The Queue

Sample code

Stack ADT: requirements

- Requirements:
  1) It must be possible to make a stack empty.
  2) It must be possible to add (‘push’) an element to the top of a stack.
  3) It must be possible to remove (‘pop’) the topmost element from a stack.
  4) It must be possible to test whether a stack is empty.
  5) It should be possible to access the topmost element in a stack without removing it.

Queue ADT: requirements

- Requirements:
  1) It must be possible to make a queue empty.
  2) It must be possible to test whether a queue is empty.
  3) It must be possible to obtain the length of a queue.
  4) It must be possible to add an element at the rear of a queue.
  5) It must be possible to remove the front element from a queue.
  6) It must be possible to access the front element in a queue without removing it.

Queue ADT: contract *(1)*

```
public interface Queue {
   // Each Queue object is a queue whose elements are objects.
   /////////////// Accessors ///////////////
   public boolean isEmpty ();
   // Return true if and only if this queue is empty.
   public int size ();
   // Return this queue’s length.
   public Object getFirst ();
   // Return the element at the front of this queue.
```

Queue ADT: contract *(2)*

- Possible contract *(continued)*:

```
///// Transformers ///////////
public void clear ();
// Make this queue empty.
public void addLast (Object elem);
// Add elem as the rear element of this queue.
public Object removeFirst ();
// Remove and return the front element of this queue.
}
```

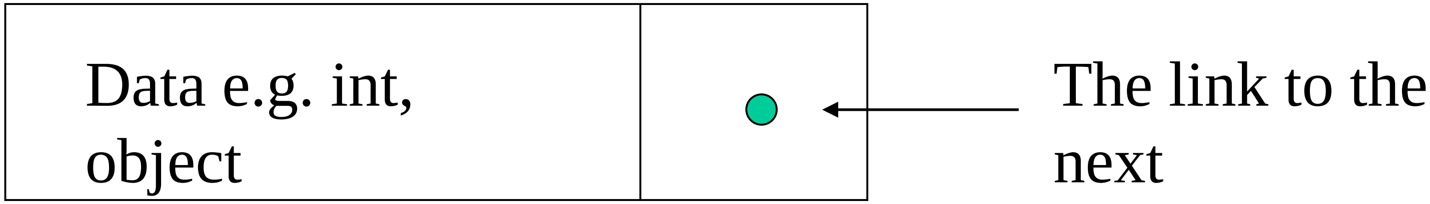
Implementation of queues

- Represent an (unbounded) queue by:
  - a Linked List, whose first node contains the front element, and whose header contains links to the first node (*front*) and last node (*rear*).

  - a variable *length* (optional).

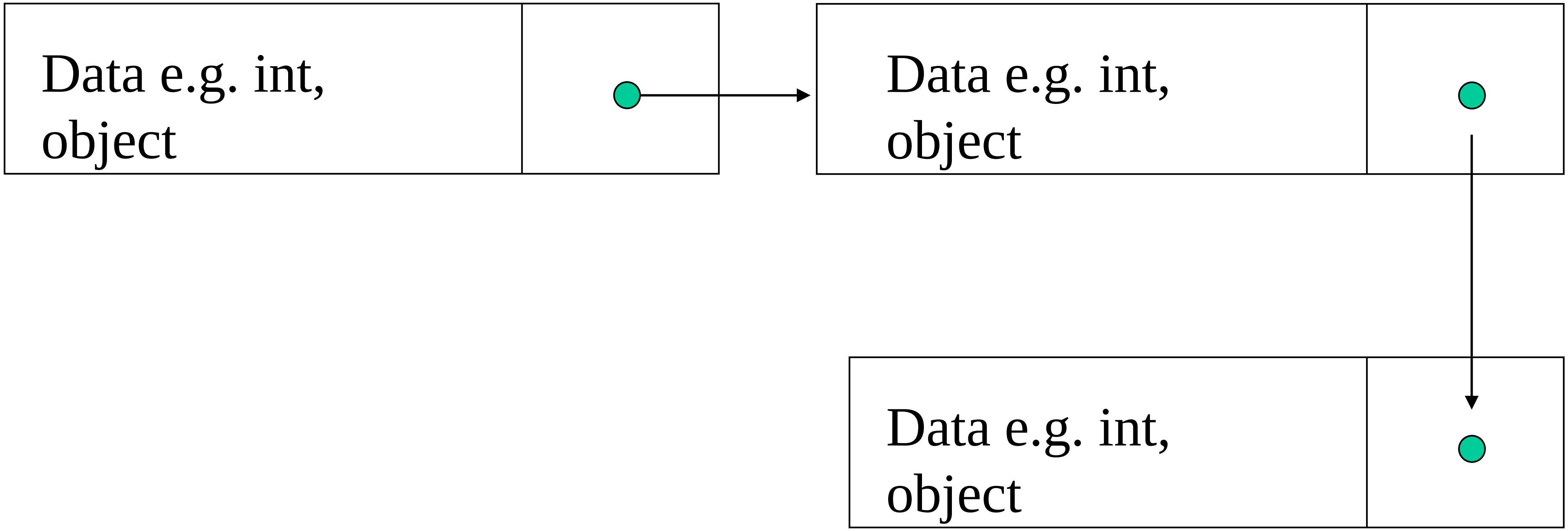
The linked list of nodes

- A linked list links items of data together
- Each node consists of the data and a link to the next node



The link to the next item will contain the address of the next item

With this approach we can build a list of items (data) dynamically, adding and removing when required



Data e.g. int, object

Data e.g. int, object

Data e.g. int, object

Sample application : The Queue

We will look at building a queue data structure using a linked list of nodes.

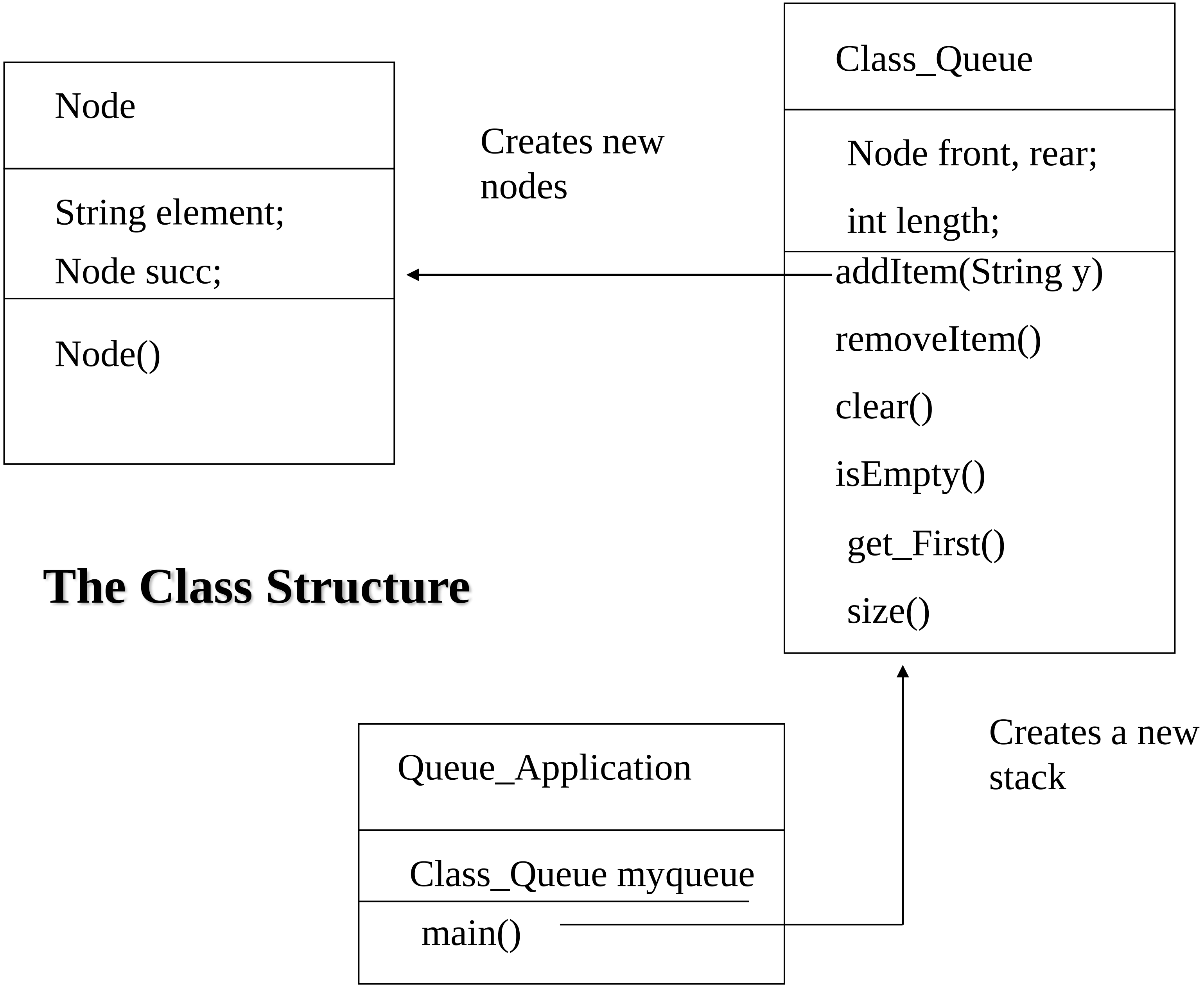
For this we will use three classes:

the class of Node – the data we will store

The class of Queue – the data structure

The application class – the program that will use the Queue class.

The Class Structure



Node

String element;

Node succ;

Node()

Creates new nodes

Class_Queue

Node front, rear;

int length;

addItem(String y)

removeItem()

clear()

isEmpty()

get_First()

size()

Queue_Application

Class_Queue myqueue

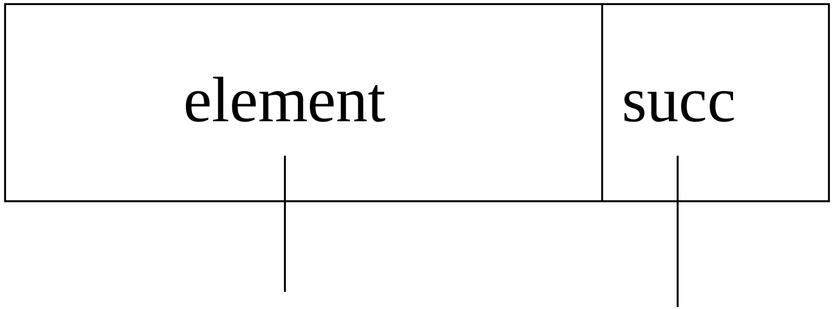
main()

Creates a new stack

NODE CLASS

```
class Node {
        protected String element;   //the data part of the instance object
        protected Node succ;   // the link to the next


public Node (String elem, Node isucc) {                  //the constructor
        element = elem;                                  // set up the values
        succ = isucc;
}
```



element

succ

An int value

The address of the next node

The application

Within the application we will create a new instance of the Class_Queue and use the methods to create a queue structure.

The application programmer need only know the method signatures to use the data type.

```
public class Queue_Application  {
      public static void main (String args[]) {
          int x;
          Class_Queue thisqueue = new Class_Queue();
             int fin = 0;
              while (fin!=1) {
                    System.out.println("1. Add an element ");
                   System.out.println("2. Remove an element ");
                   System.out.println("3. Clear the queue ");
                    System.out.println("4. List all elements ");
                   System.out.println("5. Exit ");
```

```
console.writeline(" ");
console.writeline("Please enter your choice 1-5");
x = int.Parse(Console.ReadLine());
```

```
switch (x)
      {  case 1:
                  { System.out.println ("enter an element to be added");
                              String y = console.readline();
                              thisqueue.addItem(y);
                                                      break;}
```

```
case 2: {   if (thisqueue.isEmpty())
                    console.writeln ("the queue is empty");
               else
                    String x = thisqueue.removeFirst();
                     //output value
                    break;
           }                                                              }
```

```
case 3: {
        thisqueue.clear();
        break;}

case 4: {
        thisqueue.traverse();
        break;}
case 5:{  fin = 1;
         break;}

    }}}
```

The data type: Queue

Single Linked List

The data type programmer must create
the data type and implement the methods.

```
class Class_Queue  {

               Node front, rear;
              private int length;

              /////////////// Constructor ///////////////
 public Class_Queue () {
                          front = rear = null;
                          length = 0;
                          }
```

The methods of our Queue class:

addItem(String elem)

removeItem()

clear()

isEmpty()

get_First()

size()

```
public void addItem (String elem) {
Node newest = new Node(elem, null);
if (rear != null) {
            rear.succ = newest;
            rear = newest;
            length++;        }
else
            {
               front = newest;
                rear = newest;
                length++;}  }
```

```
/////////////// Accessors //////////////

    public boolean isEmpty ()

        {return (length = = 0);}

        public int size () {

            return length;
            }
```

```
public String getFirst () {
            return front.element;}
```

```
public void clear () {
        front = rear = null;
        length = 0;}
```

```
public String removeFirst () {
        String frontElem = front.element;
        front = front.succ;
        if (front == null)
                rear = null;
        length--;
        return frontElem;}
```

```
Public void traverse() {
for (Node curr = front;curr != null;curr = curr.succ)

                    console.writeln (curr.element);

                    }
}
```

End of the class

The interaction

We will now look at:

- Create the new queue
- Add an item
- Remove an item
- Clear the list

Create the new queue

In the application we have:

```
Class_Queue thisqueue = new Class_Queue();
```

This will create the instance object with new data:

```
Node front, rear;
private int length;
```

The constructor method will execute:

```
public Class_Queue () {
                    front = rear = null;
                    length = 0;
                    }
```

Here we have not created the new instance of the node objects *(note: we have not used the new keyword)*

But we have created links to the objects

front

rear

At this point front and rear have **null** values; they do not point to an actual object i.e. they do not yet contain an address of the next node.

Add an item

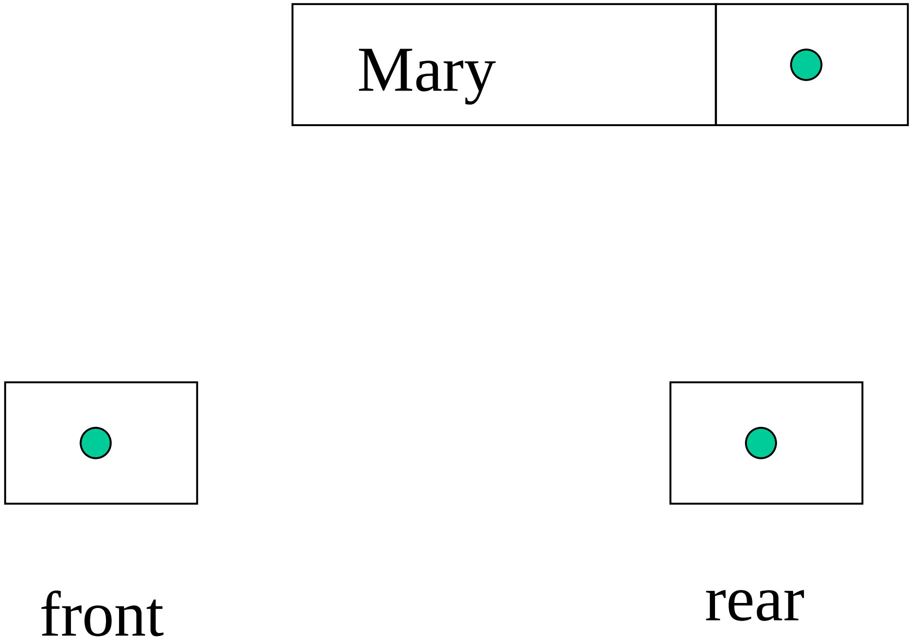
If the user selects to add an item:

```
{  case 1:
                    { System.out.println ("enter an element to be added");
                              String y = EasyIn.getString()
                              thisqueue.addItem(y);
                                                  break;}
```

The additem method is called and the parameter is passed.

```
public void addItem (String elem) {
Node newest = new Node(elem, null);
if (rear != null) {
                  rear.succ = newest;
                  rear = newest;
                  length++;        }
                  else
                  {
                    front = newest;
                     rear = newest;
                  length++;}  }
```

```
Node newest = new Node(elem, null);
```



if (rear != null) {

~~rear.succ = newest;~~

~~rear = newest;~~

~~length++;~~ }

Rear does = null

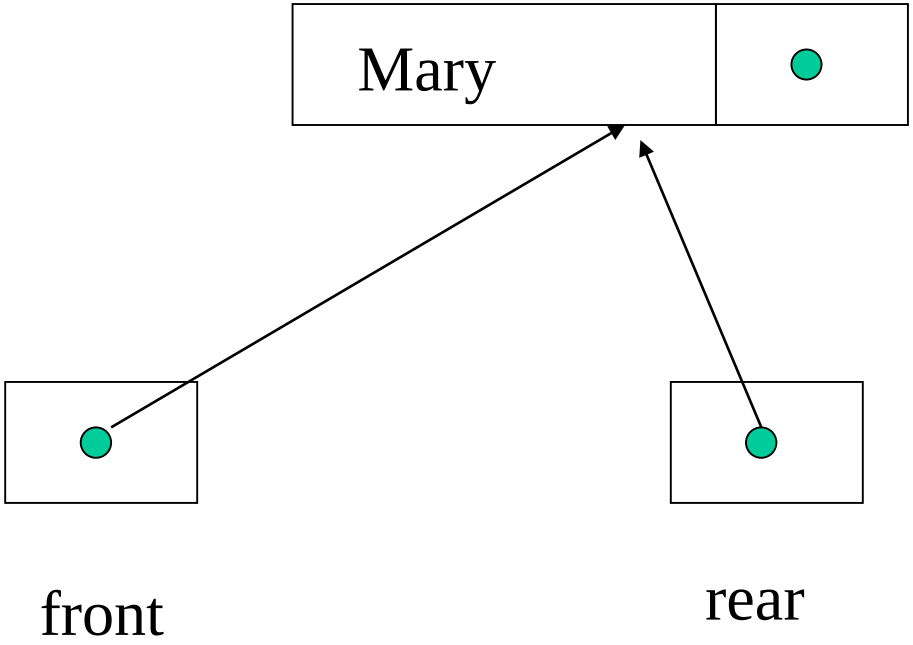
else

{

front = newest;

rear = newest;

length++;} }

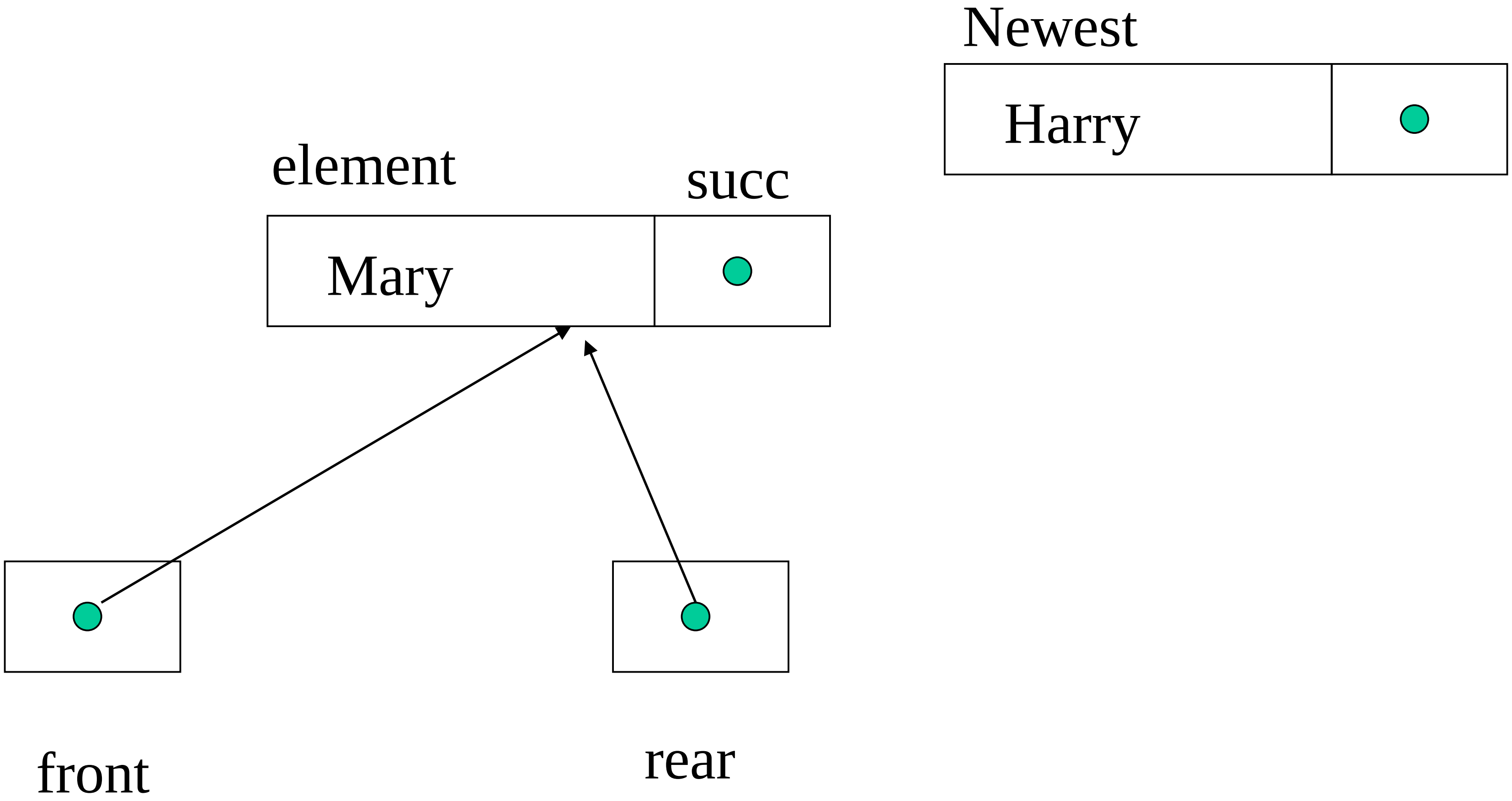

Mary
front
rear

If we add another item

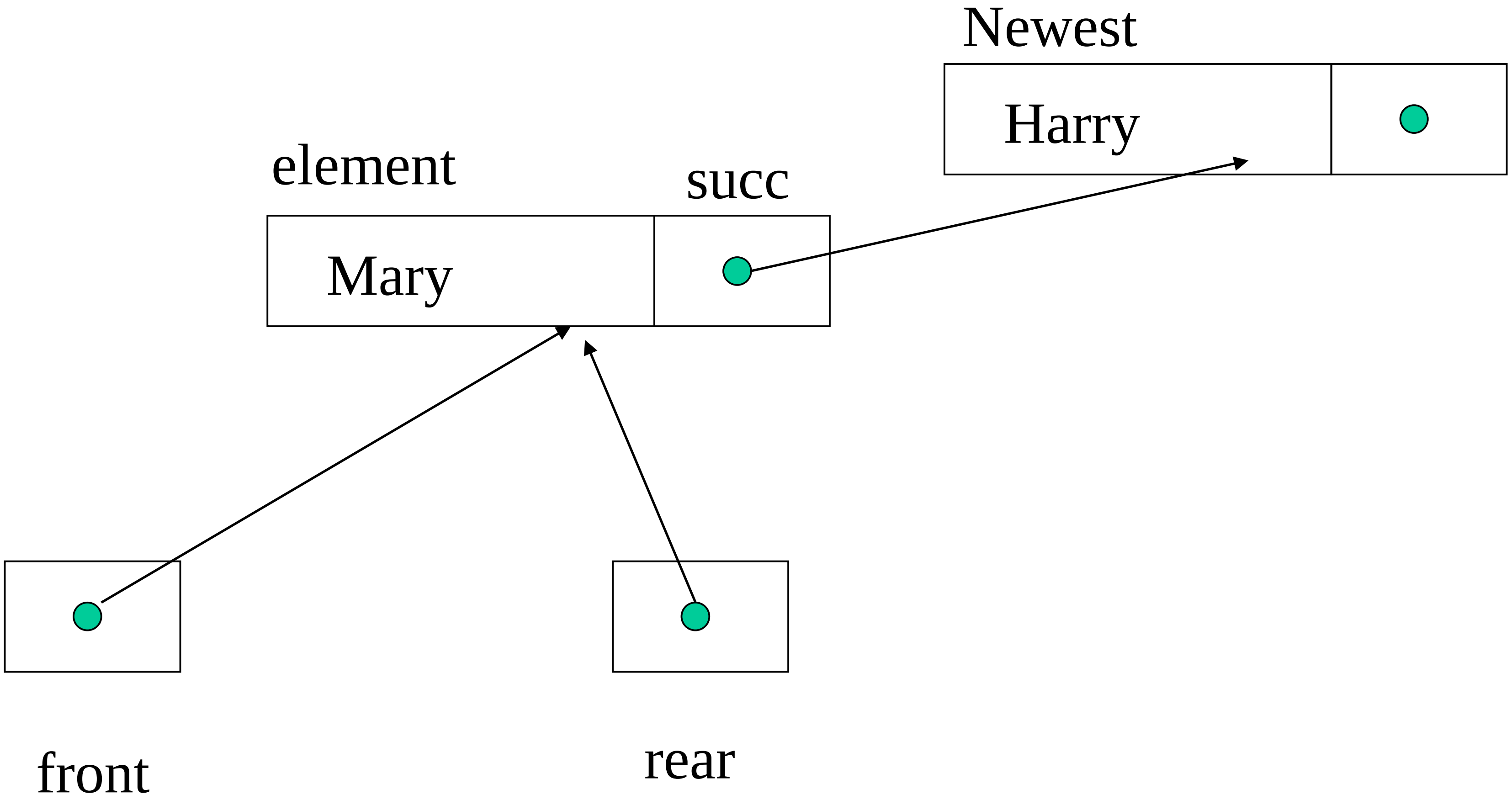
Node newest = new Node(elem, null);


Harry

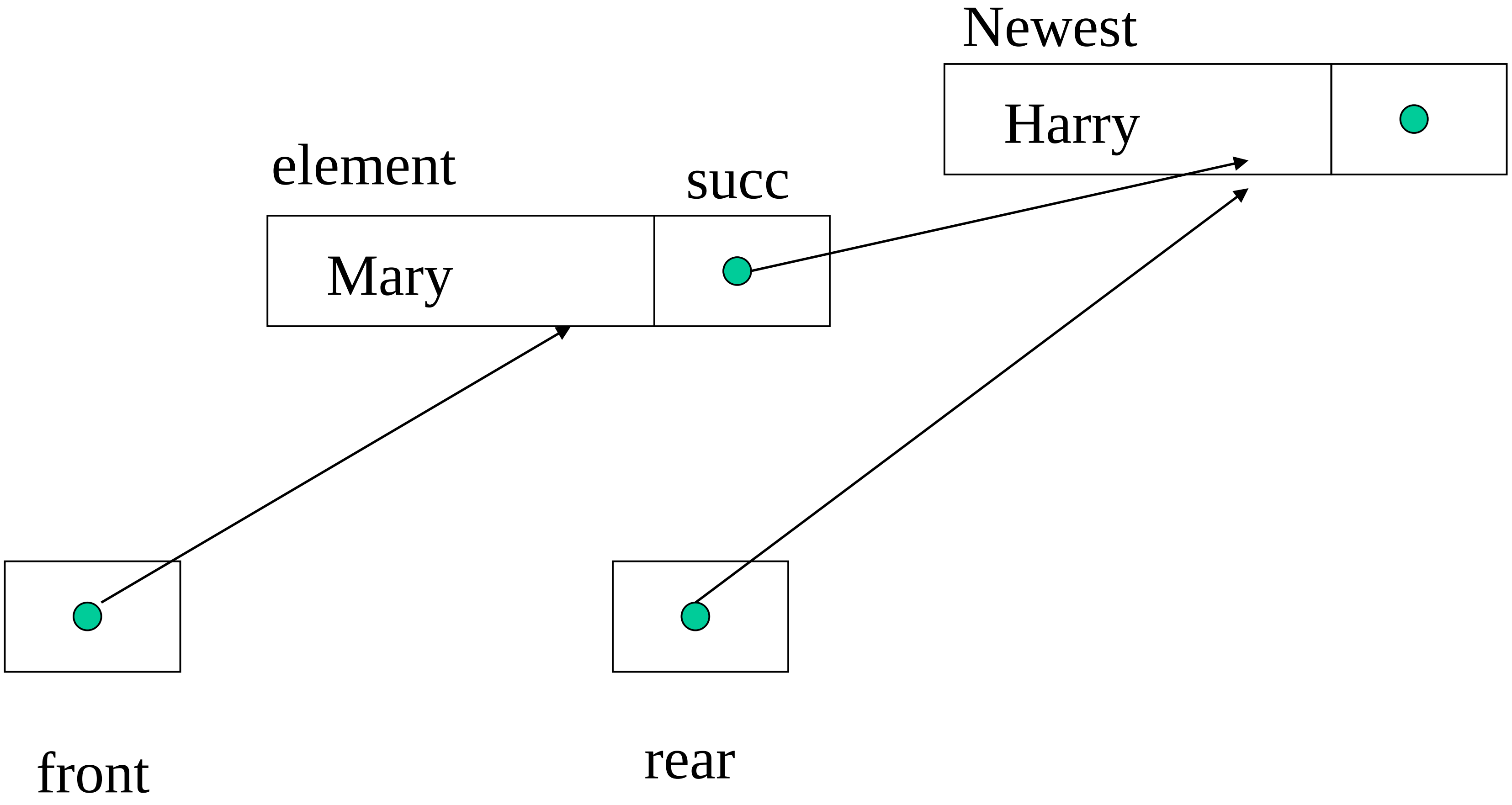
```
if (rear != null) {
                    rear.succ = newest;
                    rear = newest;
                    length++;      }
```



```
if (rear != null) {
                    rear.succ = newest;
                    rear = newest;
                    length++;        }
```

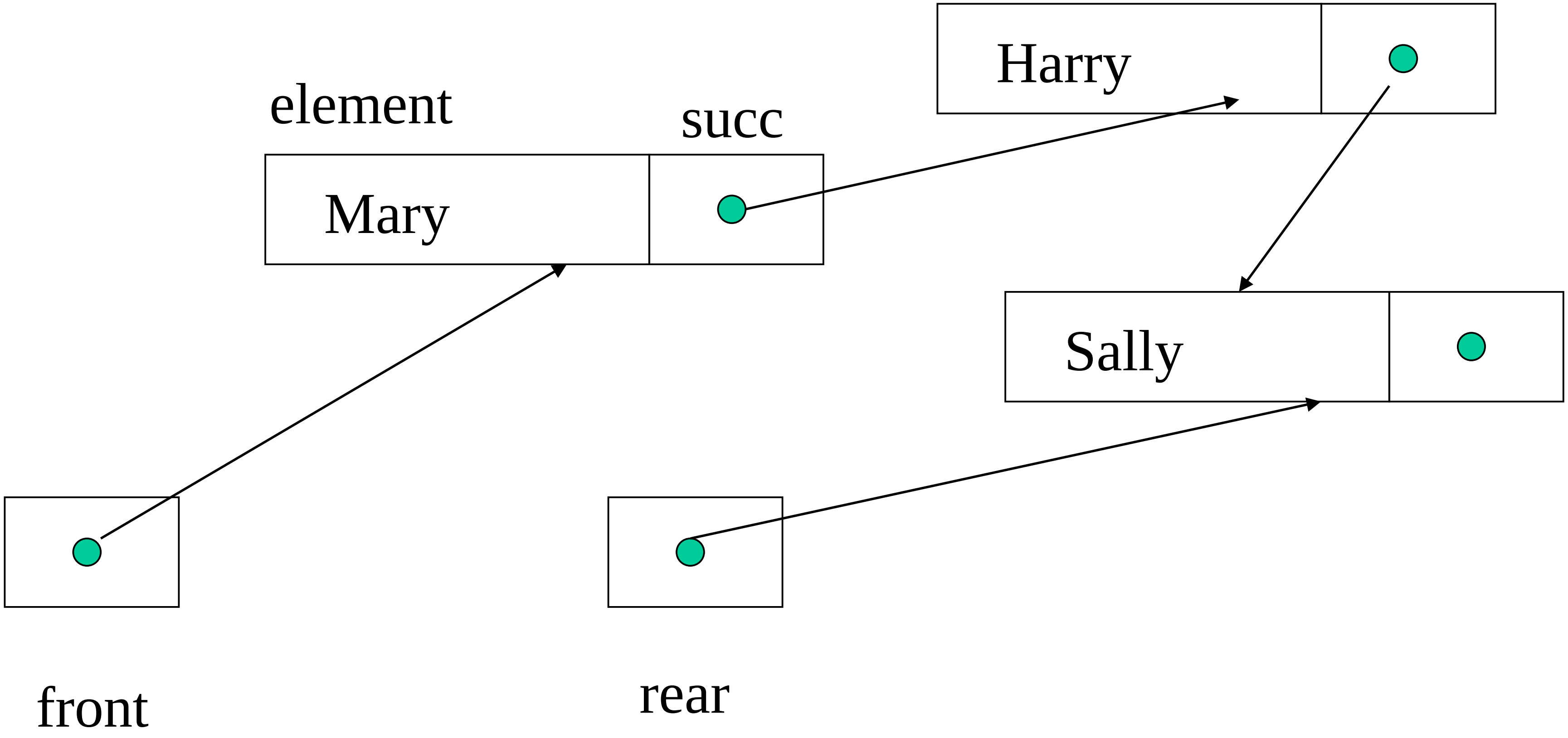


```
if (rear != null) {
                    rear.succ = newest;
                    rear = newest;
                    length++;        }
```



If we add another:

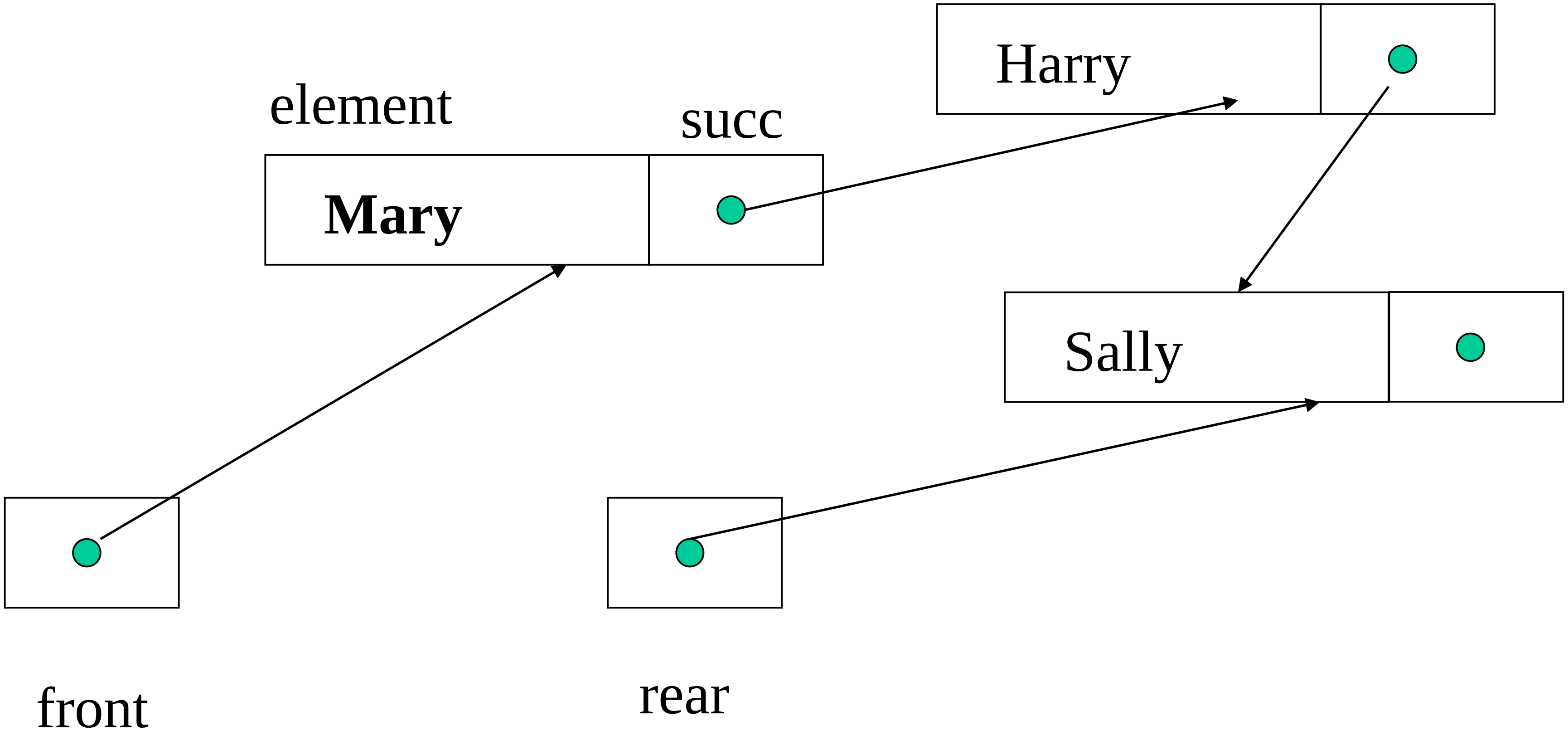
```
rear.succ = newest;
rear = newest;
```



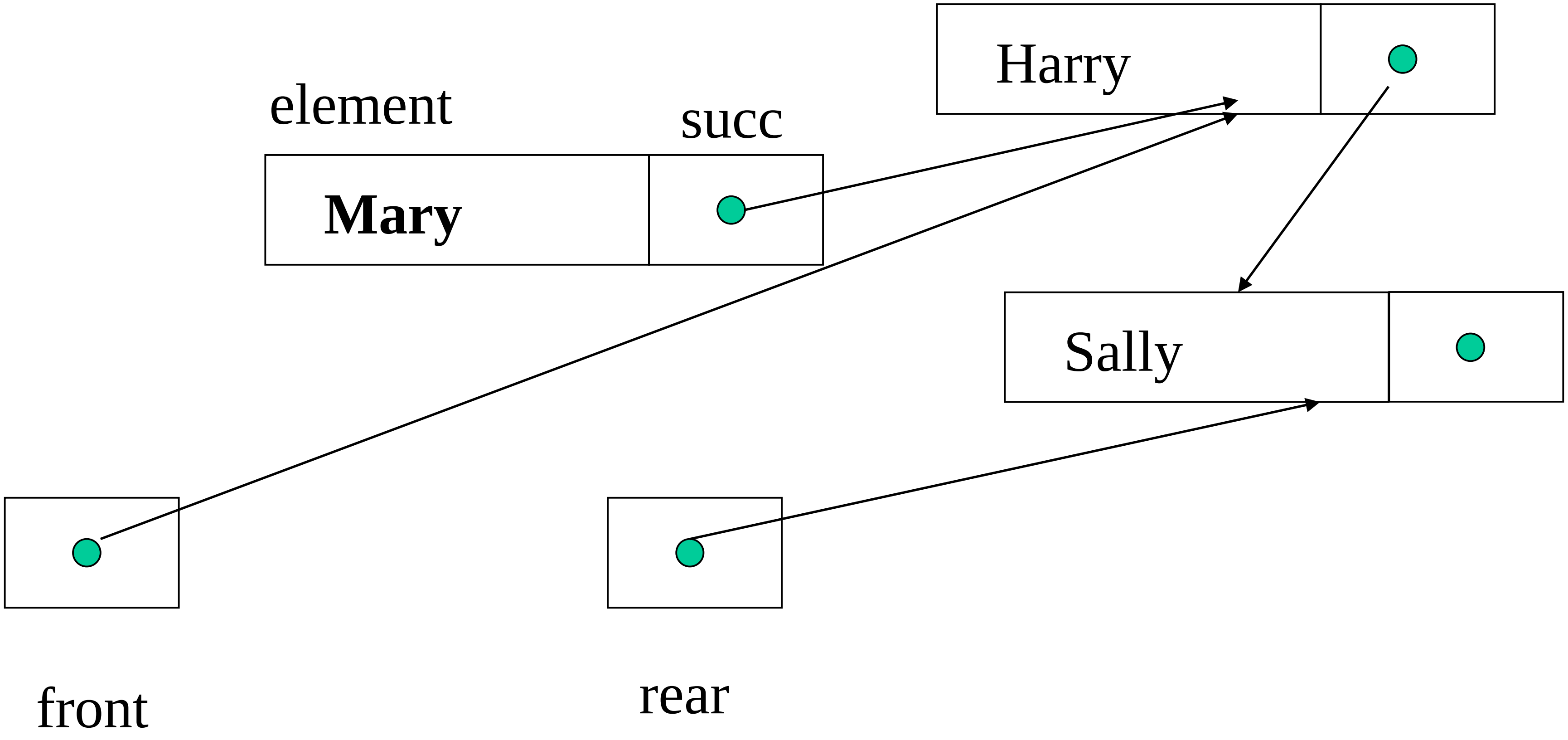
Remove an Item

```
case 2: {   if (thisqueue.isEmpty())
                    System.out.println ("the queue is empty");
            else{
                   String x = thisqueue.removeFirst();
                    //handle value
                    }
                   break;
        }
```

```
public String removeFirst () {
          String frontElem = front.element;
          front = front.succ;
          if (front == null)
          rear = null;
          length--;
          return frontElem;}
```

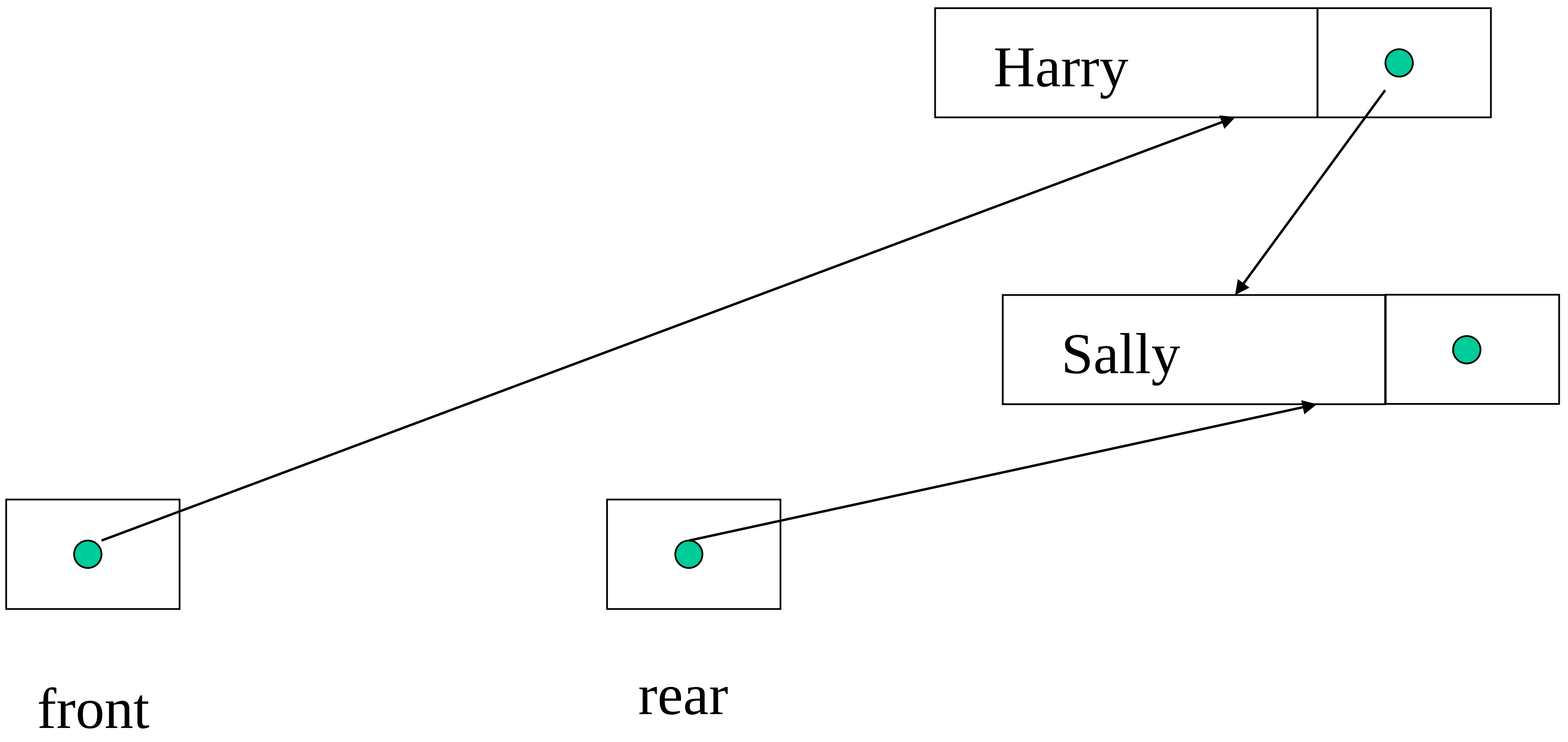


String frontElem = front.element;



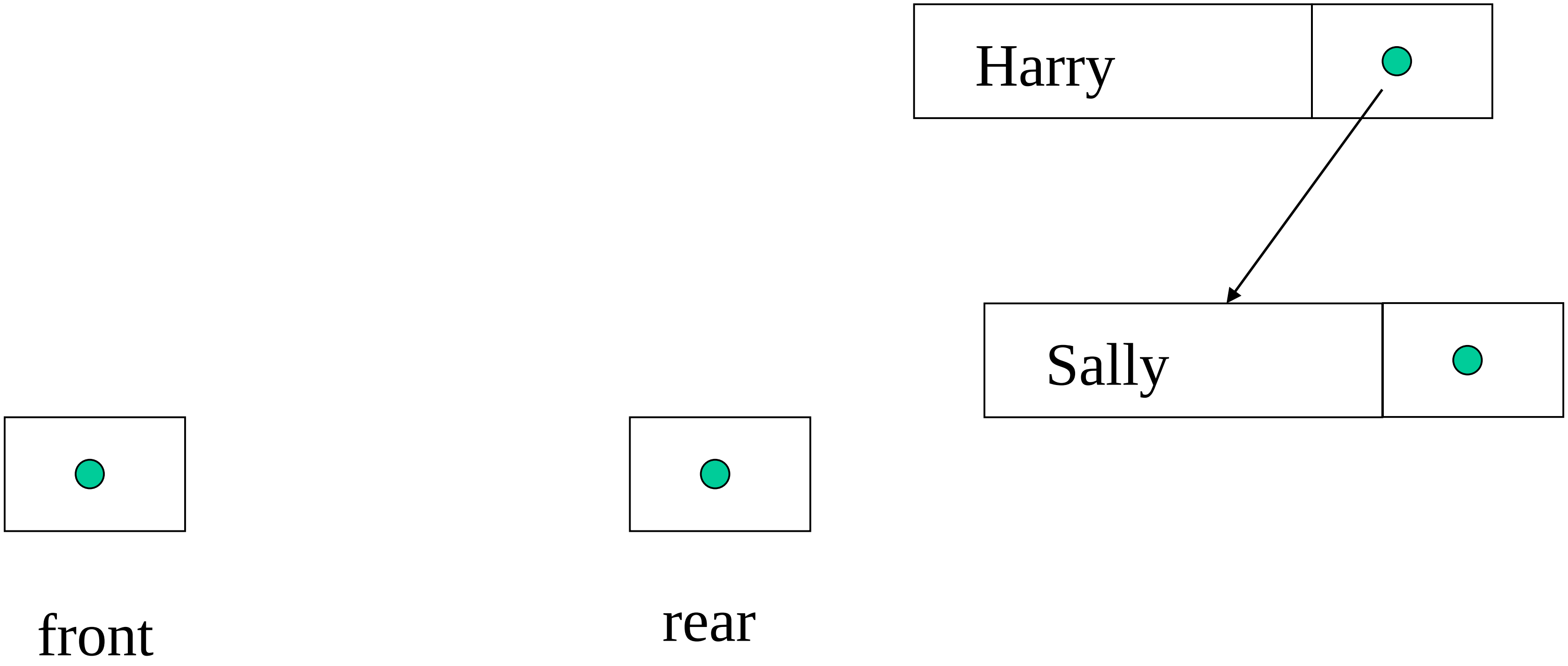
front = front.succ;

With nothing pointing to it the node will be removed automatically

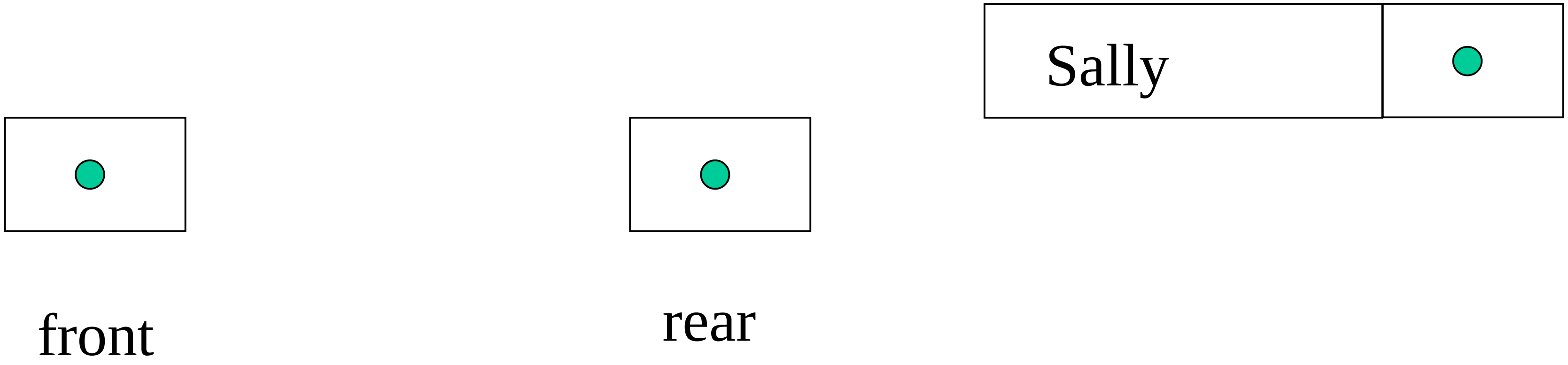

Harry
Sally
front
rear

Clear the list

```
public void clear () {
        front = rear = null;
        length = 0;}
```



With nothing pointing to the node containing Harry it will be removed



Sally

front

rear

Subsequently nothing will point to the node containing Sally and it will be removed

front

rear

Stack sample code

G.Affleck

Stack ADT: requirements

- Requirements:
  1) It must be possible to make a stack empty.
  2) It must be possible to add ('push') an element to the top of a stack.
  3) It must be possible to remove ('pop') the topmost element from a stack.
  4) It must be possible to test whether a stack is empty.
  5) It should be possible to access the topmost element in a stack without removing it.

Stack ADT: contract *(1)*

```
public interface Stack {
    //  Each Stack object is a stack whose
elements are objects.
    ///////////////// Accessors
/////////////////
   public boolean isEmpty ();
    //  Return true if and only if this stack is
empty.
   public Object getLast ();
    //  Return the element at the top of this stack.
```

Stack ADT: contract *(2)*

- Possible contract *(continued)*:

```
   //// Transformers ////
   public void clear ();
   // Make this stack empty.
   public void addLast (Object elem);
   // Add elem as the top element of this stack.
   public Object removeLast ();
   // Remove and return the element at the top of this stack.
}
```

```
        class SLL  {
//Construct an empty SLL

                SLLNode front;
               private int length;

/////////////// Constructor ///////////////
                       public SLL () {
                       front = null;
                       length = 0;
                       }
```

```
/////////////// Accessors ///////////////

     public boolean isEmpty ()

          {return (length == 0);}

          public int size () {

               return length;
               }
```

```
    public Object getFirst () {

    System.out.println (front.element);

        return front.element;}

public void clear () {

    front = null;
    length = 0;}
```

```
public void addFirst (Object elem) {
SLLNode newest = new SLLNode(elem, null);
        newest.succ = front;
        front = newest;
        length++;       }
```

```
public Object removeFirst () {
        Object frontElem = front.element;
        front = front.succ;
        console.writeln(frontElem);
        console.writeln("has been removed");
        length--;
        return frontElem;}
```

```
public void traverse() {
  for (SLLNode curr = front;curr != null;curr = curr.succ)
  console.writeln (curr.element);

                                  }
}  ///////////// end of class
```

```
class SLLNode {
//Each SLLNode object is an SLL node
//This node consists of an element (element) and a link to its successor (succ)
protected Object element;
protected SLLNode succ;
public SLLNode (Object elem, SLLNode isucc) {
//Construct an SLL node with element elem and successor succ
element = elem;
succ = isucc;
}
}
```

```
public class stacked {
public static void main (String args[]) {
        int fin = 0;
        console.writeln ("stack");
        SLL     Thisstack = new SLL();
```

```
while (fin !=1) {
 console.writeln("1. Add an element ");
 console.writeln("2. Remove an element ");
 console.writeln("3. Clear the stack ");
 console.writeln("4. List all elements ");
 console.writeln("5. Exit ");
 console.writeln(" ");
 console.writeln("Please enter your choice 1-5");
```

```
int x = int.Parse(Console.ReadLine());

switch (x)

{ case 1:

     console.writeln ("enter an element to be added");

    String y = console.readline();

    Thisstack.addFirst(y);

          break;}
```

```
case 2: {
        if (Thisstack.isEmpty())
        console.writeln ("the queue is empty");
        else
         Thisstack.removeFirst();
           break;
        }
```

```
case 3: {
            Thisstack.clear();
           break;
           }

case 4: {
            Thisstack.traverse();
           break;
           }
```

```
case 5:{
        fin = 1;
        break;
        }
        }}}}
```

Linked list In Javascript

```
function Node(data) {
        this.data = data;
        this.next = null;
}

function SinglyList() {
        this._length = 0;
        this.head = null;
}
```

```
SinglyList.prototype.add = function(value) {
    var node = new Node(value),
        currentNode = this.head;

    // an empty list
    if (!currentNode) {
        this.head = node;
        this._length++;

        return node;
    }
```

```
    // a non-empty list
    while (currentNode.next) {
        currentNode = currentNode.next;
    }

    currentNode.next = node;

    this._length++;

    return node;
};
```

```
SinglyList.prototype.searchNodeAt = function(position) {
    var currentNode = this.head,
        length = this._length,
        count = 1,
        message = {failure: 'Failure: non-existent node in this list.'};

    // 1st use-case: an invalid position
    if (length === 0 || position < 1 || position > length) {
        throw new Error(message.failure);
    }
```

```
    // 2nd use-case: a valid position
    while (count < position) {
        currentNode = currentNode.next;
        count++;
    }

    return currentNode;
};
```

```
SinglyList.prototype.remove = function(position) {
    var currentNode = this.head,
        length = this._length,
        count = 0,
        message = {failure: 'Failure: non-existent node in this list.'},
        beforeNodeToDelete = null,
        nodeToDelete = null,
        deletedNode = null;

    // 1st use-case: an invalid position
    if (position < 0 || position > length) {
        throw new Error(message.failure);
    }
```

```
// 2nd use-case: the first node is removed
if (position === 1) {
    this.head = currentNode.next;
    deletedNode = currentNode;
    currentNode = null;
    this._length--;

    return deletedNode;
}
```

```
        // 3rd use-case: any other node is removed
        while (count < position) {
                beforeNodeToDelete = currentNode;
                nodeToDelete = currentNode.next;
                count++;
        }

        beforeNodeToDelete.next = nodeToDelete.next;
        deletedNode = nodeToDelete;
        nodeToDelete = null;
        this._length--;

        return deletedNode;
};
```