Data Structures
Stacks and Queues

Phil Tayco

Slide version 1.4

Sep. 7, 2021

Stacks and Queues

Order redefined

- Some situations require a different sense of order
  - Order of function calls made requiring knowing where to go back to after a function is complete
  - People coming into a line to buy tickets while the current customer is making their purchase
  - Mathematical expressions using parentheses need to know what formula to continue with when a subcalculation completes
  - Multitasking systems need to know what context the last process was in when another task is being executed
- In each of these situations, knowing what and where the last state is when another task takes place is the recurring pattern

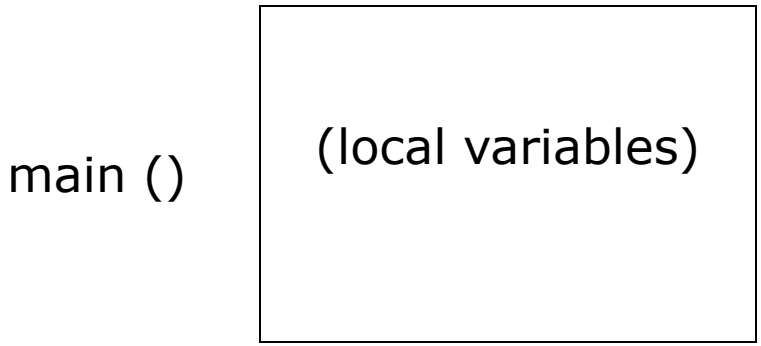
Stacks and Queues

Function Calls

- Take a program that starts at main, calls a function f and within it calls a function g
- When a function makes a call, all the variables in scope are suspended and the location where the call is made is recorded
- The new function now has control, creating and manipulating variables as needed within its own context
- When that function completes, all variables created within its scope are removed and control goes back to the last function where the call was made
- This form of order is known as Last In First Out

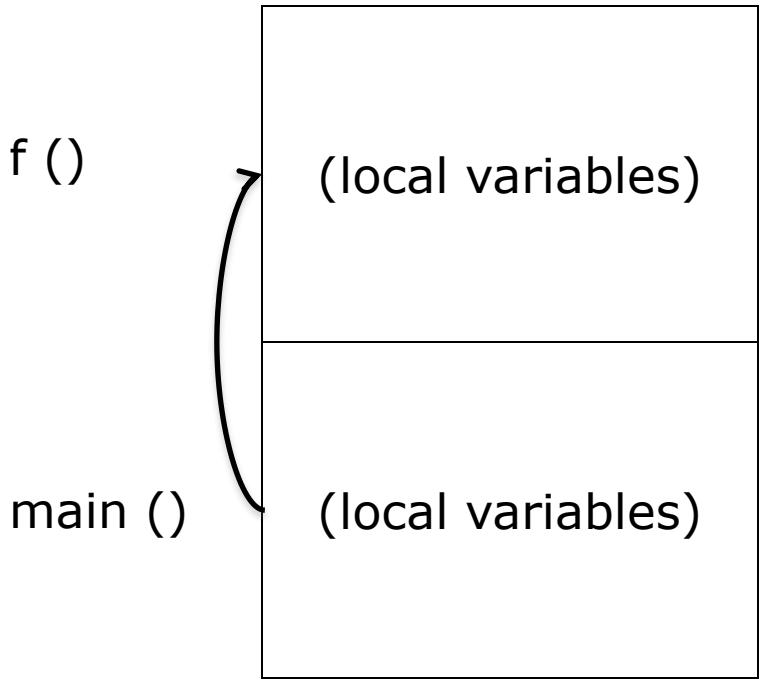
Stacks and Queues

main() starts



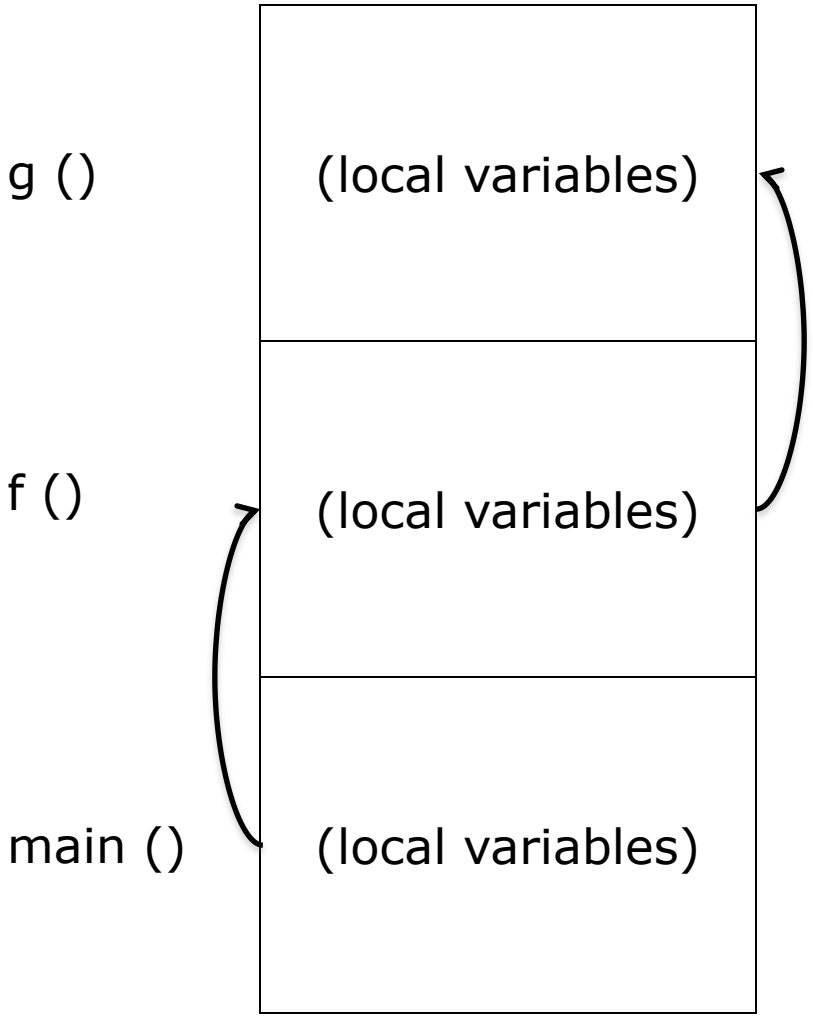
Stacks and Queues

main calls f().



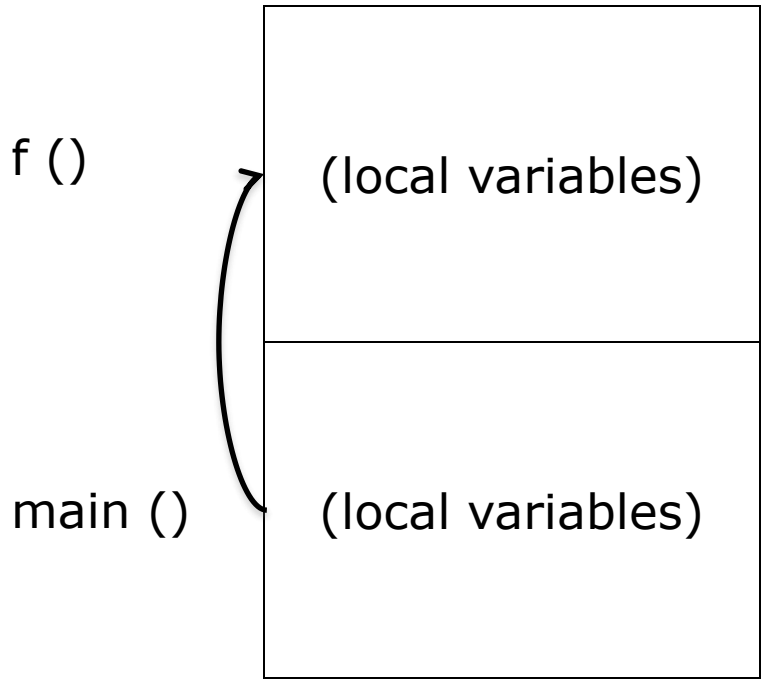
Stacks and Queues

f() calls g()



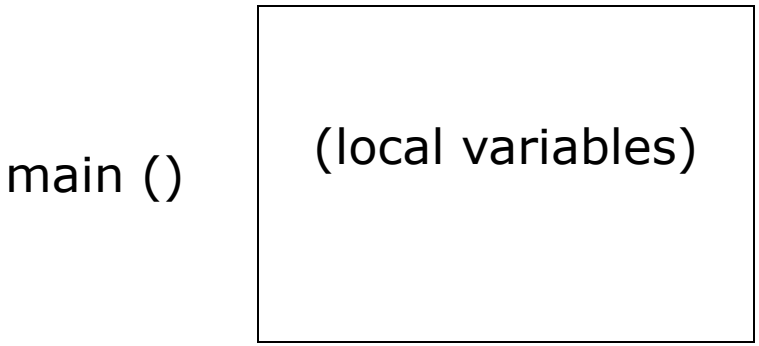
Stacks and Queues

g() finishes. f() resumes



Stacks and Queues

f() finishes. main() resumes



Stacks and Queues

LIFO

- Last in first out (LIFO) situations mean the last element "inserted" will be the first one to complete
- In the previous example, main calls f, then f calls g. Function g is the last function to be called and is the one currently being executed
- When g completes, the compiler must now go back to the last function that made the call. This means it goes back to function f where it made the call to g
- Similarly, when function f is complete, control returns to the next function in the structure, which in this case is main

Stacks and Queues

Stack

- The LIFO structure is best represented in what we call a “stack”
- Think of a stack of cafeteria trays. Each tray represents a function call:
  - The most recent function call is the tray on “top” of the stack
  - Each time a new function call is made, a tray is added on top of the stack
  - When a function completes, the top tray is removed and control moves to the next tray (the last one called)
- Order is based on the manner in which elements are added and removed
- The contents of the elements do not determine the order

Stacks and Queues

Stack terminology

- Stack insert and delete functions are known by different terms:
  - “Push”: insert a new element onto the stack
  - “Pop”: delete the last element on the stack
- Search and update are usually not addressed, but can be done if needed
- Functions that are traditionally associated with stacks are those that provide information about its current state:
  - “Top”: shows the element on top of the stack
  - “isEmpty”: tells if there are any elements in the stack
  - “size”: tells how many elements are in the stack

Stacks and Queues

Implementation

- Physically maintaining a stack can be done with either an array or linked list
- The rules for how data gets in and out of the structure is programming logic. Whether it is done with an array or linked list is independent
- Linked lists offer dynamic memory allocation making them more likely to use for implementation

Stacks as Array

```
public class NameStackArray
{
  public int currentSize;
  public String[] stack;

  public boolean isEmpty();
  public boolean isFull();
  public String top();
  public int size();
  public boolean push(String);
  public String pop();
}
```

Stacks as Array

Analysis

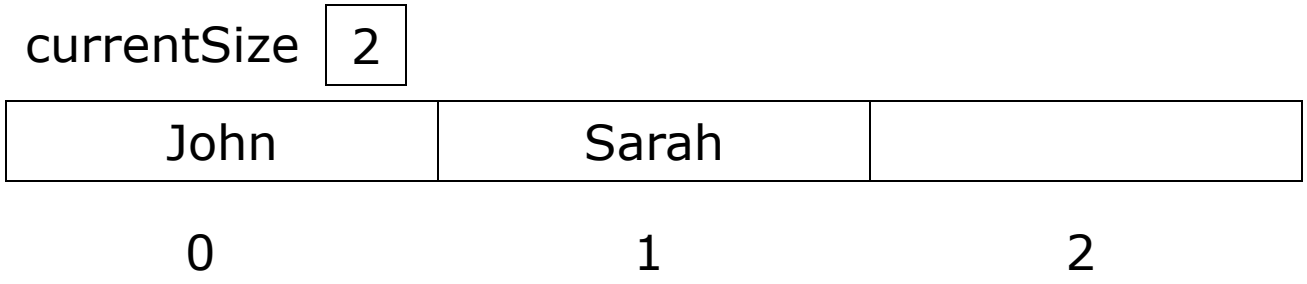
- The array implementation does not look much different from a regular array
- The different functions maintain how data is used and goes in and out, but the fundamental implementation is just a simple array
- The same pros and cons with arrays will hold where memory must be reserved and is static, while direct random access of any element can be done
- With a stack, though, the need for random access is not necessary since all major interaction is through the top of the stack
- This suggests a linked list implementation is better, but let’s first analyze the performance of the stack as an array

Stacks as Array

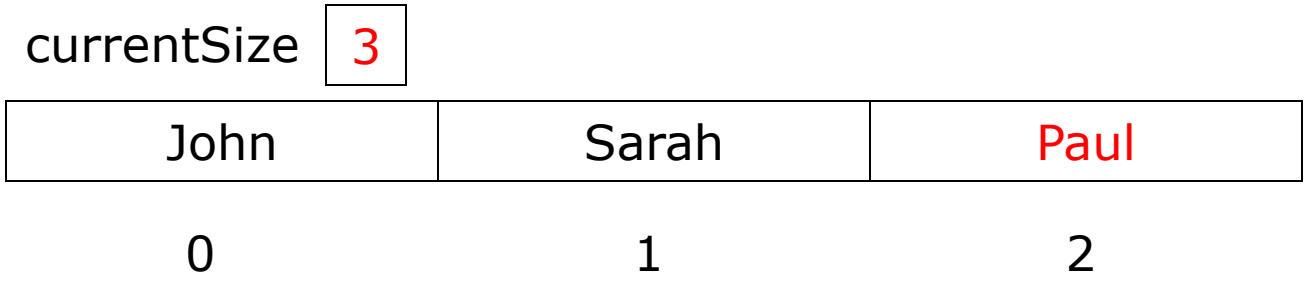
```
public boolean push(String f)
{
  if (currentSize == stack.length)
     return false;

  stack[currentSize++] = f;
  return true;
}
```

Stacks as Array



After push("Paul");



Stacks as Array

Push Analysis

- With a stack like an unordered array, the push function is the same as the append function
- Consequently, the performance is O(1) and a simple implementation!

Pop algorithm

- With an unordered array, the delete function requires searching for a particular value, removing it from the list and shifting elements to prevent holes
- For a stack, the only element that can be removed by logical rule is the last element that was pushed (the element on “top”)
- This significantly reduces the complexity of the pop function

Stacks as Array

```
public String pop()
{
  if (currentSize == 0)
     return null;

  String f = stack[currentSize - 1];
  stack[--currentSize] = null;
  return f;
}
```

Stacks as Array

currentSize | 3 |

John	Sarah	Paul
0	1	2

After pop(); // this will return the String, "Paul"

currentSize | 2 |

John	Sarah	
0	1	2

Stacks as Array

Pop Analysis

- The only comparison being made in the code is to check if the stack is empty
- After that, we have a consistent amount code being executed making the performance of this algorithm also O(1)
- Unordered array deletes are O(n) because of the search for a particular value. Pops in a stack do not require a search because by rule, the item to delete is the last one that was pushed
- This is excellent to see in that with LIFO situations, the performance of adding and removing items is constant

Stacks as Array

Other functions

- Question: if there was a need to perform search or update, what would the performance be?
- We could improve on that like we analyzed with earlier with unsorted arrays, but with the situation the way it is, there's a chance that the need to search could be unnecessary
- Other functions like “top”, “size”, and “isEmpty” are also all O(1)

Stacks as Array

```
public boolean isEmpty()
{
   return (currentSize == 0);
}

public String top()
{
   if (currentSize == 0)
       return null;
   return stack[currentSize - 1];
}

public int size()
{
   return currentSize;
}
```

Stacks as Array

Summary

- Stacks implemented as an array behaves the same as an unordered array
- Because the delete function can only take place through the top of the stack, we get a significant performance improvement to O(1) for pop
- As an array, one of the cons is the need to reserve space that may go unused
- This suggests stacks as a Linked List implementation may be better if memory usage is a factor

Stacks as Linked List

```
public class Node
{
  public String data;
  public Node next;
}

public class NameStackLinkedList
{
  public int currentSize;
  public Node stack;
}
```

Stacks as Linked List

Push and Pop

- With an array, the “top” of the stack ended up being the last element of the array
- As a linked list, we can treat the “head” node as the top of the stack and perform a push like an insert at the front of the list
- For pop, the last element pushed will be the head element, so that will be a matter of removing the first element (situation 2 of the delete function for a Linked List)

Stacks as Linked List

```
public boolean push(String f)
{
  Node temp = new Node();

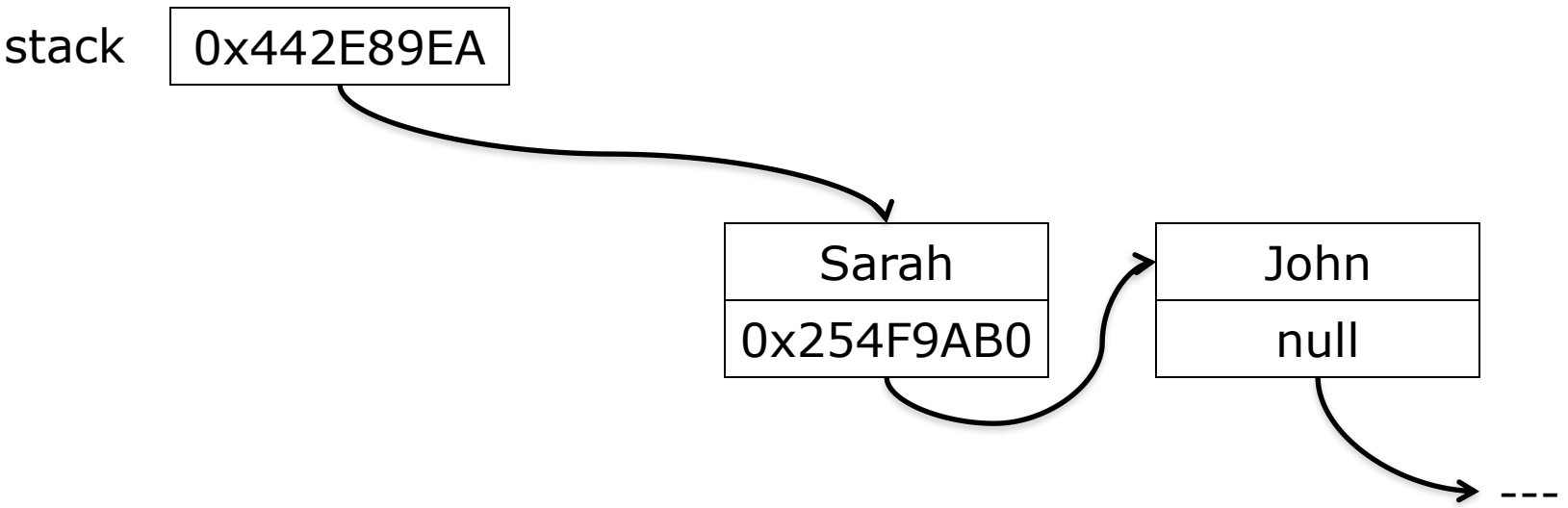
  temp.data = f;
  temp.next = stack;
  stack = temp;
  currentSize++;

  return true;
}
```

Stacks as Linked List

```
push("Paul");
```

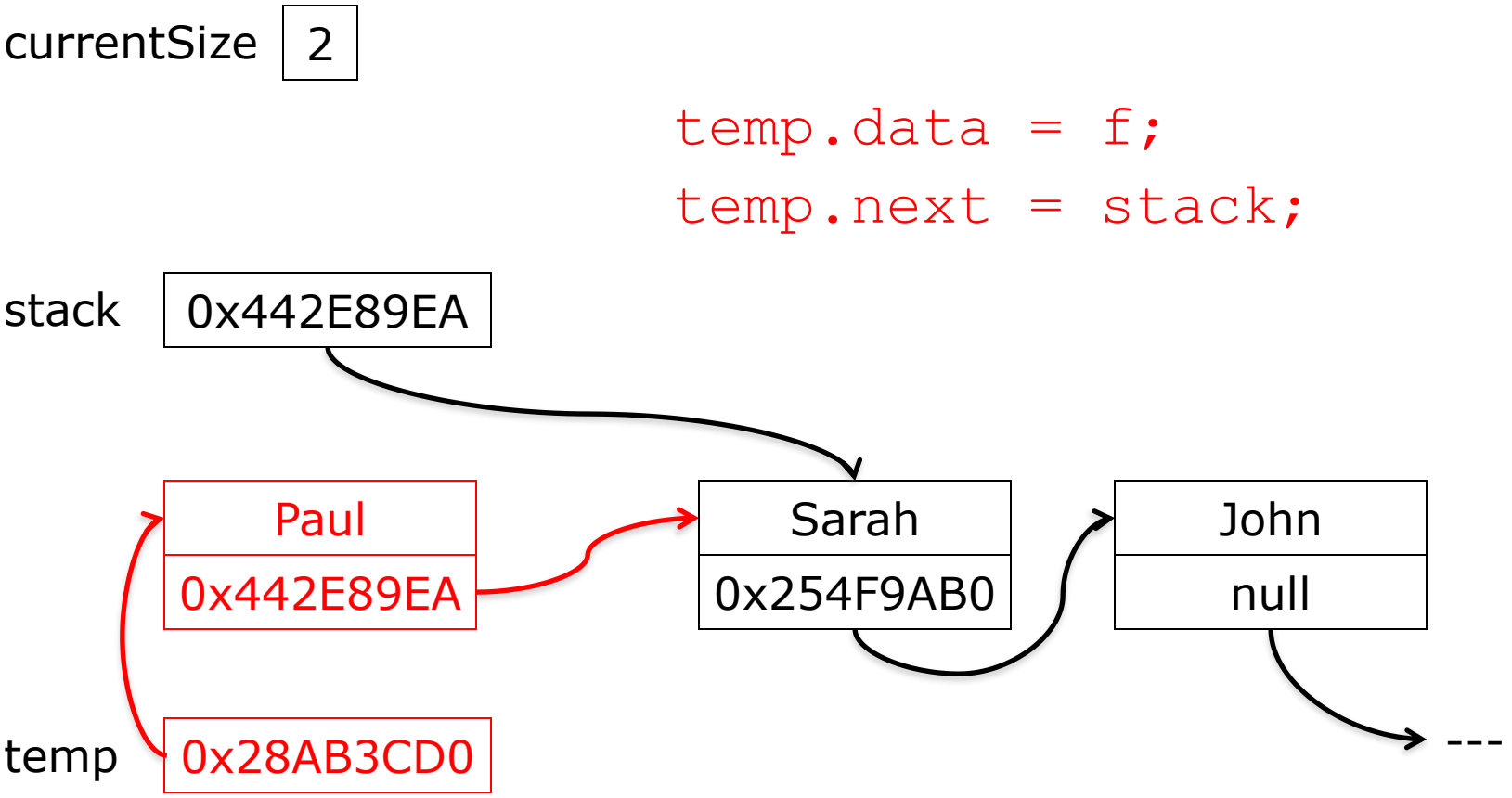
currentSize | 2



Stacks as Linked List

```
Node temp = new Node();

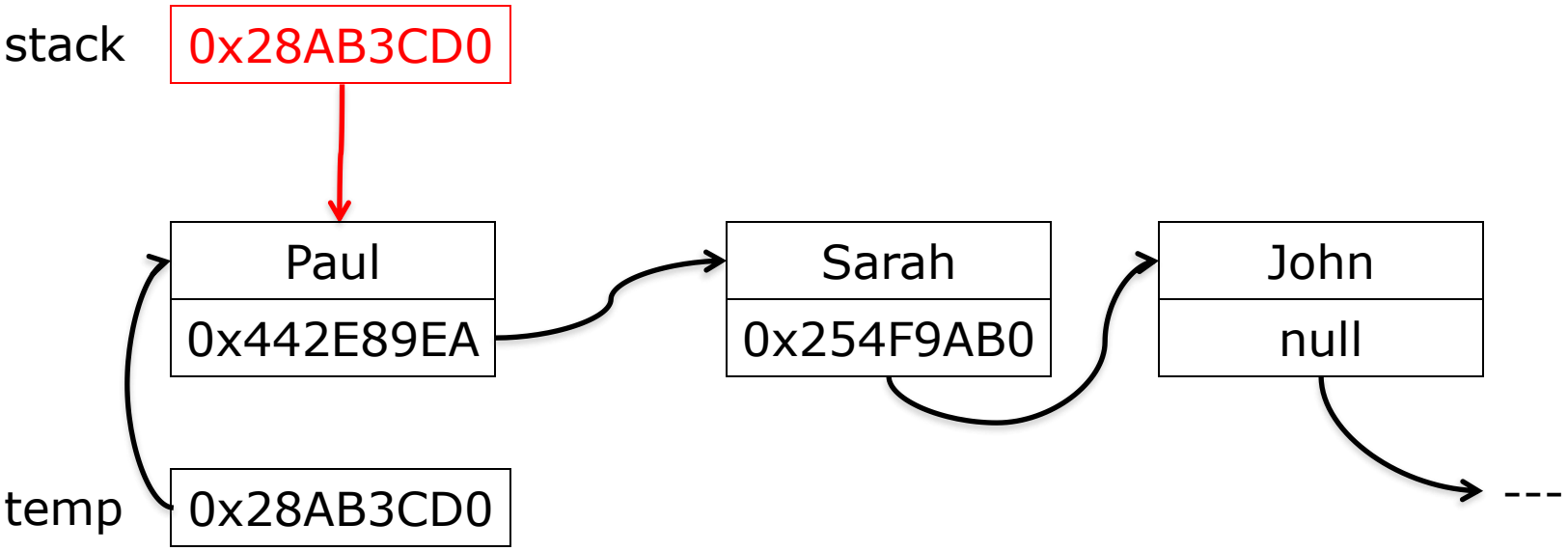
temp.data = f;
temp.next = stack;
```



Stacks as Linked List

```
stack = temp;
currentSize++;
```

currentSize | 3



Stacks as Linked List

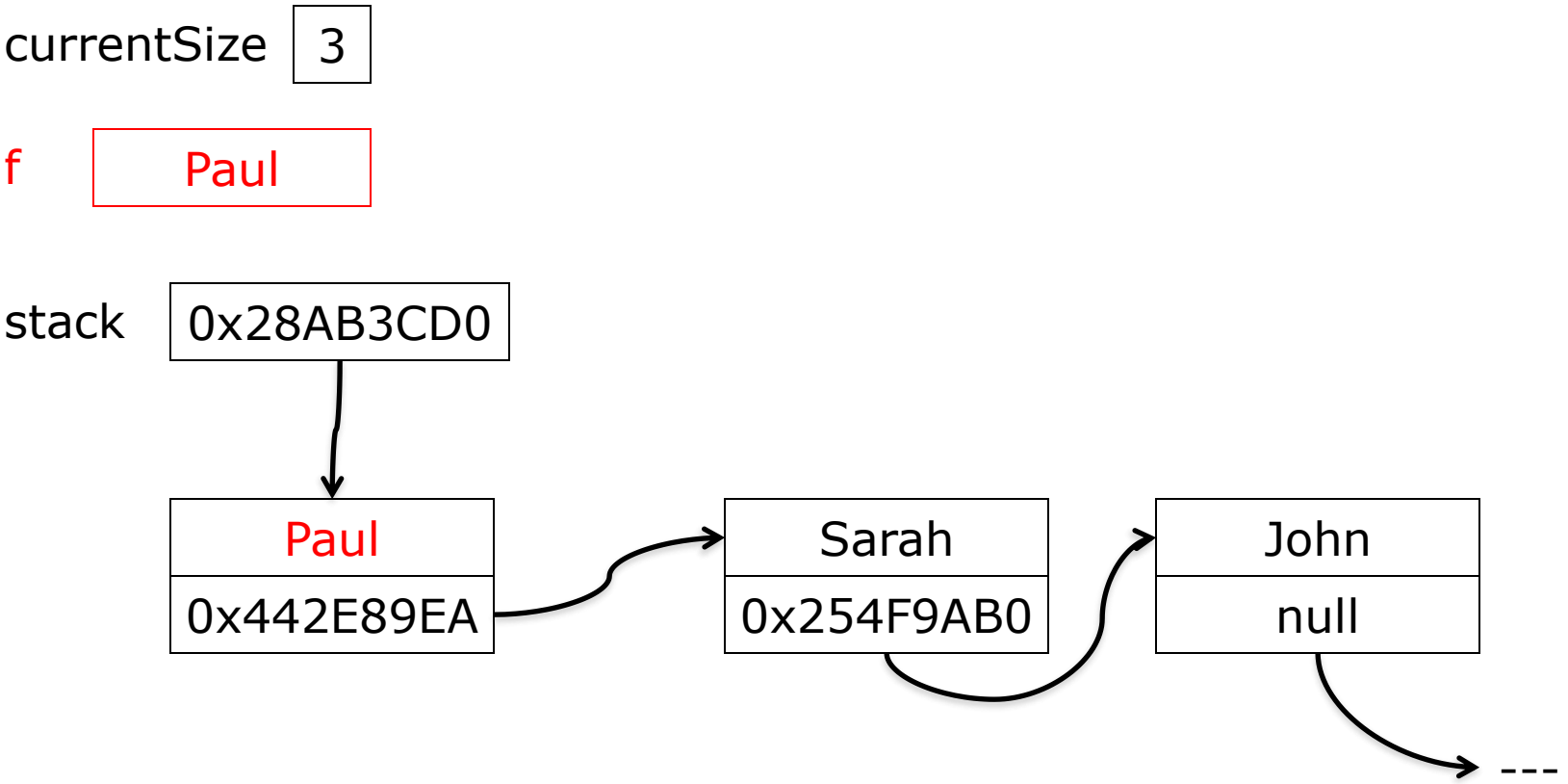
```
public String pop()
{
  if (currentSize == 0)
     return null;

  String f = stack.data;
  stack = stack.next;
  currentSize--;

  return f;
}
```

Stacks as Linked List

```
String f = stack.name;
```



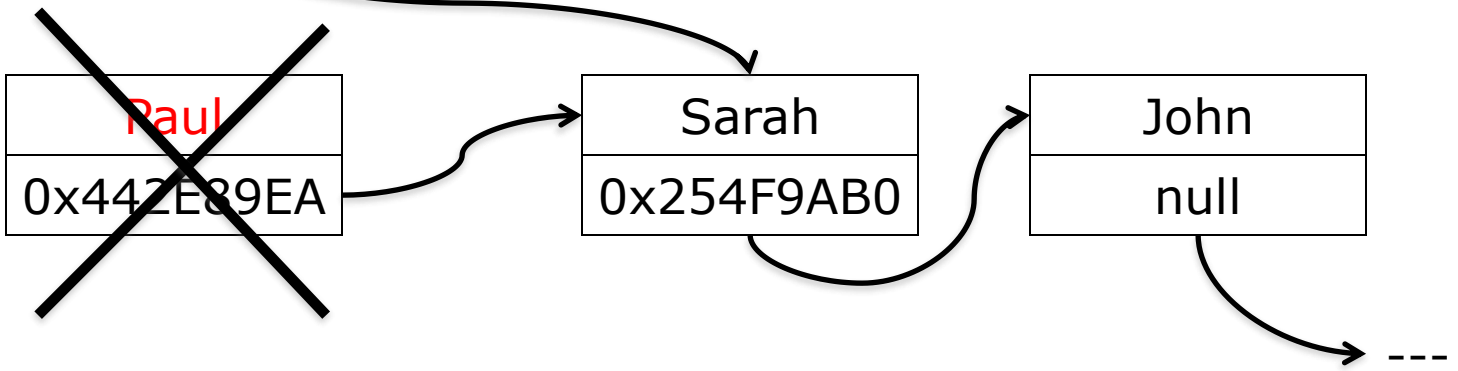
Stacks as Linked List

```
stack = stack.next;
currentSize--;
```

currentSize 2

f Paul

stack 0x442E89EA



Stacks as Linked List

Analysis

- Like arrays, the standard functionality handled by linked lists are simply applied in the context of a stack. Insert and delete at the head of the list equates to push and pop respectively
- If memory management is a concern, linked list implementations of stacks are better suited, but for performance, the two approaches work at the same Big-O category
- Other functions such as top, size and isEmpty are the same between arrays and linked list stack implementations

Stacks

Summary:

Worst Case	Unsorted Arrays	Unsorted Linked Lists	Stacks
Search	O(n)	O(n)	O(n) if needed
Insert (Push)	O(1)	O(1)	O(1)
Update	O(n)	O(n)	O(n) if needed
Delete (Pop)	O(n)	O(n)	O(1)
	Static memory usage	Dynamic memory usage	

Queues

Network Printing

- Stacks have useful applications when the situation is LIFO
- What about jobs on a printer or waiting in line to buy a ticket at the movie theater?
- If you use LIFO, the order of processing isn't fair (since the last one entering the structure is the next one to get processed)
- In these situations, it's "first come, first serve" or in using a similar acronym, FIFO for First In, First Out

Queues

Get in line

- The data structure associated with FIFO is referred to as a “queue”, which is another term for a waiting line
- Similar to stacks, there are rules for how data is inserted and removed from the queue
- There is also a notion of what element is on “top” being the next one to be removed from the list, but the top is not the last one entered into the queue
- Question: do you think an array or linked list implementation makes a difference in performance?

Queues as Array

Keep it at O(1)

- In a queue, one end of the array serves as the insert point while the other serves as the exit
- To keep the push and pop functions to O(1), we must avoid shifting elements in the array
- Add elements to the end of the array maintaining the addIndex property as the location of the next element to add
- Remove elements using the removeIndex property
- Always maintain the currentSize as the actual number of elements in the queue

Queues as Array

```
public class NameQueue
{
   public int currentSize = 0;
   public int removeIndex = 0;
   public int addIndex = 0;
   public String[] queue = new String[100];

   public boolean isEmpty();
   public boolean isFull();
   public String top();
   public int size();
   public boolean insert(String);
   public String remove();
}
```

Queues as Array

```
public boolean insert(String f)
{
  if (currentSize == queue.length)
     return false;

  queue[addIndex++] = f;
  if (addIndex == queue.length)
     addIndex = 0;

  currentSize++;
  return true;
}
```

Queues as Array

currentSize 1 addIndex 1 removeIndex 0



0 1 2

After insert("Paul");

currentSize 2 addIndex 2 removeIndex 0



0 1 2

After insert("Sarah");

currentSize 3 addIndex 0 removeIndex 0



0 1 2

Queues as Array

Insert analysis

- The currentSize and addIndex properties update accordingly with each insert
- removeIndex is untouched throughout entire process
- If addIndex reaches end of the array, it cycles back to the beginning of the array
- The example shows a full queue but checking addIndex does not mean checking for addIndex equals currentSize. This is because of how the remove function works
- The order of the insert function stays at O(1)
- Question: why not just use the append function to add new elements to the queue?

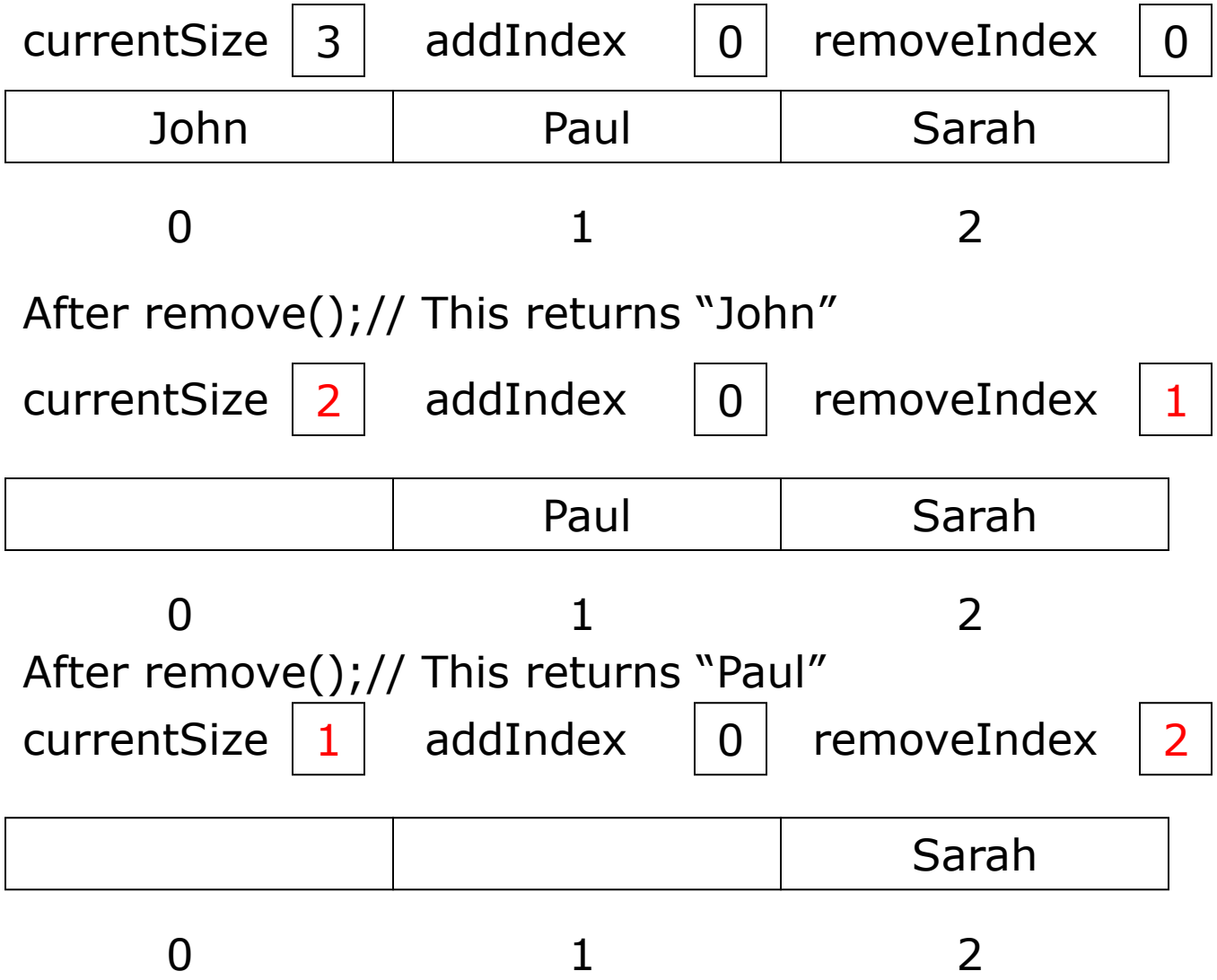
Queues as Array

```
public string remove()
{
  if (currentSize == 0)
     return null;

  String f = queue[removeIndex];
  queue[removeIndex++] = “”;
  if (removeIndex == queue.length)
     removeIndex = 0;

  currentSize--;
  return f;
}
```

Queues as Array



Queues as Array

Remove analysis

- Similar to insert, the currentSize and removeIndex properties update accordingly with each remove
- addIndex is untouched throughout entire process
- If removeIndex reaches end of the array, it cycles back to the beginning of the array
- Note how currentSize now only represents the number of elements in the array and no longer acts as the next location to add an element to it
- The order of the remove function stays at O(1)
- Supporting functions are similarly managed
- Question: which variable represents where the “top” of the queue is? How does this effect displaying the queue from top to bottom?

Queues as Array

```
public boolean isEmpty()
{
   return (currentSize == 0);
}

public String top()
{
   if (currentSize == 0)
       return null;
   return queue[removeIndex];
}

public int size()
{
   return currentSize;
}
```

Queues as Array

```
public String toString()
{
   String output = "";
   int counter = 0;
   int index = removeIndex;

   while (counter < currentSize)
   {
        output += String.format("%s\n", queue[index++]);
        if (index == queue.length)
               index = 0;
        counter++;
   }

   return output;
}
```

Queues as Array

removeIndex management

- The “top” of the queue is the removeIndex
- Because of the rotational nature of the queue as an array, currentSize and removeIndex have to be managed appropriately when going through the queue from top to bottom
- The minor complication allows insert and remove to stay at O(1)

Queues as Linked List

Keep it at O(1) too

- The linked list implementation will need to stay at O(1) to maintain its advantage over arrays as using dynamic memory allocation
- In a linked list implementation of a stack, the head of the list is the top where insert and delete take place
- In a queue, one end of the list serves as the insert point while the other serves as the exit
- If we treat the head of the list as the next element to delete, the tail end of the list is the insert point
- However, if we use a standard linked list, inserting at the end of the list is an append which is an O(n) operation
- How can we reduce the performance to O(1)? We need a second node pointer

Queues

```
public class NameQueueLinkedList
{
  public Node head = null;
  public Node tail = null;
  public int currentSize = 0;

  public boolean isEmpty();
  public String top();
  public int size();
  public void insert(String);
  public String remove();
}
```

Queues

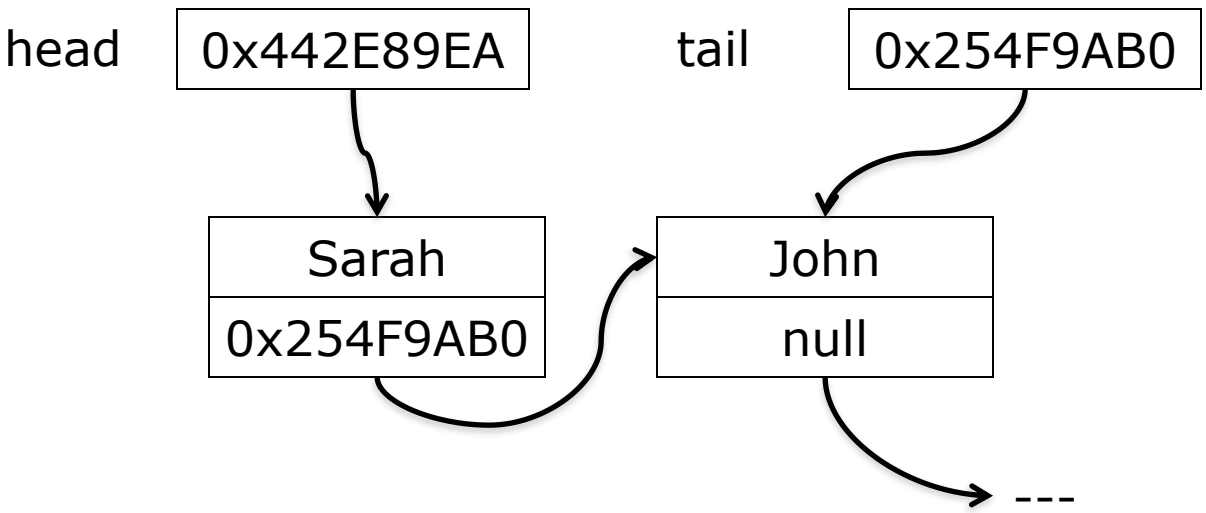
```
public void insert(String f)
{
   Node temp = new Node();
   temp.data = f;
   temp.next = null;

   if (head == null)
   {
       head = temp;
       tail = temp;
   }
   else
   {
       tail.next = temp;
       tail = temp;
   }
   currentSize++;
}
```

Queues as Linked List

```
insert("Paul");
```

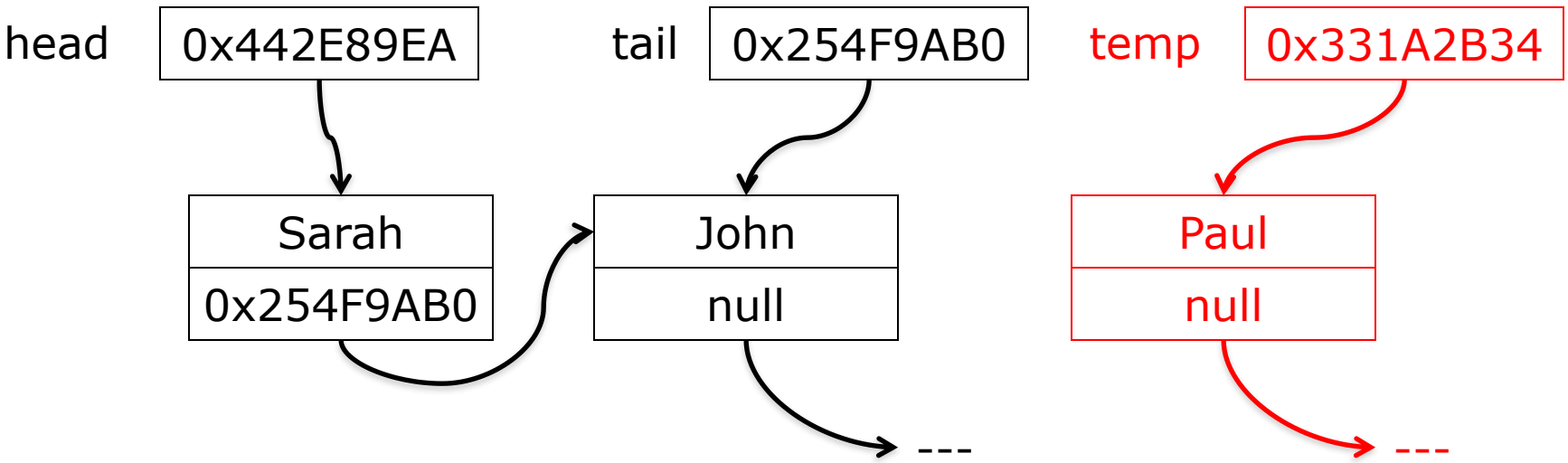
currentSize | 2



Queues as Linked List

currentSize | 2

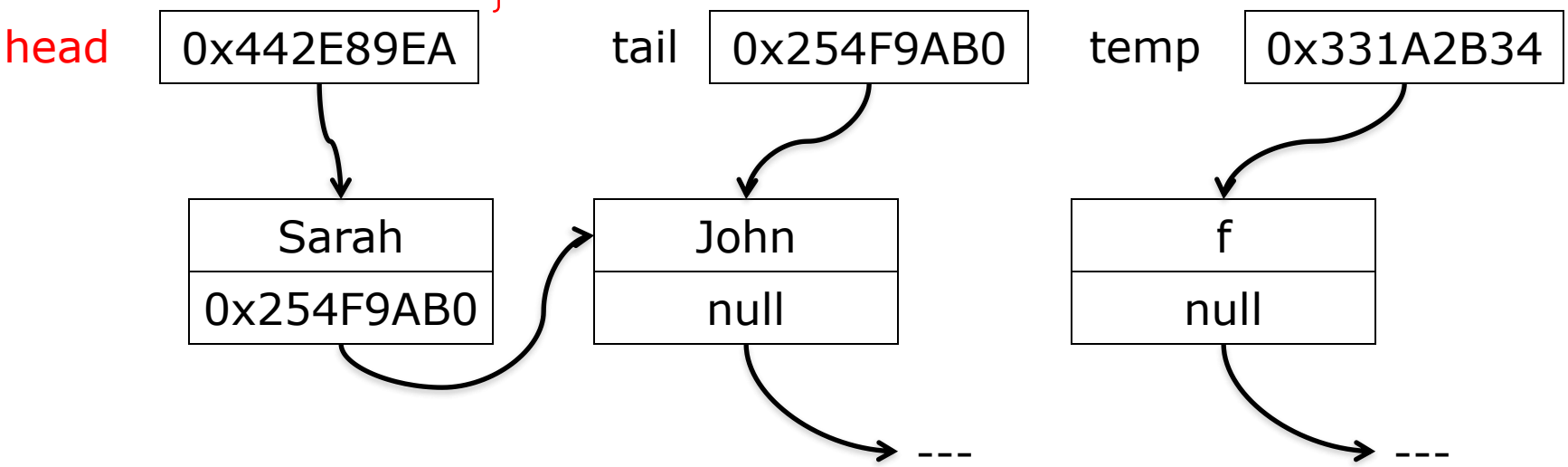
```
Node temp = new Node();
temp.data = f;
temp.next = null;
```



Queues as Linked List

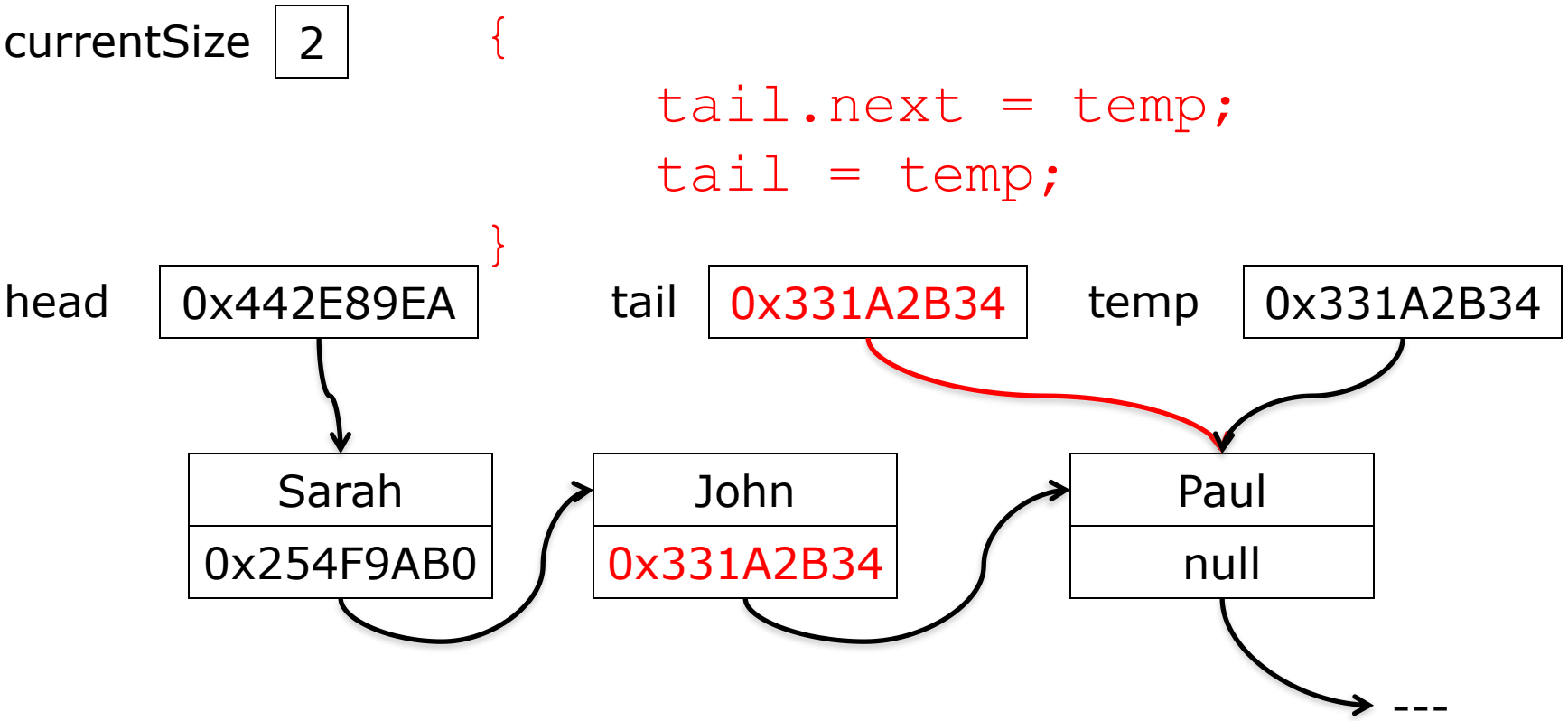
```
if (head == null)
{
    head = temp;
    tail = temp;
}
```

currentSize 2



Queues as Linked List

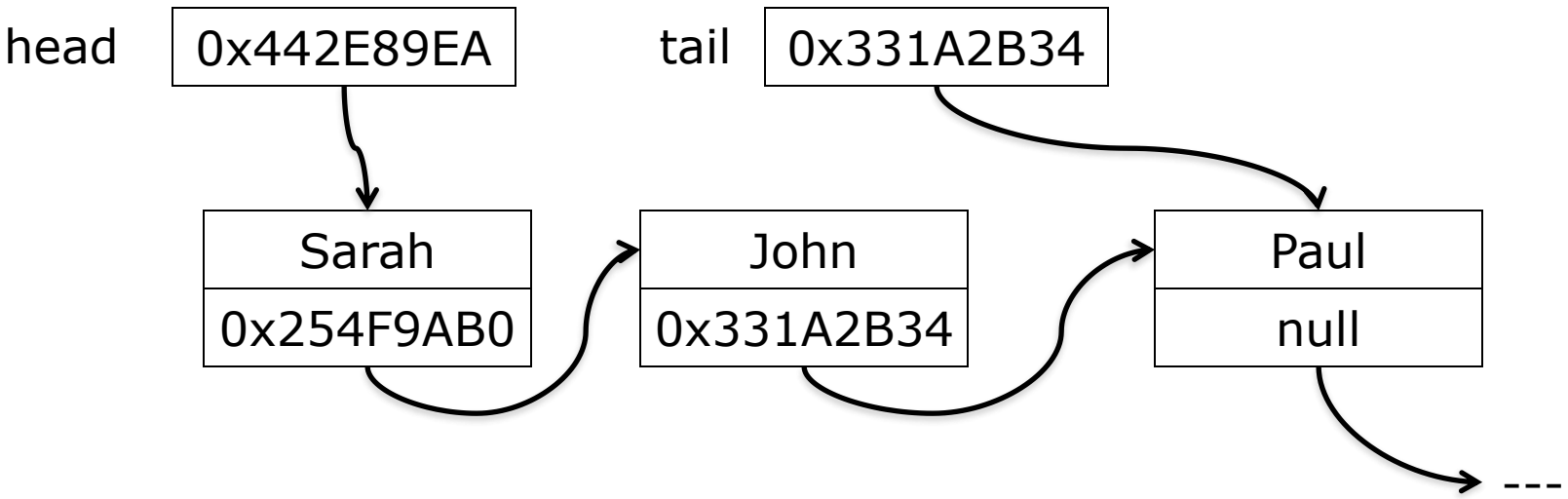
```
else
{
        tail.next = temp;
        tail = temp;
}
```



Queues as Linked List

```
currentSize++;
```

currentSize | 3



Queues

Insert Analysis

- This linked list modification is known as a “double-ended” linked list
- The insert implementation is actually an “append” in a linked list and takes advantage of the fact that a tail pointer exists
- By using this implementation, the performance of the append goes from O(n) to O(1)
- Subsequently, the remove function can remain as a delete at the head, but must also maintain managing the tail if there’s only one element left in the queue

Queues

```
public String remove()
{
  if (currentSize == 0)
     return null;

  String f = head.data;
  head = head.next;
  if (head == null)
     tail = null;
  currentSize--;

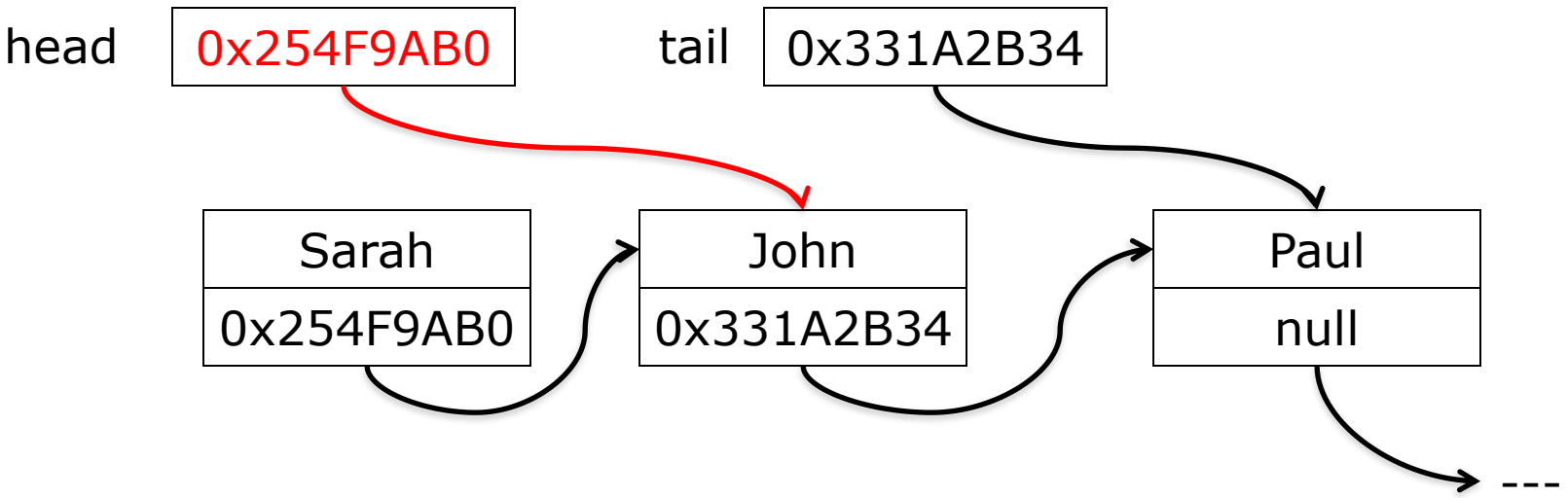
  return f;
}
```

Queues as Linked List

```
String f = head.data;
head = head.next;
```

currentSize | 3

f | Sarah

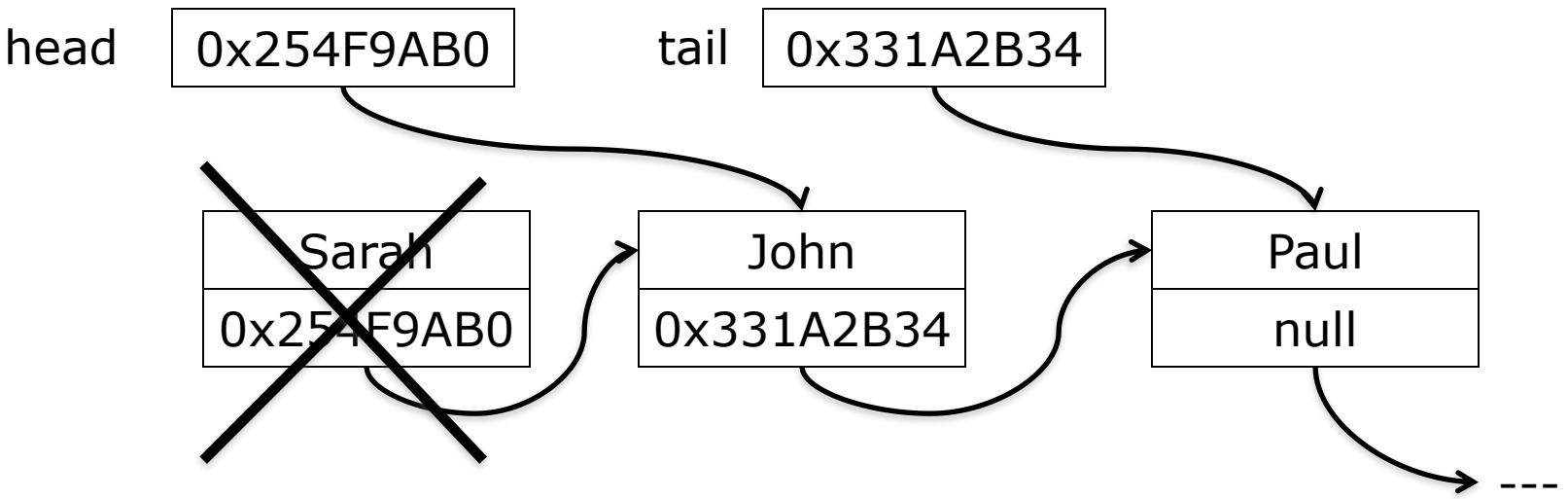


Queues as Linked List

```
if (head == null)
    tail = null;
currentSize--;
```

currentSize 3

f Sarah

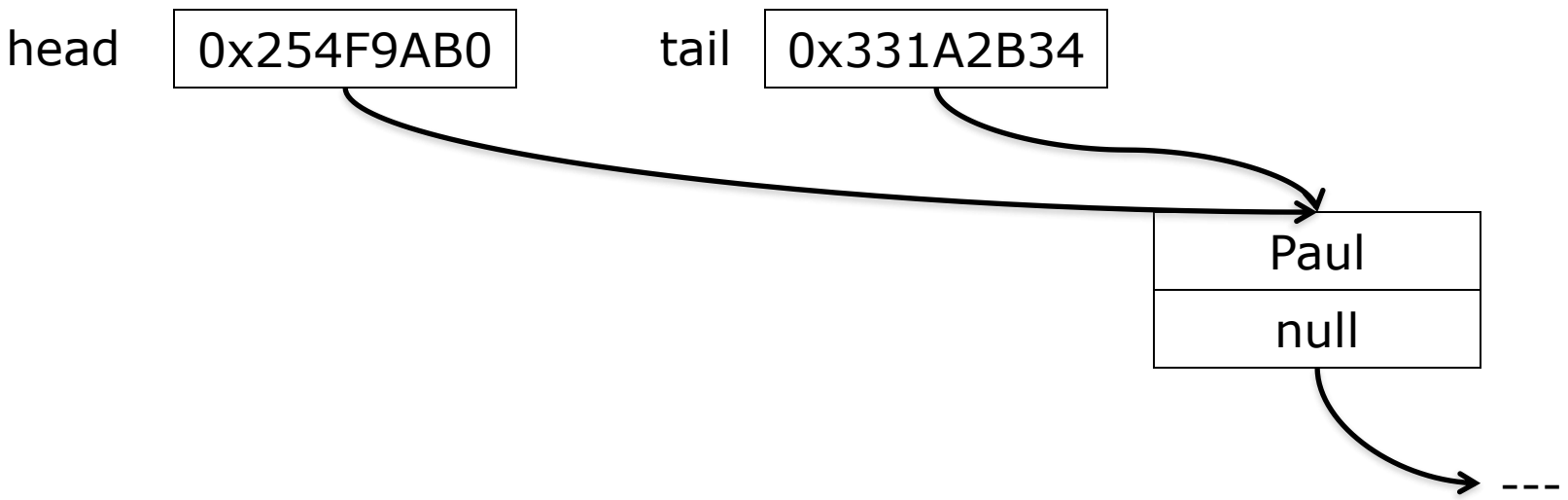


Queues as Linked List

```
if (head == null)
    tail = null;
currentSize--;
```

currentSize 1

f



Queues

Analysis

- The tail pointer does not change in the remove unless there are no elements in the list after the remove is done
- With the double-ended linked list, the queue implementation can have O(1) performance for insert and remove, just like a stack
- Like stacks, search and update are not expected actions, but if they were needed, they would have to perform at O(n)
- Supporting functions like top, size and isEmpty are exactly the same as in a stack and are also O(1) performance
- The linked list implementation is convenient and dynamically manages memory

Stacks and Queues

Summary:

Worst Case	Unsorted Arrays	Unsorted Linked Lists	Stacks and Queues
Search	O(n)	O(n)	O(n) if needed
Insert (Push)	O(1)	O(1)	O(1)
Update	O(n)	O(n)	O(n) if needed
Delete (Pop)	O(n)	O(n)	O(1)
	Static memory usage	Dynamic memory usage	

Priority Queues

The VIP Line

- The network printer example demonstrates queue implementation in a fair manner
- Other situations like a VIP (Very Important Person) line will denote each element with a level of importance for deciding who gets to be removed next
- This situation takes into account a ranking assigned to each element when they enter the queue
- This is not FIFO as the next element that would be removed is the element with highest ranking in the queue
- This modified queue is called a “priority queue”

Priority Queues

Among the Ranks

- The stacks and queues algorithms for insert and remove both perform at O(1)
- This occurs because the order of the list contents does not matter (only the location of where the first or last element entered)
- With a priority queue, the elements now have a property that represents a ranking
- This ranking has to be used when an element is removed
- The question is how to efficiently manage the rankings to get the best insert and remove performances

Priority Queues

O(1) insert

- Part of the reason for the O(1) insert in a linked list or array is because it is unordered
- This allows elements to be inserted quickly without regard of their order or even their position in the list
- As a result, if this O(1) is to continue, the rank order of the queue will not be sorted
- If the ranks in the priority queue are unsorted, then finding the highest ranking element is an O(n) search

Priority Queues

O(1) remove

- In order for the remove to perform at O(1), the highest ranking element must be in a predictable location
- In addition, the next highest ranking element must be in position to be in that same location when the higher element is removed (i.e. next to it)
- This means the ranks must be in sorted order when the remove is called
- Sorting the elements is expensive, so the order will need to be maintained as elements are inserted into the queue
- As we saw with sorted linked lists, the worst case for an insert is O(n)
- So we're stuck with one of these functions performing in linear time. On average, an ordered insert will perform better than an unordered search of an element to delete, so we'll look at an O(1) remove with an O(n) worst case insert

Priority Queues

```
public class RankedNode
{
    public String data;
    public int rank;
    public RankedNode next;
}
```

Priority Queues

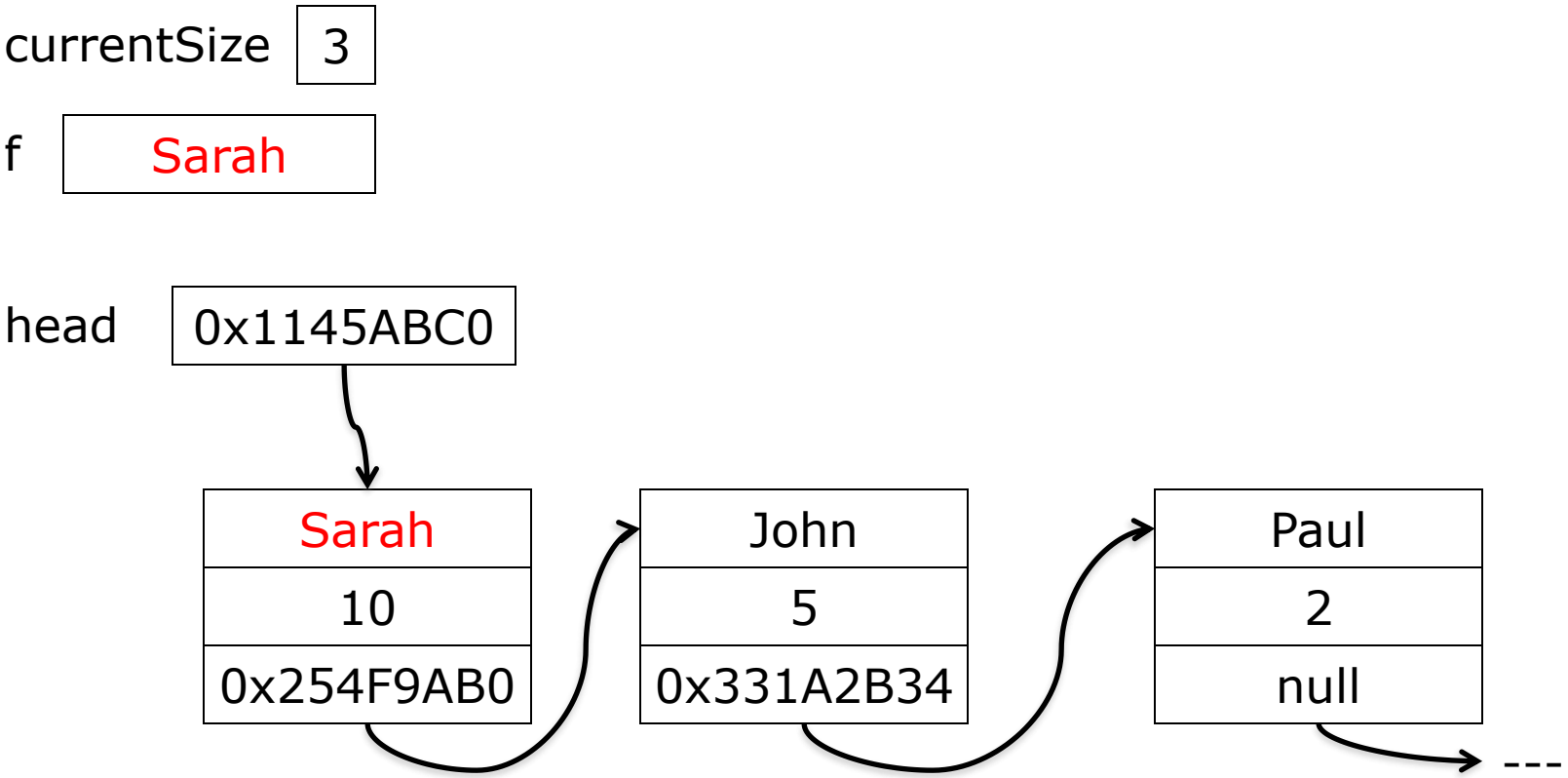
```
public String remove()
{
  if (currentSize == 0)
     return null;

  String f = head.data;
  head = head.next;
  currentSize--;

  return f;
}
```

Priority Queues

```
String f = head.name;
```



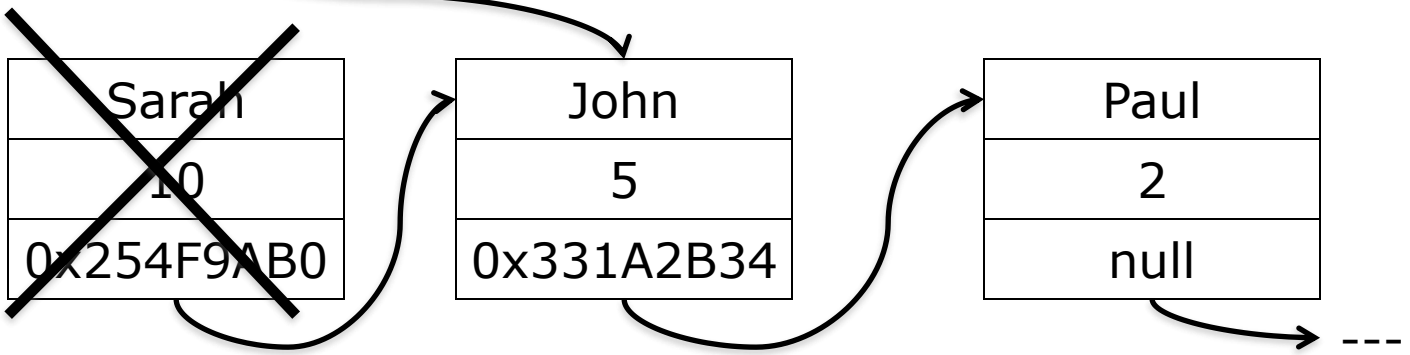
Priority Queues

```
head = head.next;
currentSize--;
```

currentSize | 2

f | Sarah

head | 0x254F9AB0



Priority Queues

Remove analysis

- The remove is a simple Linked List delete at the head
- There is no need to perform any search because the node with the highest rank is assumed to be at the head of the list
- No traversal means no linear progression through the list making this an O(1) operation
- Insert, then, must find the correct spot in the list based on the given rank (which is an additional parameter to the function)

Priority Queues

Insert analysis

- The insert starts with creating a new ranked node
- 3 scenarios are addressed for inserting the new node
  - Inserting the node into an empty queue
  - Inserting the node as the highest rank (i.e. at the head of the list)
  - Inserting the node in the appropriate spot in the list
- The first 2 scenarios are handled on the next slide with no loop required
- The 3rd scenario is a bit more complicated requiring a previous and a current node
- The process is similar to what we used in the standard Linked List delete function
- The diagrams are purposefully left off the presentation here for you to do as practice…

Priority Queues

```
public void insert(String f, int r)
{
   RankedNode temp = new RankedNode();
   temp.data = f;
   temp.rank = r;
   temp.next = null;

   if (currentSize == 0)
       head = temp;
   else if (temp.rank > head.rank)
   {
       temp.next = head;
       head = temp;
   }
```

Priority Queues

```
    else
    {
         RankedNode prev = head;
         RankedNode current = head.next;
         while (current != null)
         {
                 if (temp.rank > current.rank)
                         break;
                 else
                 {
                         prev = prev.next;
                         current = current.next;
                 }
         }
         prev.next = temp;
         temp.next = current;
    }
    currentSize++;
}
```

Priority Queues

Insert analysis

- The 3rd scenario makes the order of the insert perform at O(n) in the worst case
- Worst case scenario is inserting a new node with the lowest rank of all nodes in the queue
- In the reverse situation where O(1) is done on insert, the insert would always take place at the head of the list
- The remove becomes O(n) performing like a standard Linked List delete

Stacks and Queues

Summary:

Worst Case	Stacks and Queues	Priority Queues
Search	O(n) if needed	O(n) if needed
Insert (Push)	O(1)	O(1) or O(n)
Update	O(n) if needed	O(n) if needed
Remove (Pop)	O(1)	O(n) or O(1)
		If Insert is O(1), Remove is O(n) and vice versa

Stacks, Queues and Priority Queues

Summary

- Stacks and queues get constant performance for insert and remove
- Search and update is usually not expected, but if needed would perform at O(n)
- Overall performance is similar to unsorted arrays and linked lists with significant improvement for remove
- Stacks and queues are primed for LIFO and FIFO situations respectively
- Priority queues are a specialized form of queues that perform like unsorted arrays and linked lists
- Priority queues are useful for situations where node rank is a factor
- Maintaining order has clear impacts. What if data order is not maintained and reserved for sorting all data when desired? Next topic we will look at will be the sorting algorithms