ArrayList Stack Queue

Juan Zhai

juan.zhai@rutgers.edu

java.util.ArrayList

- *ArrayList* is a generic array that can resize itself *automatically* on demand → dynamic length
- capacity of *ArrayList*: the number of array locations for which memory space has been set aside.
- size of ArrayList: the current number of elements in the list. → size ≤ *capacity*


```
ArrayList<String> al = new ArrayList<String>(5);
for(int i=0; i<5; i++)
    al.add(new String(i));
//automatic resizing when one more item is added
al.add(new String(5))
```

java.util.ArrayList

- When reach the full capacity:
  - Allocate a new array with a bigger capacity
  - Copy items in the old array to the new array
    - Worst case for adding an element at the end:
      → resize the array, O(n)
  - Append the new item to the new array
  - The original array is ready for garbage collection

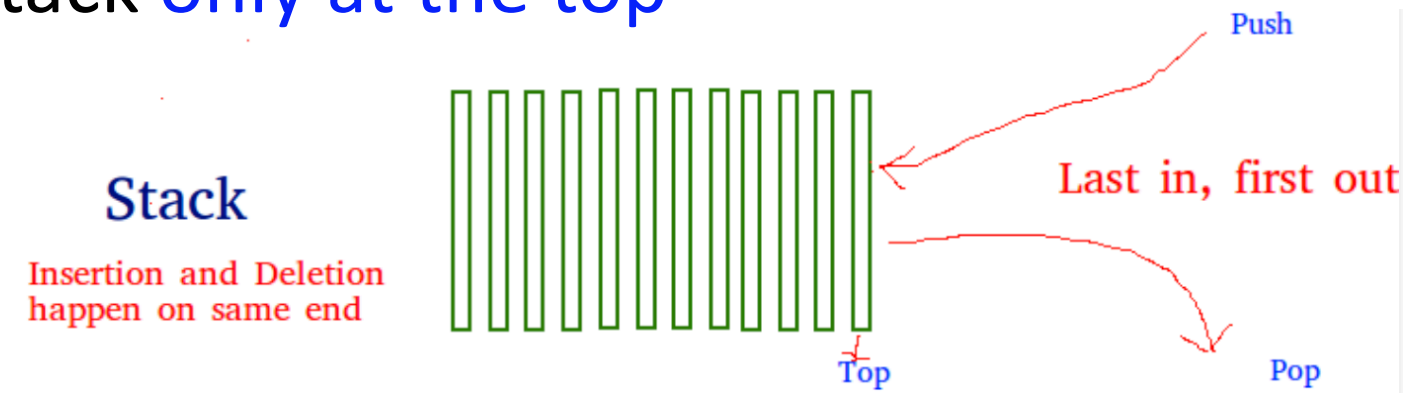
- No guarantee that there is enough space for extending array in the original space

java.util.ArrayList

- *ArrayList* provides numerous methods
  - *boolean add (E e):* appends e to the end. → O(n)
  - *E remove(int index):* removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices). → O(n)
  - *E get(int index):* returns the element at the specified position in this list. → O(1)
  - https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

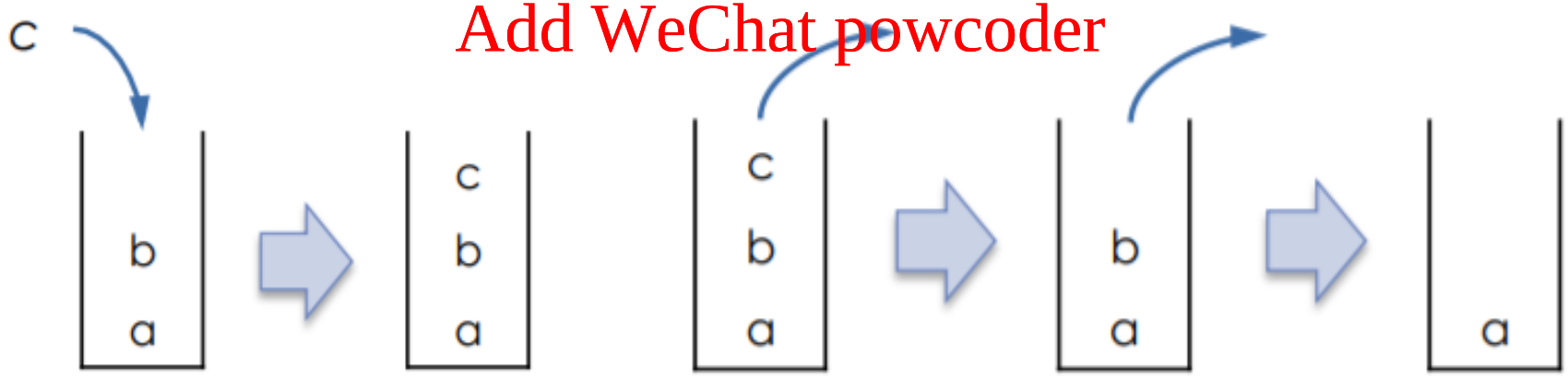
Stack

- *Stack* is a linear data structure which is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle
- Elements are added and removed from the stack only at the top



Stack

- Two fundamental operations
  - push: adds an item to the top of the stack
  - pop: removes and returns the item from the top



Stack operations

- void push(T item)
- T pop(): returns the top entry and removes it
- T peek(): returns the top entry without removing it

If stack does not provide peek operation, how to get the top element?

→ First pop the top entry and then push the entry back

- Utility methods to improve efficiency
  - int size()
  - boolean isEmpty()
  - void clear()

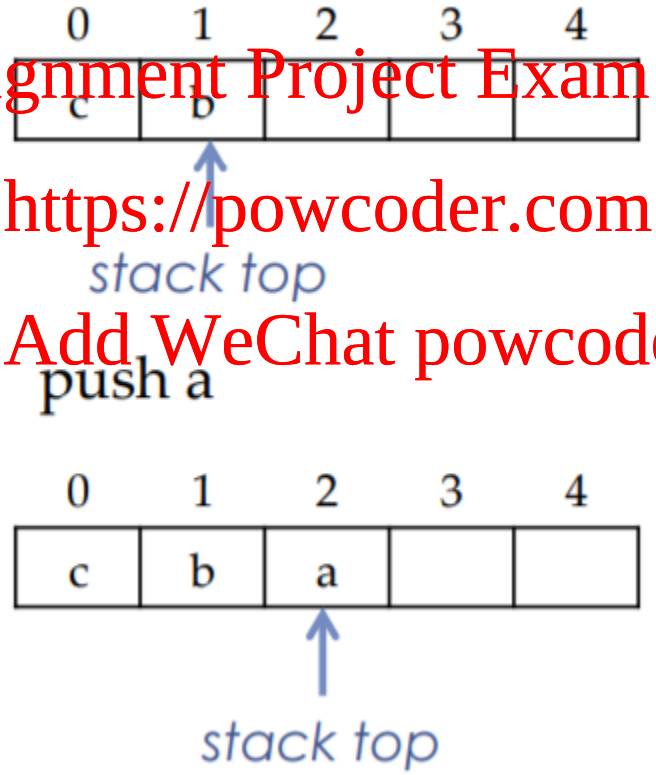
```java
public class Stack<T> {
    private Node<T> front;
    private int size;

    public Stack() {
        front = null;
        size = 0;
    }

    public void push(T item) {
        front = new Node<T>(item, front);
        size++;
    }

    public T pop() throws NoSuchElementException {
        if (front == null) {
            throw new NoSuchElementException();
        }
        T temp = front.data;
        front = front.next;
        size--;
        return temp;
    }

    public boolean isEmpty() {
        return front == null;
    }

    public int size() {
        return size;
    }

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

Implement a stack from scratch

Sakai Code

Stack Implementation Using ArrayList

- The last entry is used as the top



```java
import java.util.ArrayList; //import the class in order to use it
import java.util.NoSuchElementException;

public class StackUsingArrayList<T> {

    private ArrayList<T> list = new ArrayList<T>();

    public void push(T v) {
        list.add(v);
    }

    public T peek() {
        return list.get(list.size()-1);
    }

    public T pop() throws NoSuchElementException{
        if(list.size()==0)
            throw new NoSuchElementException();
        return list.remove(list.size()-1);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int size() {
        return list.size();
    }

    public static void main(String[] args) {
        StackUsingArrayList<String> stack = new StackUsingArrayList<String>();
        stack.push("!");
        stack.push("CS112");
        stack.push(" ");
        stack.push("to");
        stack.push(" ");
        stack.push("Welcome");
        while (!stack.isEmpty()) {
            System.out.print(stack.pop()); //Welcome to CS112
        }
    }
}
```

Reuse ArrayList

Sakai Code

Stack Implementation Using Linked List

- The front node is used as the top
- Use insertFront to push and deleteFront to pop

push a

Reuse LinkedList

```
public class StackUsingLinkedList<T> {

    private LinkedList<T> list = new LinkedList<T>();

    public void push(T v) {
        list.addFront(v);
    }

    public T peek() {
        return list.getFront();
    }

    public T pop() {
        return list.deleteFront();
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public static void main(String[] args) {
        StackUsingLinkedList<String> stack = new StackUsingLinkedList<String>();
        stack.push("CS112");
        stack.push(" ");
        stack.push("to");
        stack.push(" ");
        stack.push("Welcome");
        while (!stack.isEmpty()) {
            System.out.print(stack.pop());  //Welcome to CS112
        }
    }
}
```

```
public T deleteFront() throws NoSuchElementException {
    if (front == null) {
        // throw new NoSuchElementException();
        throw new NoSuchElementException("Can't delete from an empty list");
    }
    T tmp = front.data;
    front = front.next;
    size--;
    return tmp;
}
```

deleteFront takes care of whether there exists an item or not

Sakai Code

Worst Case Running time

Operation	ArrayList	Linked List
push	O(n)*	O(1)
pop	O(1)	O(1)
peek	O(1)	O(1)
isEmpty	O(1)	O(1)
size	O(1)	O(1)

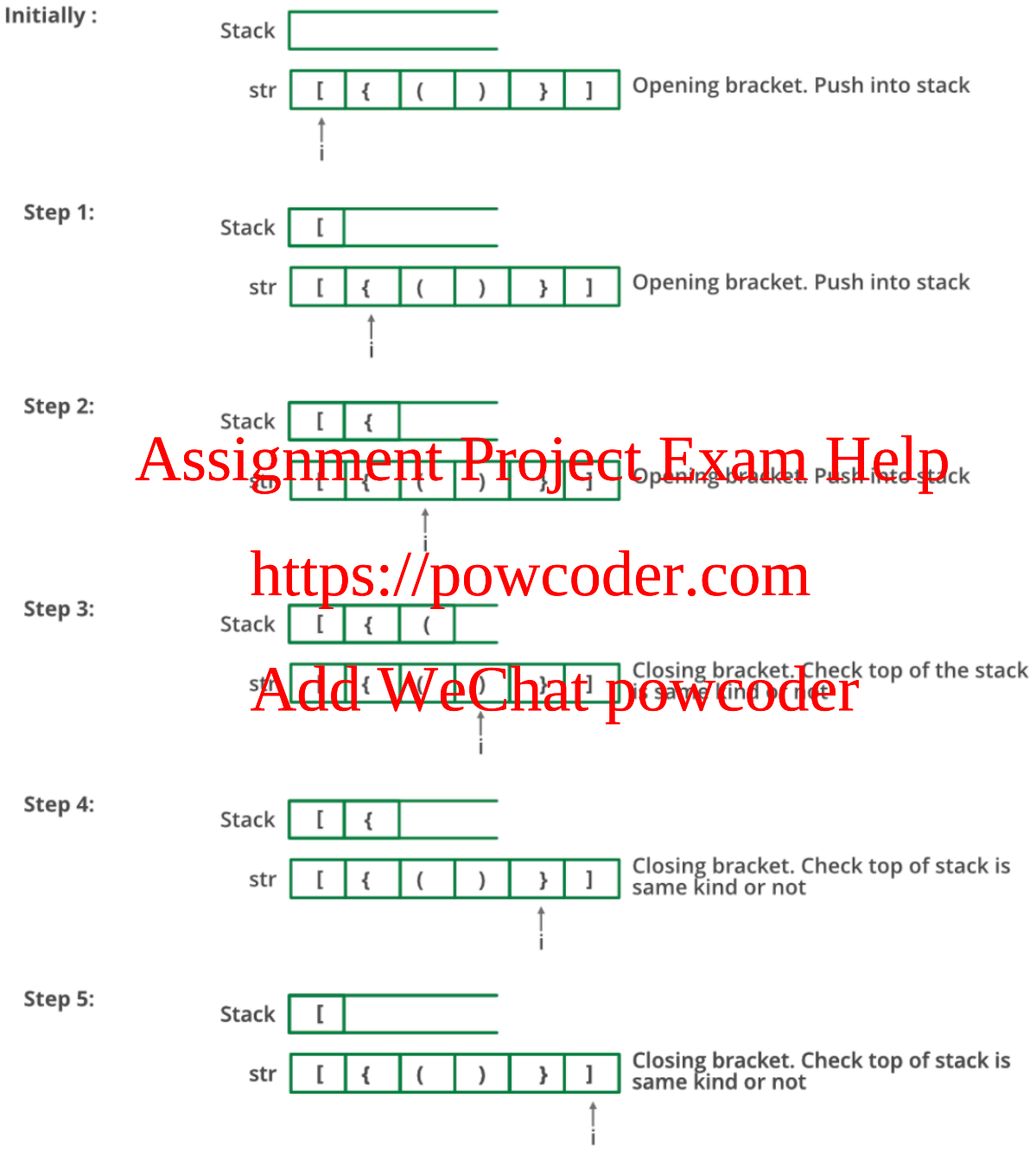
*copy items in old array to new array when capacity is full

Applications of Stack

- Reverse a word: push a given word to stack - letter by letter - and then pop letters from the stack
- "undo" mechanism in text editors: a chain of undos erases actions in the sequence latest to earliest
- Balanced parentheses: each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested

Balanced parentheses

- A compiler checks expressions in a program to ensure that all parentheses are matched
  - (a+b) ✓
  - (a+b.)a+b(, ✗
  - (a+b)*c-d*(c+a[2])/(x+y)) ✗
- If you see an opening parenthesis, push it into the stack
- If you see a closing parenthesis, the top element in the stack must be an opening parenthesis

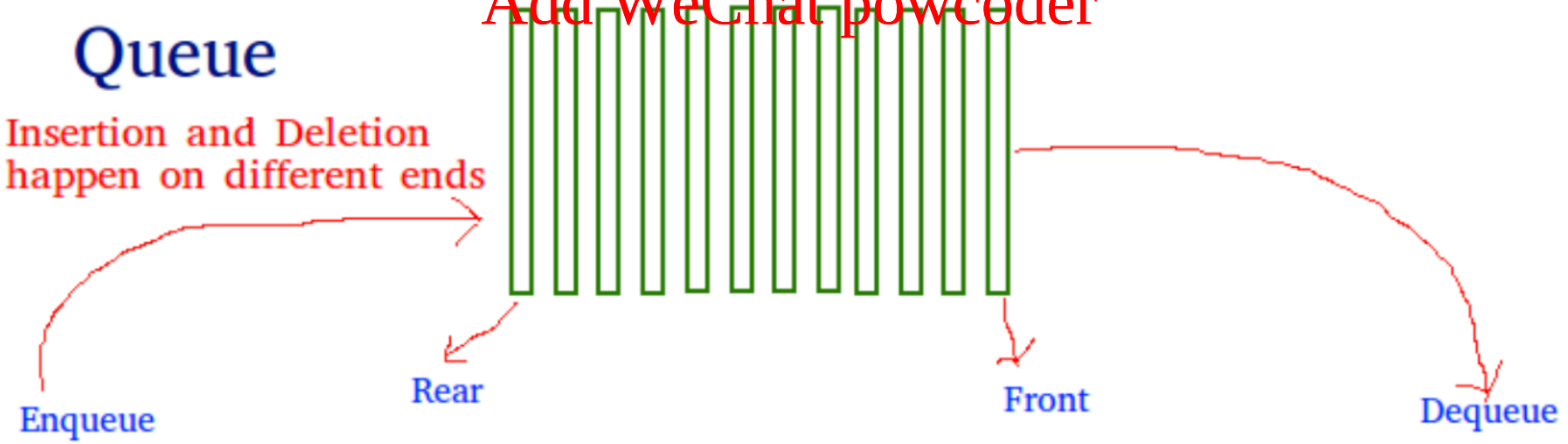


```java
public static boolean isParenthesisMatch(String str) {
    Stack<Character> stack = new Stack<Character>();
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        // If c is a opening parenthesis then push it
        if (c == '{' || c == '(' || c == '[')
            stack.push(c);
        /*
         * If c is a closing parenthesis, pop a parenthesis from stack and check if the
         * popped parenthesis and c is a matched pair
         */
        if (c == '}' || c == ')' || c == ']') {
            // No opening parenthesis exists for the current closing parenthesis
            if (stack.isEmpty())
                return false;
            // pop a parenthesis and compare
            char c2 = stack.pop();
            if (!isMatchingPair(c2, c))
                return false;
        }
    }
    /*
     * If there is something left in stack, then there is a starting parenthesis
     * without a closing parenthesis
     */
    if (stack.isEmpty())
        return true;
    else
        return false;
}

private static boolean isMatchingPair(char character1, char character2) {
    if (character1 == '(' && character2 == ')')
        return true;
    else if (character1 == '{' && character2 == '}')
        return true;
    else if (character1 == '[' && character2 == ']')
        return true;
    else
        return false;
}
```


Sakai Code

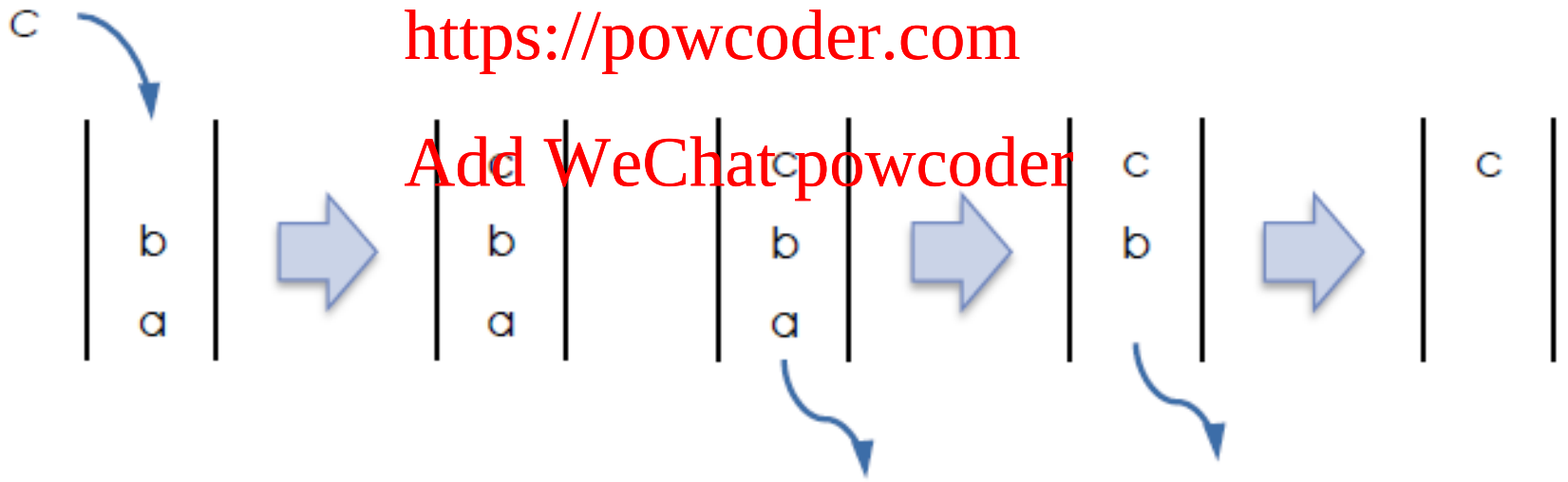
Queue

- *Queue* is a linear data structure which is a container of objects that are inserted and removed according to the first-in first-out (FIFO) principle.
- Example: any queue of consumers for a resource where the consumer that came first is served first.



Queue

- Two fundamental operations
  - enqueue: adds an item at the rear of the queue
  - dequeue: removes and returns the item at the front of the queue



Stack v.s. Queue

- Stack: remove the most recently added item
- Queue: remove the least recently added item

Queue operations

- void enqueue(T item)
- T dequeue(): returns the front entry and removes it
- T peek(): returns the front entry without removing it

If queue does not provide peek operation, how to get the top element?

→ Dequeue all entries and enqueue them

- Utility methods to improve efficiency
  - int size()
  - boolean isEmpty()
  - void clear()

Applications of Queue

- Service requests for shared resources
  - Print jobs

Queue Implementation

- An efficient implementation is to maintain a reference to the rear and a reference to the front to make these positions accessible in one step, namely in O(1) time

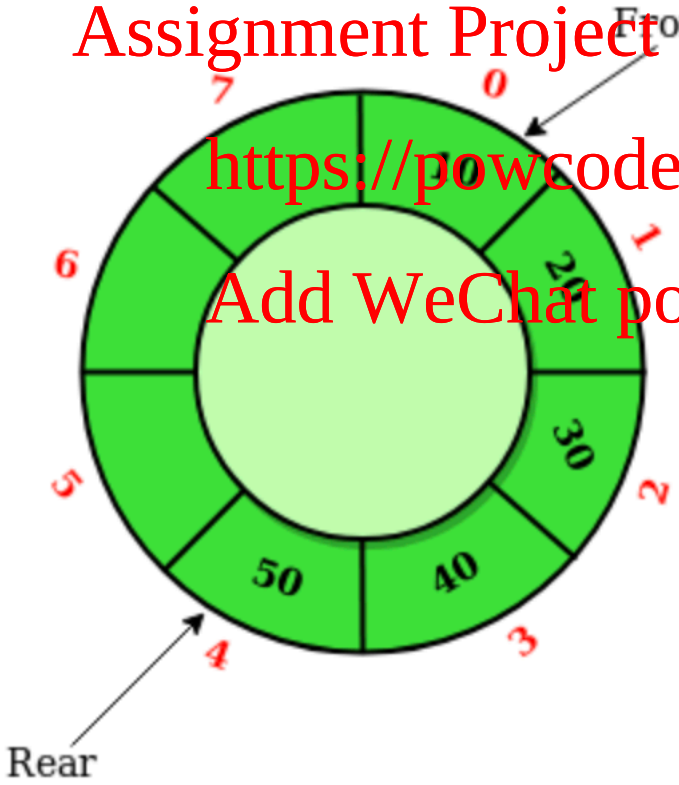
Queue implementation using ArrayList

- Lower end of an array list as the front of the queue and higher end as the rear
- Enqueue an entry: advance the rear index by one
- Dequeue an entry: 0th position becomes empty
- For the empty spot
  - Moving all the entries from right to left: O(n), inefficient
  - Increase the front index by one: O(1), but waste space

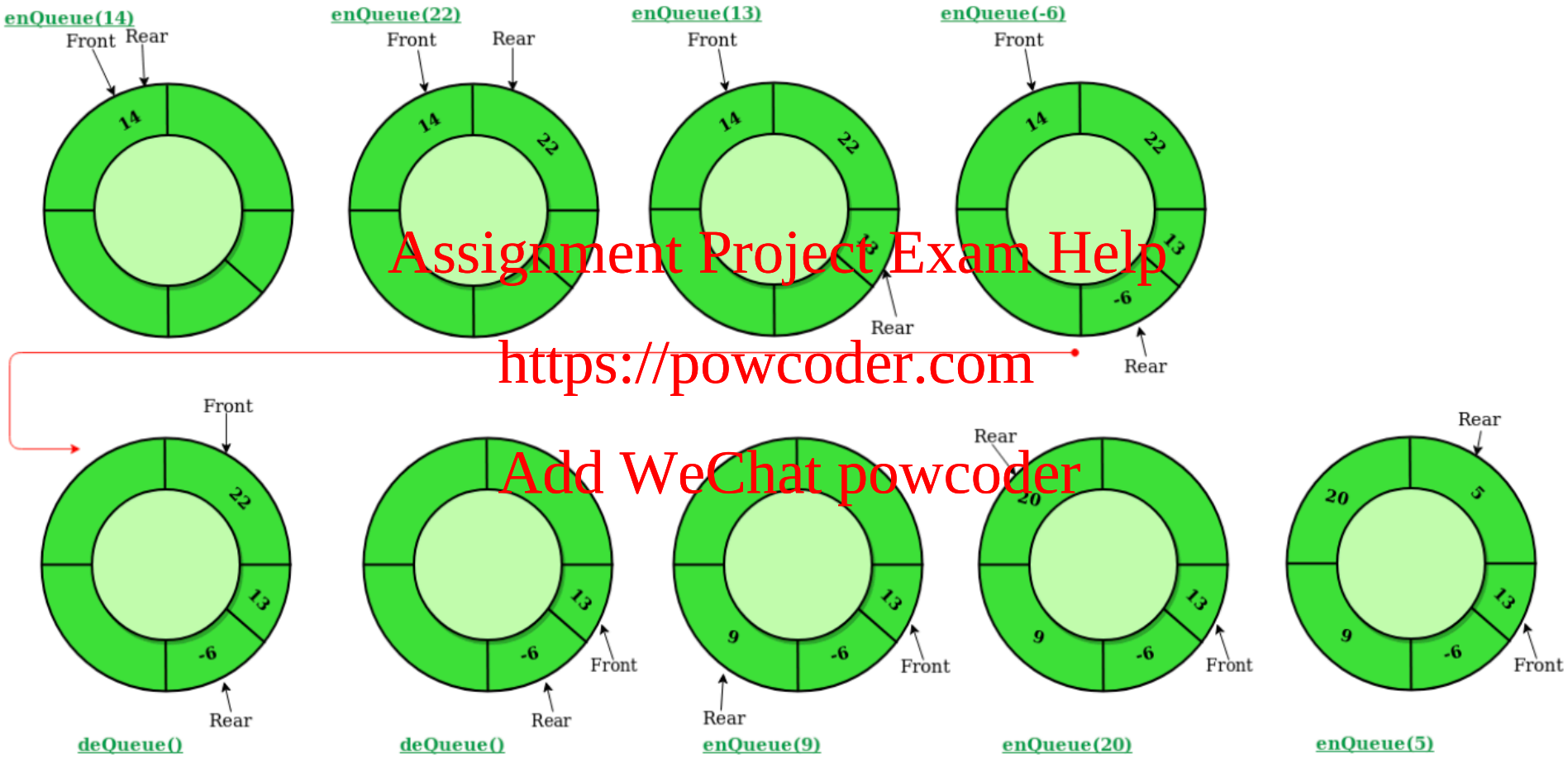
Increase the front index on a *fixed-length array* and *recycle space (wrap around)*

Bounded Queue: Queue implementation using circular array

- Circular array: a data structure that used an array as if it were connected end-to-end



Queue implementation using circular array



Data field for a bounded queue

- `T[] items;`
- `int rear;`
- `int head;`
- `int count;`
  - Keep the total number of entries
  - Starting with 0 when the queue is created.
  - Each time an entry is enqueued or dequeued, increase or decrease the count by one respectively

Create a new bounded queue

```
public BoundedQueue(int capacity){
    if(capacity < 2){
        System.out.println("insufficient capacity");
        items = (T[]) new Object[10];
        //items = new T[10]; will trigger a compile error
    }
    else
        items = (T[]) new Object[capacity];
    rear = -1; // in an empty queue, rear equals to -1
    head = -1; // in an empty queue, head equals to -1
    count = 0; // in an empty queue, count equals to 0
}
```

the total number of items that can be stored in the array

the total number of items that are already stored in the array

Utility methods

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

public boolean isEmpty(){
   return count == 0;
}

public boolean isFull(){
   return count == items.length;
}
```

```
public void enqueue(T item) {
    //1. check whether the queue is already full
    if(isFull()) { System.out.println("full queue");
                                                return ;   }
    //2. check whether the queue is empty
    else if(isEmpty()){
        rear = 0; head = 0;
    }
    //3. check whether rear points to end of the array
    else if(rear == items.length-1)
        rear = 0;
    //4. advance rear to the next spot
    else
       rear++;
    //5. enqueue item into the spot pointed by rear
    items[rear] = item;
    //6. increase the total number of items in the queue
    count++;
}
```

```
public T dequeue() {
    //1. check whether the queue is empty
    if(isEmpty()) throw new NoSuchElementException("empty
queue");
    //2. obtain the item being removed
    T item = items[head];
    //3. check whether there is only one item in the queue
    if(head == rear){
        head = -1; rear = -1;
    }
    //4. check whether head points to the end of the array
    else if(head == items.length - 1)
        head = 0;
    //5. advance head to the next spot
    else
        head++;
    //5. decrease the total number of items in the queue
     count--;
    //6. return the front item
    return item;
}
```

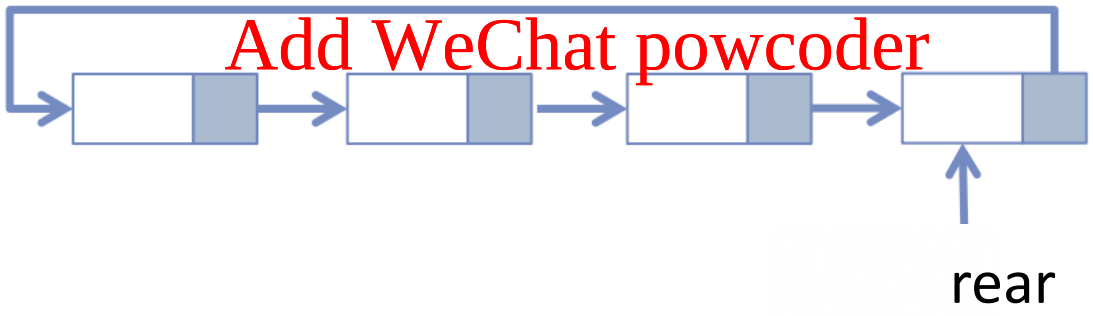
```
public void traverse() {
    //1. check whether the queue is empty
    if(isEmpty())  return ;
    //2. rear is greater than head
    if(rear > head){
       for (int i = head; i <= rear; i++)
          System.out.print(items[i] + " ");
    }
    //3. rear is smaller than head
    else {
        for (int i = head; i < items.length; i++)
          System.out.print(items[i] + " ");
       // the queue wraps around the end of the array
       for (int i = 0; i <= rear; i++)
          System.out.print(items[i] + " ");
    }
}
```

Queue implementation using array

- Circular array is still an array
- The inherent space underestimation or overestimation problem
- Linked list is more attractive

Queue Implementation

- An efficient implementation is to maintain a reference to the rear and a reference to the front to make these positions accessible in one step, namely in O(1) time



Queue implementation using circular linked list

```
public class Queue<T> {

    private CircularLinkedList<T> list = new CircularLinkedList<T>();

    public void enqueue(T item) {
        list.addTail(item);
    }

    public T dequeue() {
        if (list.isEmpty()) {
            throw new NoSuchElementException();
        }
        return list.removeFront();
    }

    public T first() {
        return list.getFront();
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }
}
```


Each operation can finish in O(1) time

Questions from students

- Each class has a default implementation of the method toString() which will be invoked in methods like System.out.println().
- The default implementation will use the class name and hash code to represent an object.
- Need to have your own implementation of toString() in your class to display the object as you like.


```
64
65      public static void main(String[]args) {
66          IntNode front = new IntNode(3, null);
67          front = insert(front,6);
68          traverse(front);
69      }
70  }
71
```

Problems @ Javadoc Declaration Console

```
<terminated> LLApp [Java Application] /Library/Java/JavaVirtualMachines,
IntNode@6ff3c5b5->3
```

```
12      /**
13       * return the string representation of an IntNode object
14       */
15      public String toString() {
16          return data+"";
17      }
18  }
```