O(1) < O(log n) < O(n) < O(n*log n) < O(n^2)<...O(2^n)<O(n!)

Time Trade Offs

Operation	Linked List	Array
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)
Lookup	O(n)	O(1)

```
for (int i = 1; i < n; i++) {
    for (int j = i; j >= 0; j--) {
        System.out.println("Hello World!");
    }
}
```

$O(f(n)) = O(n^2); T(n) = (n(n+1)/2 - 1$

```
public static Node max(Node list) {
            if (list == null) {
                        throw new IllegalArgumentException("Cannot
e max of empty list.");
            }
            Node maxNode = null;
            Node curr = list;
            int val, max = curr.data;
            while (curr != null) {
                        val = curr.data;
                        if (val > max) {
                              max = val;
                              maxNode = curr;
                        }
                        curr = curr.next;
            }
            return maxNode;
```

```
public static Node maxify(Node list) {
    if (list == null) {
                        return null;
    }
    Node maxNode = max(list);
    Node curr = list;
    Node holder = curr;
    while (curr.next != maxNode) {
        curr = curr.next;
    }
    curr.next = curr.next.next;
    maxNode.next = holder;
    return maxNode;
}
```

```
private void fillStack() {
    for (int i=0; i<inputString.length(); i++) {
        charStack.push(inputString.charAt(i));
    }
}

private String buildReverse() {
    StringBuilder s = new StringBuilder();

    while (!charStack.empty()) {
        s.append(charStack.pop());
    }

    return s.toString();
}

public boolean isPalindrome() {
    return inputString.replaceAll("\\s+","").equalsIgnoreCase
            (this.buildReverse().replaceAll("\\s",""));
}
```

```
public E push(E item) {
    stack = new Node<E>(item,stack);
        ;
        item;

eek() {
ze==0) {
w new EmptyStackException();

    stack.data;
```

```
public E pop() {
    if (size==0) {
        throw new EmptyStackException();
    }
    E temp = stack.data;
    stack = stack.next;
    size--;
    return temp;
}

public boolean empty() {
    return size==0;
}
```

that increments in one unit the priority of every node whose current priority is `lower_bound` or more. Note that the list is not sorted in any way. What is the time complexity of your implementation?

that increments in one unit the priority of every node whose current priority is `lower_bound` or more. Note that the list is not sorted in any way. What is the time complexity of your implementation?

```
public int top_priority() {
    NodeP<E> current = this;
    int max = current.priority;

    while(current != null) {
        max = Math.max(max, current.top_priority());
        current = current.next;
    }

    return max;
}

public void bump_if(int lower_bound) {
    NodeP<E> current = this;

    while(current != null) {
        if (current.priority > lower_bound) {
            current.priority++;
        }
        current = current.next;
    }
}
```

```
public static boolean isBalance(String expression) {
    //Complete
    StackSLL<Character> s = new StackSLL<Character>();
    int i = 0;
    boolean balanced = true;

    while (i<expression.length() && balanced) {
        if (isOpen(expression.charAt(i))) {
            // opening delimeter
            s.push(expression.charAt(i));
            } else {
            //closing delimeter
            balanced = !s.empty() && OPEN.indexOf(s.pop()) == CLOSE.indexOf(expression.charAt(i));
        }
        i++;
    }
    return balanced && s.empty();
}

private static boolean isOpen(char ch) {
    return OPEN.indexOf(ch) > -1;
}

private static boolean isClose(char ch) {
    return CLOSE.indexOf(ch) > -1;
}
```

```
// Data fields
private Node<E> head;
private int size;

// Constructor
SingleLL() {
    head=null;
    size=0;
}

// Methods
public boolean isEmpty() {
    return size==0;
}

public void addFirst(E item) {
    head = new Node<E>(item,head);
    size++;
}

public void addLast(E item) {
    if (head==null) {
        this.addFirst(item);
    } else {
        Node<E> current = head;

        while (current.next!=null) {
            current=current.next;
        }

        current.next = new Node<E>(it
        size++;
    }
}

public E get(int index) {
    if (index<0 || index>size-1) {
        throw new IllegalArgumentExce
    }
    Node<E> current = head;

    for(int i=0; i<index; i++) {
        current = current.next;
    }

    return current.data;

}
```

```
public E removeFirst() {
    if (head==null) {
        throw new IllegalStateException();
    }
    E temp = head.data;
    head = head.next;
    size--;
    return temp;
}

public E removeLast() {
    if (size==0) { // empty list
        throw new IllegalStateException();
    }
    if (size==1) { // singleton list
        return this.removeFirst();
    }
    // list has two or more elements
    Node<E> current=head;

    while(current.next.next!=null) {
        current=current.next;
    }
    E temp = current.next.data;
    size--;
    current.next = null;
    return temp;
}

public E remove(int index) {
    if (index<0 || index>size-1) {
        throw new IllegalArgumentException();
    }
    if (size==1) {
        return this.removeFirst();
    } else {
        Node<E> current=head;
        Node<E> previous=head;

        for (int i=0; i<index; i++) {
            previous = current;
            current = current.next;
        }
        E temp = current.data;
        size--;
        previous.next = current.next;
        return temp;
```

```
public static ArrayList<Integer> toBinary(int i) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    StackSLL<Integer> binaryStack = new StackSLL<Integer>();

    if (i==0) {
        result.add(0);
        return result;
    }

    while(i>0) {
        binaryStack.push(i%2);
        i=i/2;
    }

    int size = binaryStack.size();
    for (int j =0; j<size; j++) {
        result.add(binaryStack.pop());
    }

    return result;
}
```

```
public boolean member(E item) {
    Node<E> current=head;

    while (current!=null && !current.data.equals(it
        current = current.next;
    }

    return current!=null;
}

public SingleLL<E> take(int n) {
    SingleLL<E> l = new SingleLL<E>();
    int i = 0;
    Node<E> current = head;

    while (current!=null && i<n) {
        l.addLast(current.data);
        current = current.next;
        i++;
    }

    return l;
}

public SingleLL<E> take2(int n) {
    SingleLL<E> l = new SingleLL<E>();
    int i = 0;
    Node<E> current = head;
    Node<E> last = new Node<E>();
    Node<E> newHead = last;

    while (current!=null && i<n) {
        last.next = new Node<E>(current.data);
        last = last.next;
        current = current.next;
        i++;
    }
    l.head = newHead.next;
    l.size = i;
    return l;
}
```

```
    public boolean hasRepetitions() {
                Node<E> current=head;

                while (current!=null
&& !member(current.next,current.data)) {
                            current=current.next;
                }
            return current!=null;
    }

    public void stutter() {

            Node<E> current = head;

            while (current!=null) {
                        current.next = new
Node<E>(current.data,current.next);
                        current = current.next.next;
            }
    }
```

```
public void take3(int n) {
    if (head==null || n<=0) { // list is empty or n==0
        head=null;
        size=0;
    } else { // n>0 and list is nonempty

        int i = 0;
        Node<E> current = head;
        while (current.next!=null && i<n-1) {
            current = current.next;
            i++;
        }
        current.next=null;
        size = i;
    }

}

public String toString() {
    StringBuilder s = new StringBuilder();

    s.append("[");
    Node<E> current = head;
    while (current!=null) {
        s.append(current.data.toString()+";");
        current = current.next;
    }
    s.append("]");
    return s.toString();

}
```