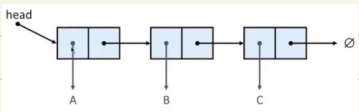
Video 1. Singly Linked Lists

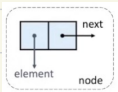
linked list – collection of nodes that have references between them



start with head pointer

tail – last node of the list; next = null

traversal – moving through the list (next)

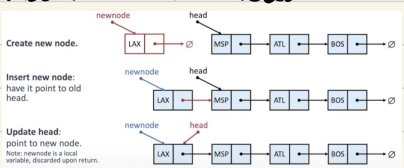


(reference to) element

reference to next node in the list

```
public class Node<E> {
  private E element;
  private Node<E> next;
  public Node(E e, Node<E> n) {
    element = e;
    next = n;
  }
  public E getElement() {
    return element;
  }
  public Node<E> getNext() {
    return next;
  }
  public void setNext(Node<E> n) {
    next = n;
  }
}
```

insert at the head

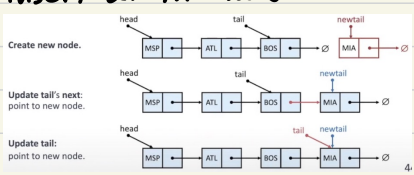



```
public void addFirst(E e) {
  head = new Node<>(e, head);
  if (size == 0)
    tail = head;
  size++;
}
```

time: O(1)

space: O(1)

insert at the tail




```
public void addLast(E e) {
  Node<E> n =
      new Node<>(e, null);
  if (isEmpty()) head = n;
  else tail.setNext(n);
  tail = n;
  size++;
}
```

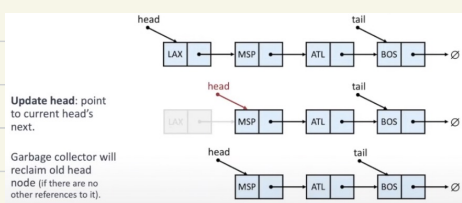
time: O(1)

space: O(1)

For a(n) array/list with **n** elements:

	Array	SLL
Access element (by index)	**O(1)**	O(n)
Search element	O(n)	O(n)
Insertion		
At head	O(n)	**O(1)**
At tail	**O(1)**	**O(1)**
Deletion		
At head	O(n)	**O(1)**
At tail	**O(1)**	O(n)
Clone/copy	O(n)	O(n)

remove head




```
public void removeFirst(E e){
 if (isEmpty()) return null;
 E e = head.getElement();
 head = head.getNext();
 size--;
 if (size == 0) tail = null;
 return e;
}
```

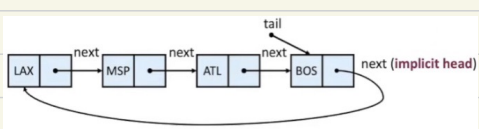
time: O(1)

space: O(1)

advantages:
- efficient growth

disadvantages:
- access = traversal

Video 2. Circularly Linked Lists

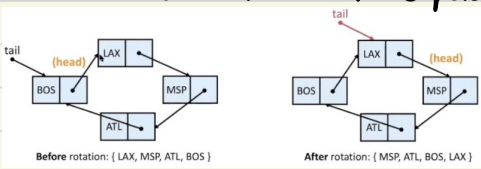


SLL with tail.next() = head

SLL operations + rotate()

implicit head (tail.next())

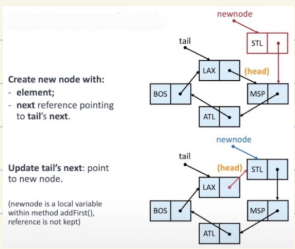
rotate – tail moved to explicit head (next node)




```
public void rotate(){
 if (tail != null)
 tail = tail.getNext();
}
```

time: O(1)
space: O(1)

insert at head



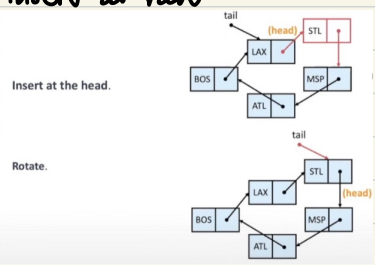

```
public void addFirst(E e){
 if (size == 0) {
   tail = new Node<>(e, null);
   tail.setNext(tail);
 }
 else {
   Node<E> n = new Node<>(e,
     tail.getNext());
   tail.setNext(n);
 }
 size++;
}
```

time: O(1)
space: O(1)

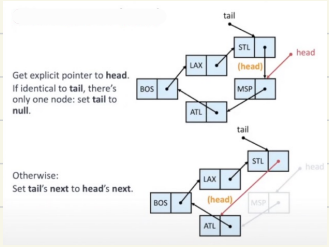
time: O(1) space: O(1)

```
public void addLast(E e)
  addFirst(e);
  tail = tail.getNext();
}
```

insert at tail



remove head




```
public E removeFirst(){
 if (isEmpty())
   return null;
 Node<E> h = tail.getNext();
 if (h == tail)
   tail = null;
 else
   tail.setNext(h.getNext());
 size--;
 return h.getElement();
}
```

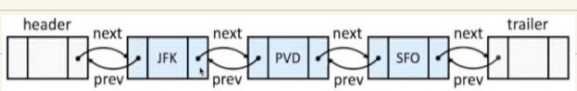
For a(n) array/list with **n** elements:

	Array	SLL	CLL
Access element (by index)	**O(1)**	O(n)	O(n)
Search element	O(n)	O(n)	O(n)
Insertion			
At head	O(n)	**O(1)**	**O(1)**
At tail	**O(1)**	**O(1)**	**O(1)**
Deletion			
At head	O(n)	**O(1)**	**O(1)**
At tail	**O(1)**	O(n)	O(n)
Clone/copy	O(n)	O(n)	O(n)

time: O(1)
space: O(1)

advantages: efficient expansion
disadvantages: access = traversal

Video 3. Doubly Linked Lists

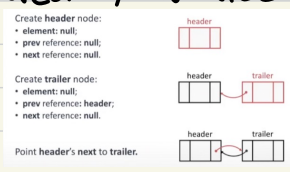


each node contains:
- element
- reference to next node
- reference to previous node

sentinel header and trailer

```
public class DoublyLinkedList<E> {
 private static class Node<E> {
   private E element;
   private Node<E> prev;
   private Node<E> next;
   public Node(E e, Node<E> p, Node<E> n){
    ...
   }
   public E getElement(){ return element; }
   public Node<E> getPrev(){ return prev; }
   public Node<E> getNext(){ return next; }
   public void setPrev(E p){ prev = p; }
   public void setNext(E n){ next = n; }
 }
 private Node<E> header;
 private Node<E> trailer;
 private int size = 0;
```

create/initialize




```
public DoublyLinkedList(){
  header = new Node<>(null, null, null);
  trailer = new Node<>(null, header, null);
  header.setNext(trailer);
}
```

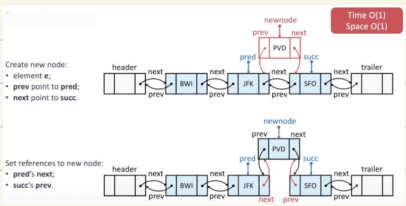
time: O(1)
space: O(1)

```
private void addBetween(E e, Node<E> predecessor, Node<E> successor)
private E remove(Node<E> node)
```

Insert at the head:
```
public void addFirst(E e) {
        addBetween(e, header, header.getNext());
}
```

Insert at the tail:
```
public void addLast(E e) {
        addBetween(e, trailer.getPrev(), trailer);
}
```

insert

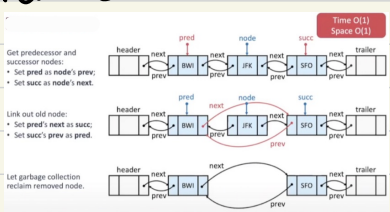



```
private void addBetween(E e, Node<E> predecessor, Node<E> successor)
private E remove(Node<E> node)
```

Remove head:
```
public E removeFirst() {
        if(isEmpty()) return null;
        else return remove(header.getNext());
}
```

Remove tail:
```
public E removeLast() {
        if(isEmpty()) return null;
        else return remove(trailer.getPrev());
}
```

remove



For a(n) array/list with **n** elements:

	Array	SLL	CLL	DLL
Access element (by index)	**O(1)**	O(n)	O(n)	O(n)
Search element	O(n)	O(n)	O(n)	O(n)
Insertion				
At head	O(n)	**O(1)**	**O(1)**	**O(1)**
At tail	**O(1)**	**O(1)**	**O(1)**	**O(1)**
Deletion				
At head	O(n)	**O(1)**	**O(1)**	**O(1)**
At tail	**O(1)**	O(n)	O(n)	**O(1)**
Clone/copy	O(n)	O(n)	O(n)	O(n)

advantages: efficient expansion, ins/del at head/tail
disadvantages: access = traversal, added space for prev reference