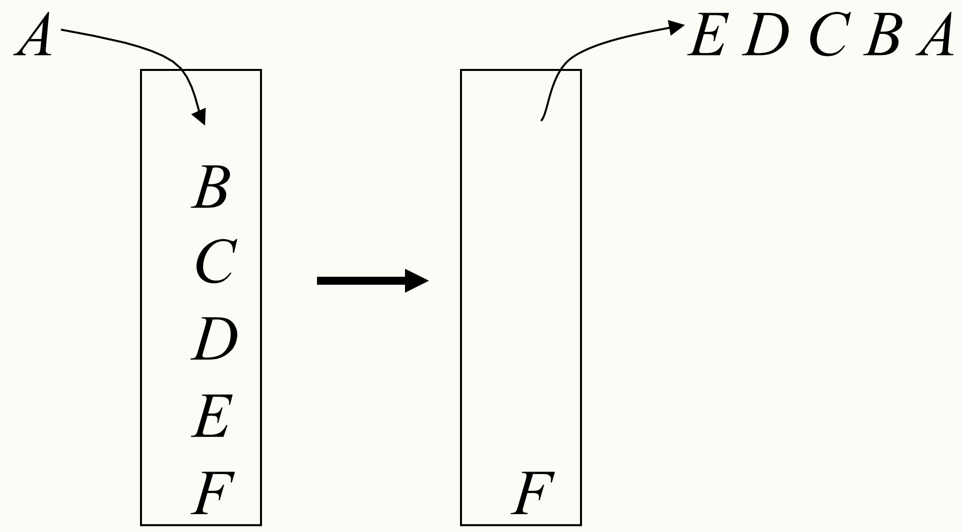
Circular Array vs. Linked List

- Ease of implementation?
- Generality?
- Speed?
- Memory use?

- In general, many different data structures can implement an ADT, each with different trade-offs. You must pick the best for your needs.

Stack ADT

- Stack operations
  - create
  - destroy
  - push
  - pop
  - top
  - is_empty



- Stack property: if x is pushed before y is pushed, then x will be popped after y is popped

LIFO: Last In First Out

Stacks in Practice (Call Stack)

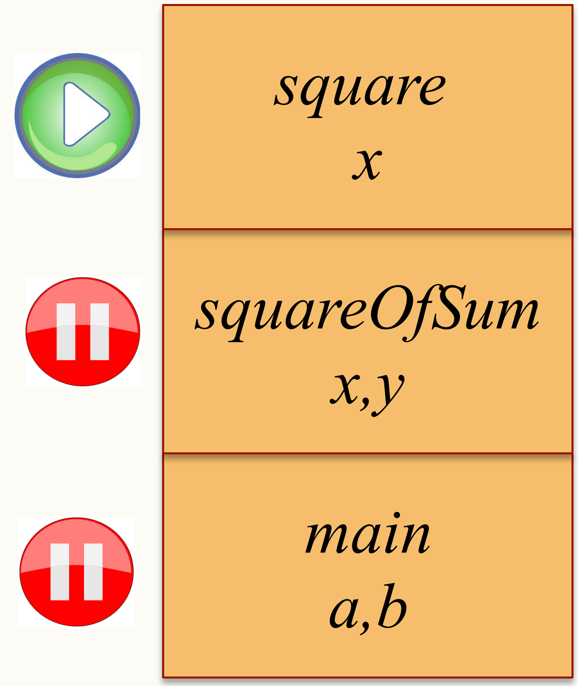
```
int square (int x){
    return x*x;
}

int squareOfSum(int x, int y){
    return square(x+y);

}

int main() {
    int a = 4;
    int b = 8;
    int total = squareOfSum(a, b);
    cout << total<< endl;
}
```

Stack

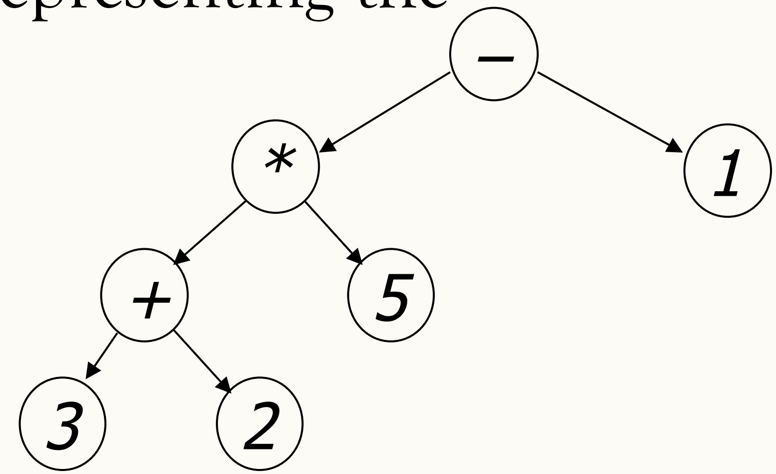


Stacks in Practice (Arithmetic expressions)

- **Application: Binary Expression Trees**

Arithmetic expressions can be represented using binary trees. We will build a binary tree representing the expression:

(3 + 2) * 5 – 1



Now let's print this expression tree using postorder traversal:

3 2 + 5 * 1 -

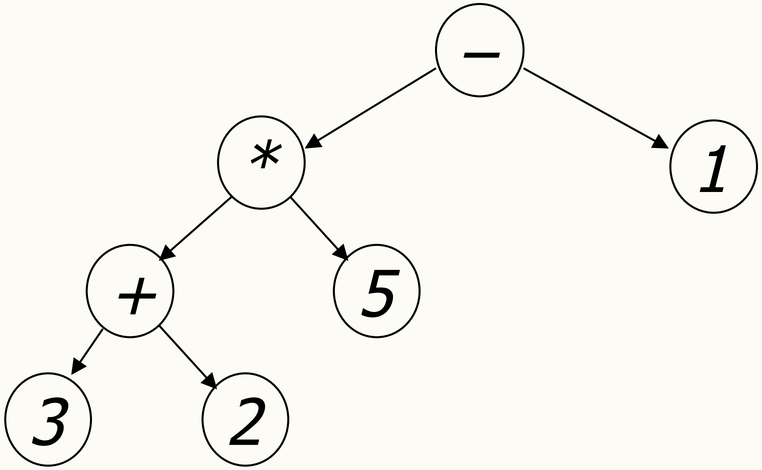
We'll cover this topic in detail later in the course

Stacks in Practice (Arithmetic expressions)

Now let's compute this expression using a Stack

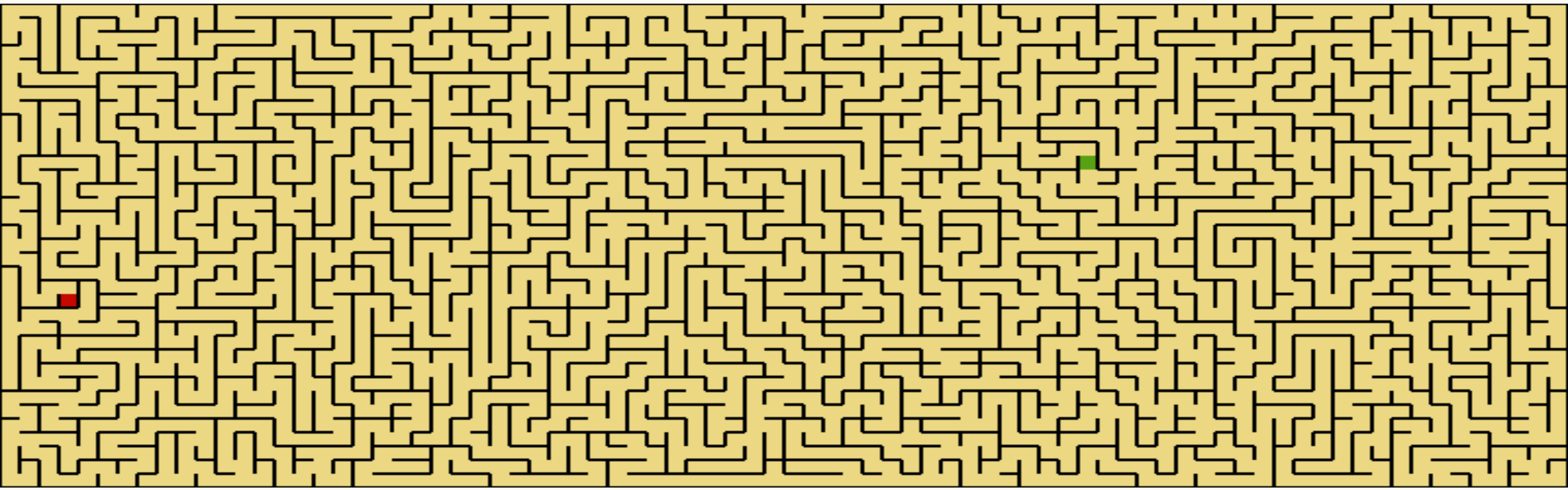
3 2 + 5 * 1 -

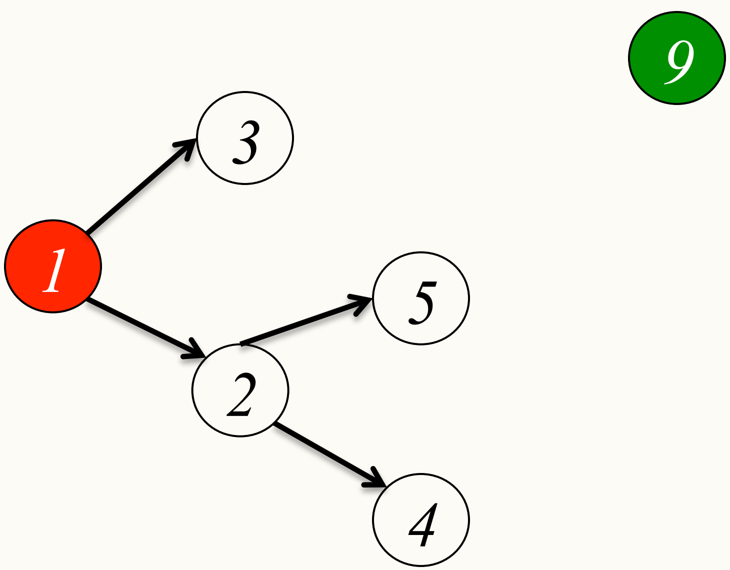
Character scanned	Stack
3	3
2	3, 2
+	5
5	5, 5
*	25
1	25,1
-	24



We'll cover this topic in detail later in the course

Stacks in Practice (Backtracking)



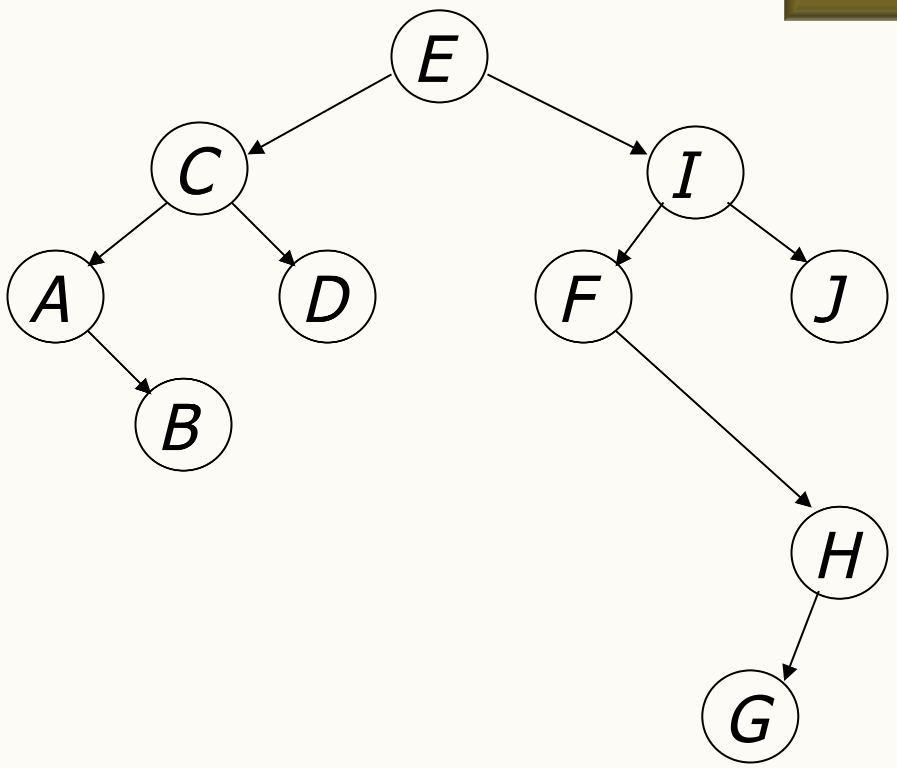


Stack
1
3, 2
3,5,4
3,5
3
9

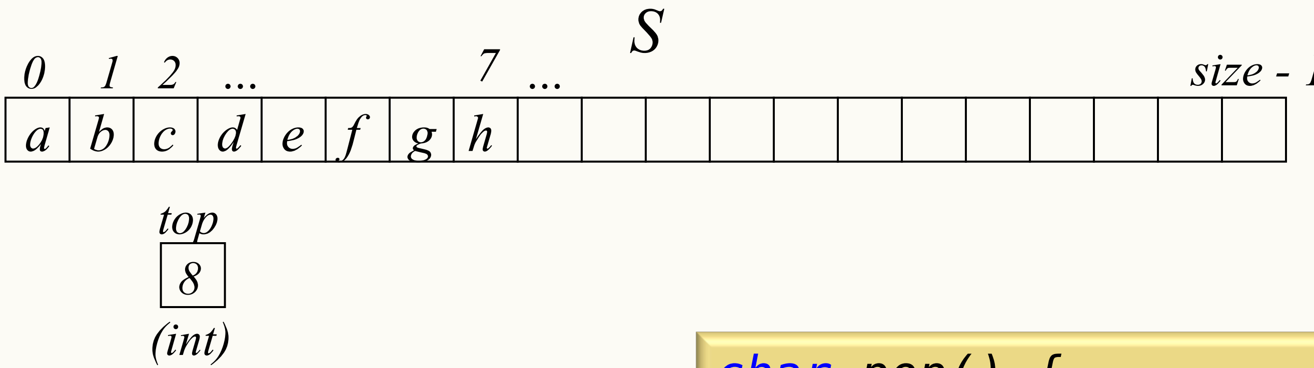
We'll cover this topic in detail later in the course

Stacks in Practice (depth first search)

We'll cover this topic in detail later in the course



Array Stack Data Structure




```
void push(char x) {
    assert(!is_full())
    S[top] = x
    top++
}

char top() {
    assert(!is_empty())
    return S[top - 1]
}
```

```
char pop() {
    assert(!is_empty())
    top--
    return S[top]
}

bool is_empty() {
    return top == 0
}

bool is_full() {
    return top == size
}
```

Example Stack with Arrays

push B

pop

push K

push C

push A

pop

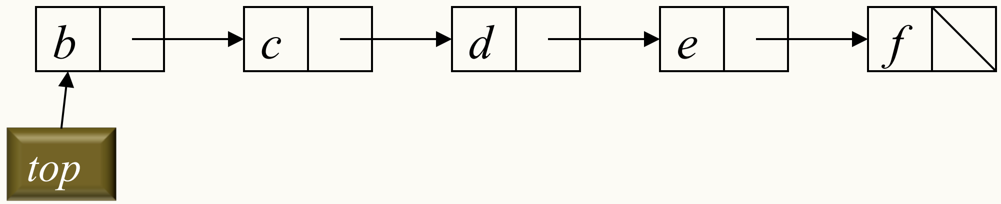
pop

pop

Example Stack with Arrays

push B	*1*	*B*			
pop	*0*				
push K	*1*	*K*			
push C	*2*	*K*	*C*		
push A	*3*	*K*	*C*	*A*	
pop	*2*	*K*	*C*		
pop	*1*	*K*			
pop	*0*				

Linked List Stack Data Structure



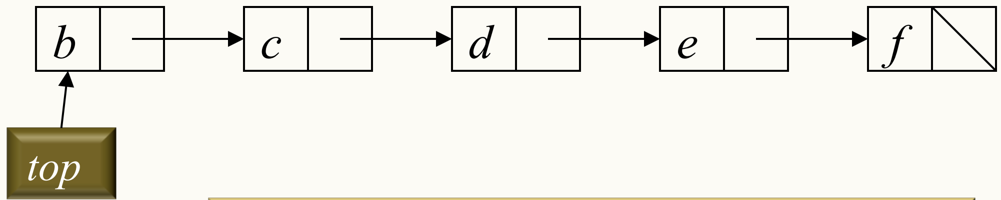

```
void push(char x) {
    temp = top;
    top = new Node(x);
    top->next = temp;
}

char top() {
    assert(!is_empty())
    return top->data;
}
```

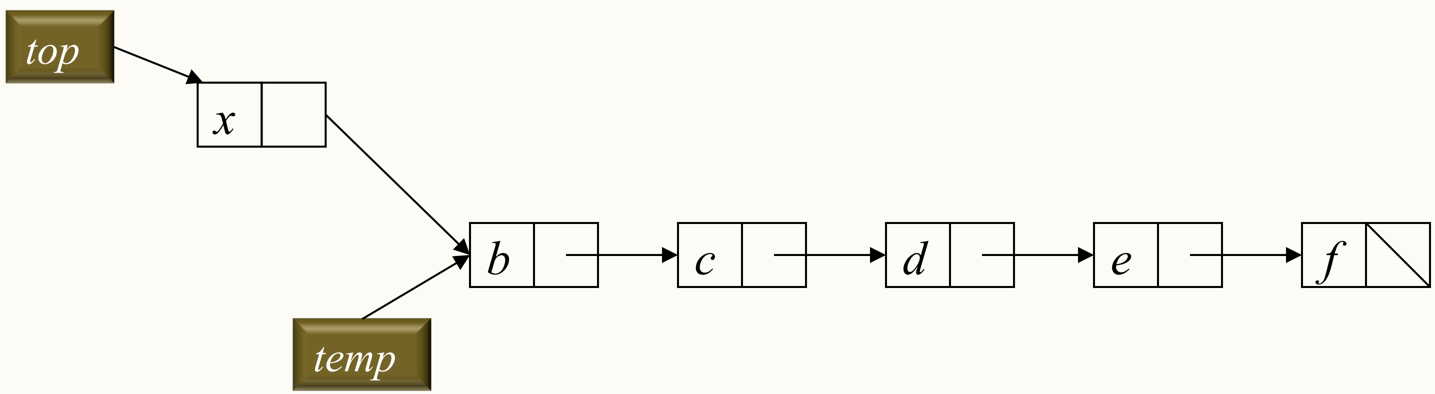
```
char pop() {
  assert(!is_empty())
  char return_data = top->data;
  temp = top;
  top = top->next;
  delete temp;
  return return_data;
}

bool is_empty() {
    return top == nullptr;
}
```

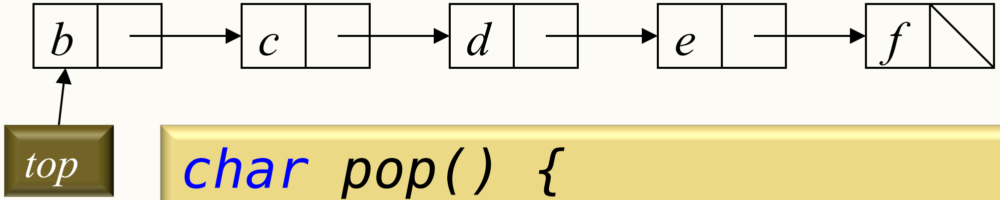
Linked List Stack Data Structure (push)



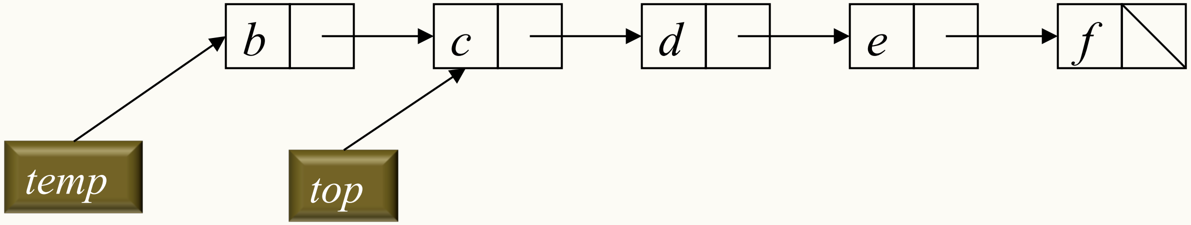

```
void push(char x) {
    temp = top;
    top = new Node(x);
    top->next = temp;
}
```



Linked List Stack Data Structure (pop)




```
char pop() {
  assert(!is_empty())
  char return_data = top->data;
  temp = top;
  top = top->next;
  delete temp;
  return return_data;
}
```



Example Stack with Linked List

- Try at home

push B

pop

push K

push C

push A

pop

pop

Learning goals revisited

- Differentiate an abstraction from an implementation.
- Define and give examples of problems that can be solved using the abstract data types stacks and queues.
- Compare and contrast the implementations of these abstract data types using linked lists and circular arrays in C++.
- Manipulate data in stacks and queues(irrespective of any implementation).