CSCI 132:
Basic Data Structures and Algorithms

Stacks (Linked List implementation)

Reese Pearsall
Spring 2023

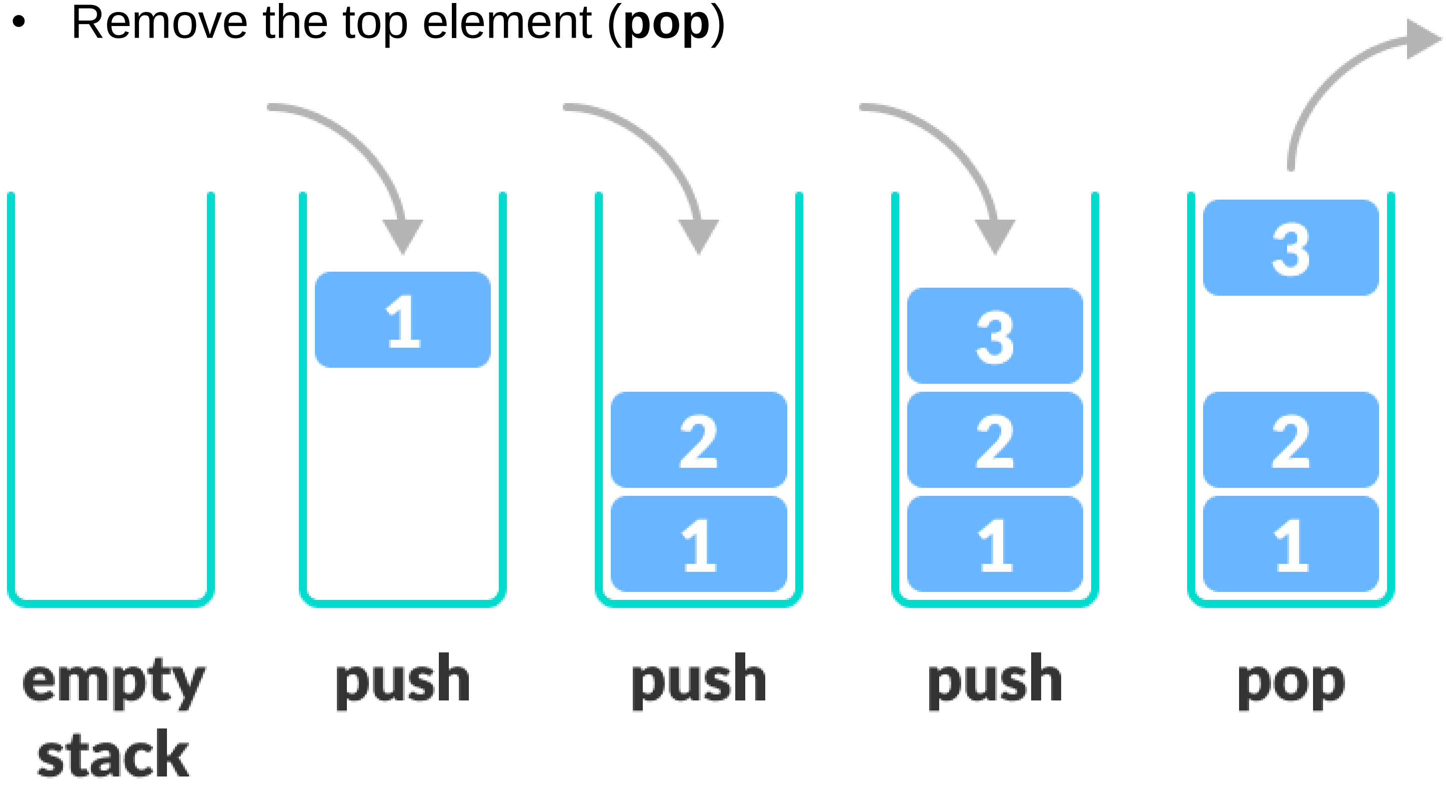
Announcements

Program 3 is Due April 2nd

A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle

We can:

- Add an element to the top of the stack (**push**)
- Remove the top element (**pop**)

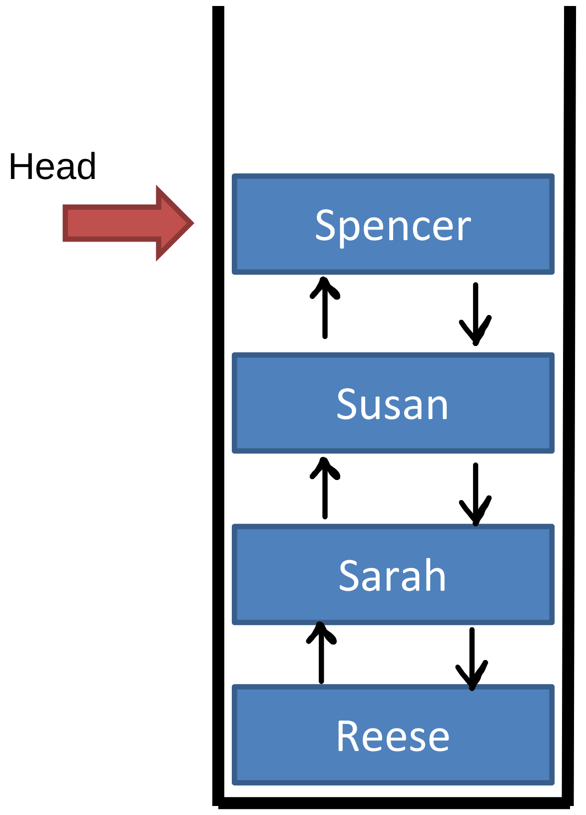


We can implement a Stack using an Array, or a linked List

Stack Implementation (Linked List)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

We will import the Linked List Library (we will not write our own linked list class)



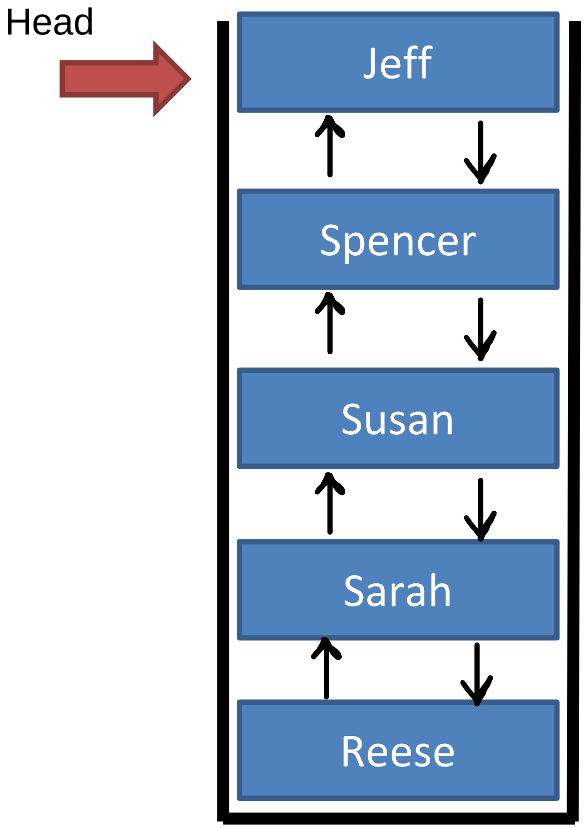
If we don’t know how big out stack needs to be ahead of time, then using a linked list will be a better choice than an array/arraylist

The top of the stack will be the head of the linked list

Stack Implementation (Linked List)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

We will import the Linked List Library (we will not write our own linked list class)




```
stack.add("Jeff")
```

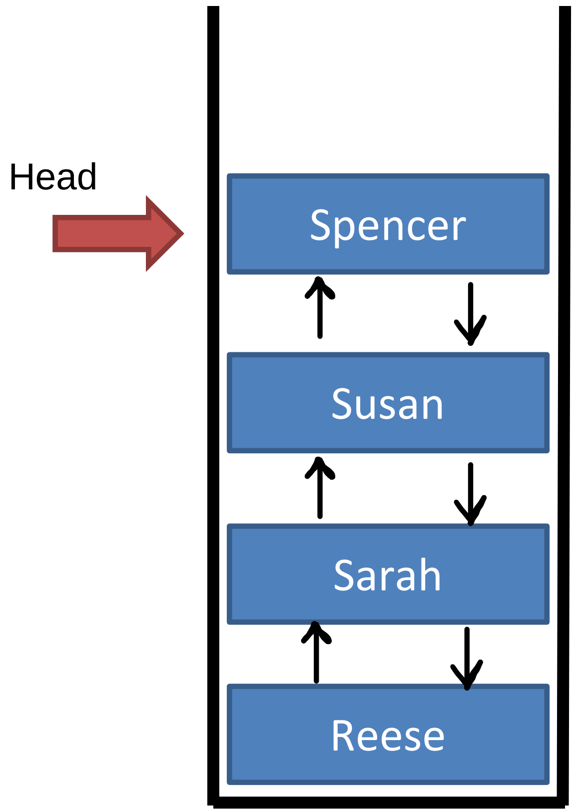
Whenever we add something to the stack, we add the element to the <u>front</u> of the linked list

The top of the stack will be the head of the linked list

Stack Implementation (Linked List)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

We will import the Linked List Library (we will not write our own linked list class)




```
stack.pop()
```

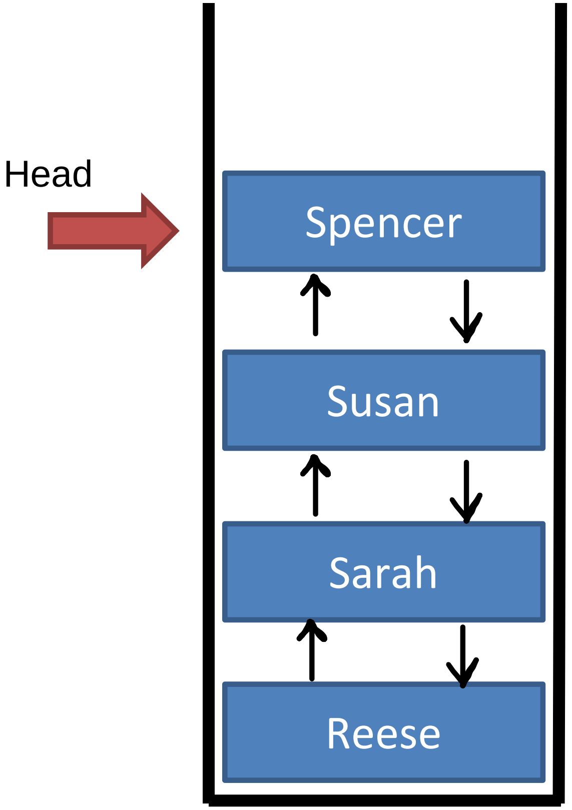
Whenever we remove an element from the stack (`pop()`), we always remove the head node of the linked list

The top of the stack will be the head of the linked list

Stack Implementation (Linked List)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

We will import the Linked List Library (we will not write our own linked list class)



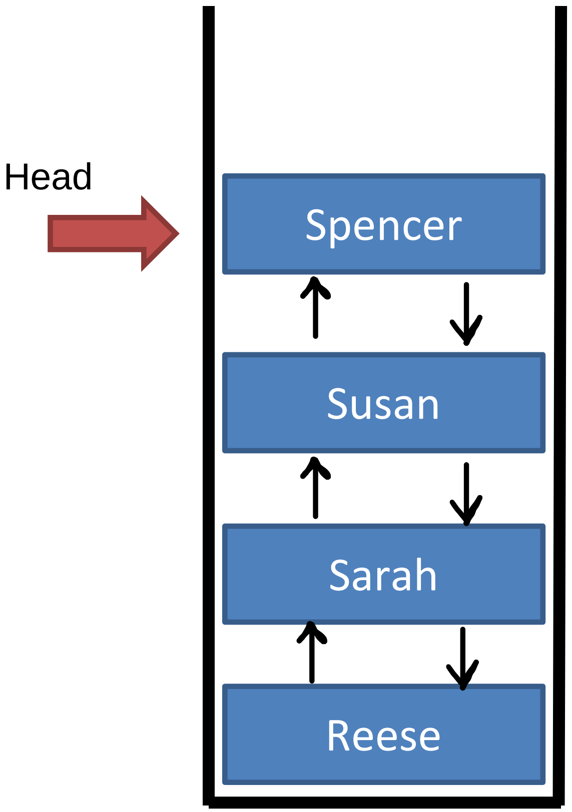

```
public void push(newElement){

    addToFront(newElement);
    size++

}
```

When we use a linked list, we are no longer restricted by a fixed size

Stack Implementation (Linked List)

We will import the Linked List Library (we will not write our own linked list class)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()




```
public void push(newElement){

   addToFront(newElement);
   size++
   top_of_stack = head

}

public void pop(){
    If size == 0:
       return
    Else:
       removeFront()
       size--
       top_of_stack = head

}
```

When we use a linked list, we are no longer restricted by a fixed size

Stack Runtime Analysis

(Array Implementation)

```
public StackArray() {
    data = new Hall[8];
    top_of_stack = -1;
    size = 0;
}
```

(Linked List Implementation)

```
public StackLinkedList() {
   data = new LinkedList<Hall>();
   top_of_stack = null;
   this.size = 0;
}
```

Algorithm	w/ Array	w/ Linked List
Creation		
Push()		
Pop()		
peek()		
Print()		

Stack Runtime Analysis

(Array Implementation)

```
public StackArray() {
    data = new Hall[8]; O(n)
    top_of_stack = -1; O(1)
    size = 0; O(1)
}
```

Total Running time: O(n) n = | array |

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()		
Pop()		
peek()		
Print()		

(Linked List Implementation)

```
public StackLinkedList() {
   data = new LinkedList<Hall>(); O(1)
   top_of_stack = null;O(1)
   this.size = 0; O(1)
}
```

Total Running time: O(1)

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()		
Pop()		
peek()		
Print()		

(Array Implementation)

```
public void push(Hall newHall) {

   if(this.size == this.data.length) {
     return;
   }
   else {
     This.top_of_stack++;
     data[this.top_of_stack] = newHall;
     this.size++;
   }
}
```

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	
Pop()		
peek()		
Print()		

(Array Implementation)

```
public void push(Hall newHall) {

    if(this.size == this.data.length) { O(1)
      return;  O(1)
    }
    else {
      This.top_of_stack++;  O(1)
      data[this.top_of_stack] = newHall; O(1)
      this.size++;O(1)
    }
}
```

Total Running Time: O(1)

Stack Runtime Analysis

(Linked List Implementation)

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	
Pop()		
peek()		
Print()		

```
public void push(Hall newHall) {

    data.addFirst(newHall);
    this.top_of_stack = this.data.getFirst();
    this.size++;

}
```

Stack Runtime Analysis

(Linked List Implementation)

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()		
peek()		
Print()		

```
public void push(Hall newHall) {

    data.addFirst(newHall); O(1)
    this.top_of_stack = this.data.getFirst(); O(1)
    this.size++; O(1)

}
```

Total Running Time: O(1)

Stack Runtime Analysis

(Array)

```
public void pop() {
    if(this.size == 0) {
        return;
    }
    else {
        this.data[this.top_of_stack] = null;
        this.top_of_stack--;
        this.size--;
    }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()		
peek()		
Print()		

(Linked List)

```
public void pop() {
    if(this.size == 0) {
        return;
    }
    else {
        this.data.removeFirst();
        this.top_of_stack = this.data.getFirst();
        this.size--;
    }
}
```

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()		
Print()		

(Array)

```
public void pop() {
    if(this.size == 0) {  O(1)
        return;  O(1)
    }
    else {
        this.data[this.top_of_stack] = null; O(1)
        this.top_of_stack--; O(1)
        this.size--; O(1)
    }
}
```

Total Running Time: O(1)

```
public void pop() {  (Linked List)
    if(this.size == 0) {  O(1)
        return; O(1)
    }
    else {  O(1)
        this.data.removeFirst(); O(1)
        this.top_of_stack = this.data.getFirst(); O(1)
        this.size--; O(1)
    }
}
```

Total Running Time: O(1)

Stack Runtime Analysis

(Array)

```
public Hall peek() {
   if(this.size != 0) {
      return this.data[this.top_of_stack];
   }
   else {
     return null;
   }
}
```

(Linked List)

```
public Hall peek() {
   if(this.size != 0) {
      return this.top_of_stack;
   }
   else {
      return null;
   }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()		
Print()		

Stack Runtime Analysis

(Array)

```
public Hall peek() {
   if(this.size != 0) {
      return this.data[this.top_of_stack];
   }
   else {
     return null;
   }
}
```

O(1)

O(1)

O(1)

Total Running Time: O(1)

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()		

(Linked List)

```
public Hall peek() {
   if(this.size != 0) {
      return this.top_of_stack;
   }
   else {
      return null;
   }
}
```

O(1)

O(1)

O(1)

O(1)

Total Running Time: O(1)

Stack Runtime Analysis

(Array)

```java
public void printStack() {
   for(int i = this.size-1; i >= 0; i--) {
      this.data[i].printInfo();
   }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()		

(Linked List)

```java
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
       this.data.get(i).printInfo();
    }
}
```

Stack Runtime Analysis

(Array)

```
public void printStack() {
    for(int i = this.size-1; i >= 0; i--) {
        this.data[i].printInfo();
    }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()		

Both of these for loops go through the stack and print out all N elements → O(n) where n = # of elements in the stack

(Linked List)

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {
        this.data.get(i).printInfo();
    }
}
```

Stack Runtime Analysis

(Array)

```
public void printStack() {                    O(n)
    for(int i = this.size-1; i >= 0; i--) {
        this.data[i].printInfo(); O(1)
    }
}
```

Total Running Time: O(n)

(Linked List)

```
public void printStack() {
    for(int i = 0; i < this.data.size(); i++) {  O(n)
        this.data.get(i).printInfo();  O(1)
    }
}
```

Total Running Time: O(n)

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

n = # of elements in the stack

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Takeaways: Adding and removing elements from a stack runs in constant time (`O(1)`) *(we like algorithms that run in constant time!!)*

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Takeaways: Adding and removing elements from a stack runs in constant time (`O(1)`) *(we like algorithms that run in constant time!!)*

Downside: Stacks operate in a LIFO structure, which might not be ideal for some data

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

With an array, our stack size is limited by the size of the array

With a linked list, our stack can grow infinitely*

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Arrays are more memory efficient (contiguous memory), but there might be a lot of unused space in an array (not ideal)

Stack Runtime Analysis

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Do you know how big the stack needs to be?

Yes→ Array

No → Linked List