

### Push

```
public void push(T newEntry) {
    // Add to beginning of
chain:
    try {
        LinkedStack.Node
newNode = new
LinkedStack.Node(newEntry);
        newNode.next =
firstNode; // Make new node
reference rest of chain
        // (firstNode is null
if chain is empty)
        firstNode = newNode; //
New node is at beginning of chain
        numberOfEntries++;
    } catch (OutOfMemoryError
e) {
        throw new
IllegalStateException();
    }
    //return true;
}
```

### Pop

```
@Override
public T pop() {
    T result = null;
    if (firstNode != null) {
        result =
firstNode.data;
        firstNode =
firstNode.next; // Remove first
node from chain
        numberOfEntries--;
    } else {
        throw new
NoSuchElementException();
    }
    return result;
}
```

### Top

```
public T top() {
    if (firstNode != null) {
        return firstNode.data;
    }
    throw new
NoSuchElementException("Stack is
Empty");
}
```

### Induction Proofs

Claim: *for any*  $n \geq 1$ ,  $1+2+3+4+\dots+n = \frac{n \cdot (n+1)}{2}$

Proof:

• Base case:  $n=1$   $1 = \frac{1 \cdot 2}{2}$  ✓

• Induction step:

*for any*  $k \geq 1$ , if  $1+2+3+4+\dots+k = \frac{k \cdot (k+1)}{2}$

then  $1+2+3+4+\dots+k+(k+1) = \frac{(k+1) \cdot (k+2)}{2}$

### Recursive Fern

```
public void drawFern(double x,
double y, double angle, double
size) {
    if (size > 1.0) { // STOP if size
<= 1.0!
        double[] end;
        double length = size * 0.5;
        end = drawStem(x, y, angle,
length); // private method
        double smaller = size * 0.5; //
SMALLER!
        drawFern(end[0], end[1], angle+60,
smaller);
        drawFern(end[0], end[1], angle,
smaller);
        drawFern(end[0], end[1], angle-60,
smaller);
    }
}
```

### Recursive Binary

```
public static <T> int
binaryFind(Comparable<T> item, T[]
v, int lo, int hi) {
    if (lo > hi) { return -1;
}
    int mid = lo + (hi - lo) /
2;
    if (item.compareTo(v[mid])
< 0) {
        return
binaryFind(item, v, lo, mid - 1);
    } else if
(item.compareTo(v[mid]) > 0) {
        return
binaryFind(item, v, mid+1, hi);
    } else { return mid; } //
found it!
}
```

### Selection Sort

```
public static <T extends
Comparable<? super T>>
void selectionSort(T[]
a) {
    for (int i = 0; i <
a.length - 1; ++i) {
        int minPos = i;
        for (int j = i + 1; j
< a.length; j++) {
            if
(a[j].compareTo(a[minPos]) < 0) {
                minPos = j;
            }
        }
        T temp =
a[minPos];
        a[minPos] = a[i];
        a[i] = temp;
    }
}
```

### Shell Sort

```
public static <T extends Comparable<?
super T>>
    void shellSort(T[] a) {
        int gap = a.length / 2;
        while (gap >= 1) {
            if (gap % 2 == 0) ++gap;
            for (int i = gap; i <
a.length; ++i) {
                int p = i;
                T temp = a[p];
                while (p >= gap && a[p-
gap].compareTo(temp) > 0) {
                    a[p] = a[p-gap];
                    p -= gap;
                }
                a[p] = temp;
            }
            gap /= 2;
        }
    }
```

### Insertion (cont)

```
    }
}
```

### Insertion

```
public static <T extends Comparable<?
super T>>
    void insertionSort(T[] a) {
        for (int i = 0; i < a.length -
1; ++i) {
            int p = i + 1;
            T temp = a[p];
            while (p > 0 && a[p-
1].compareTo(temp) > 0) {
                a[p] = a[p-1];
                --p;
            }
            if (p > 0) // count the
last a[p-1] comparison
                a[p] = temp;
        }
    }
```

