Lists and Linked Lists

List ADT: some or all of these operations

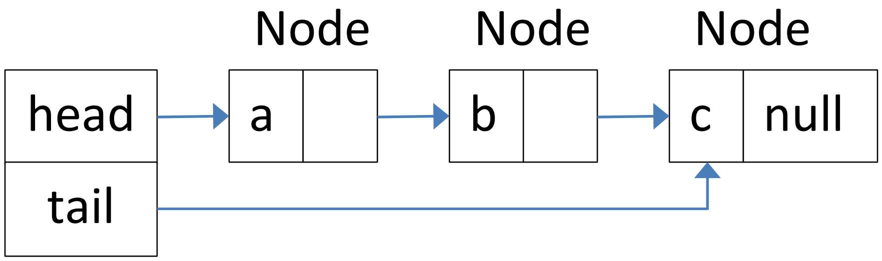
```
void insert (ElementType  x)
void insertFirst (ElementType  x)
void insertLast (ElementType  x)
void insertAfter (Position p, ElementType  x)
void insertBefore (Position p, ElementType  x)
void insertAtRank (int rank, ElementType  x)
void remove (ElementType  x)
void removeFirst( )
void removeLast( )
void remove (Position p)
void removeAtRank (int rank)
void replace (Position p, ElementType  x)
void replaceAtRank (int rank, ElementType  x)
boolean find (ElementType  x)
Position findPosition (ElementType  x)
int findRank (ElementType  x)
ElementType  element (Position p)
ElementType  elementAtRank (int rank)
int  toRank (Position p)
Position  toPosition (int rank)
Position first( )
Position last( )
Position after (Position p)
Position before (Position p)
boolean isEmpty( )
int size( )
```

Standard data structures for List ADT:

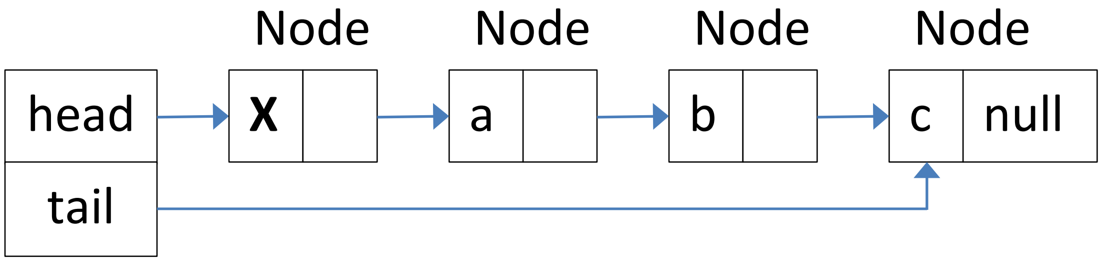
- Array
- Circular array
- Singly-linked list (several variations)
- Doubly-linked list (several variations)

In efficient implementations, some operations might take $\theta(1)$ time and other operations might take $\theta(n)$ time

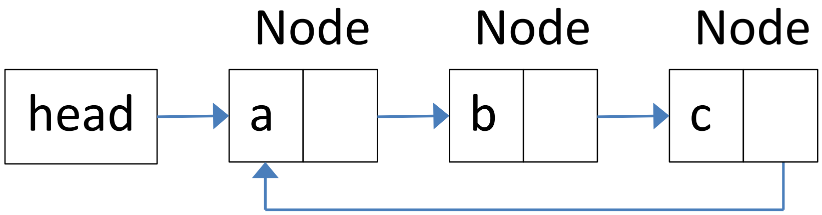
Singly-linked list



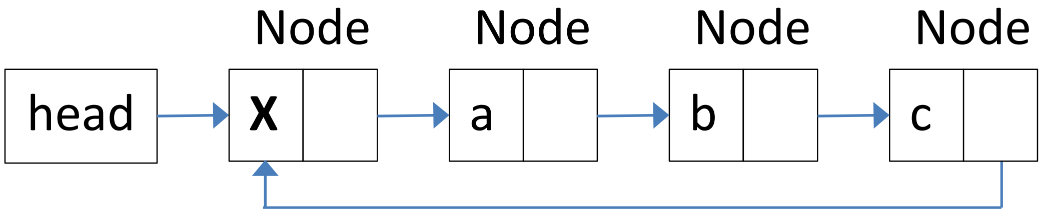
Singly-linked list with header node



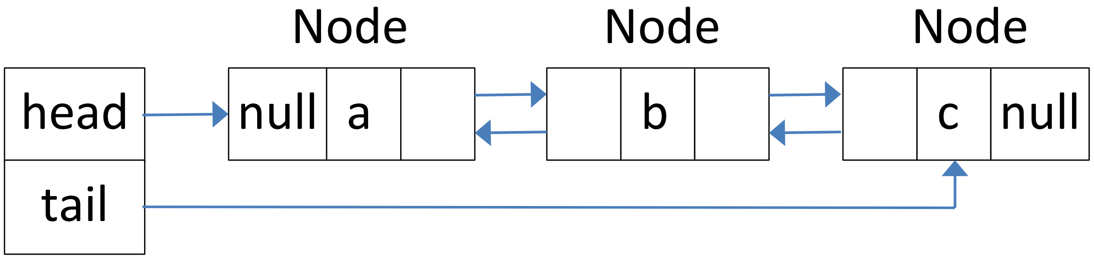
Circular singly-linked list



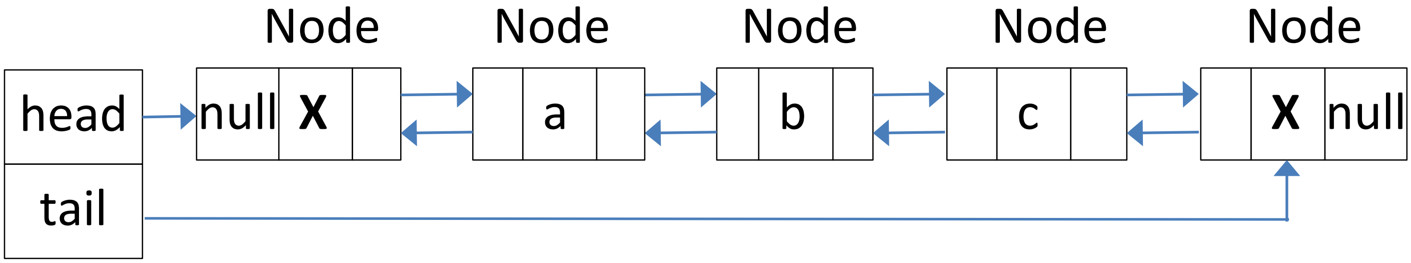
Circular singly-linked list with header node



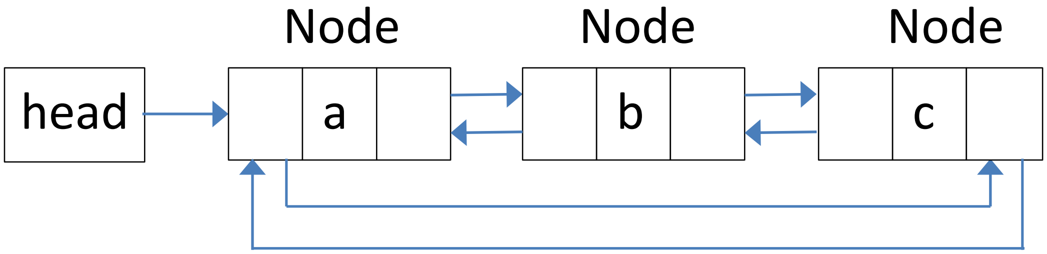
Doubly-linked list



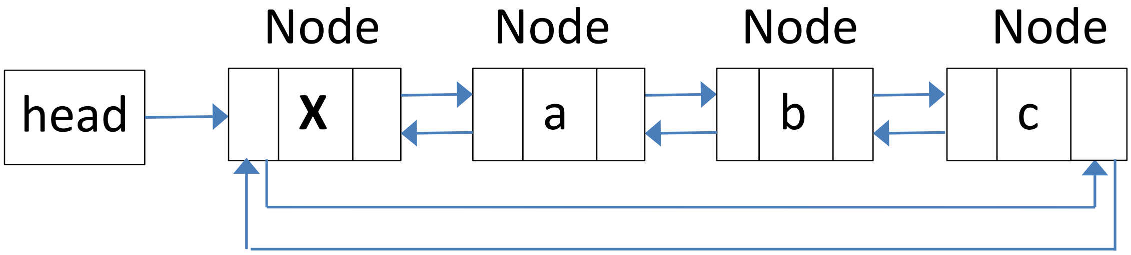
Doubly-linked list with header and trailer nodes



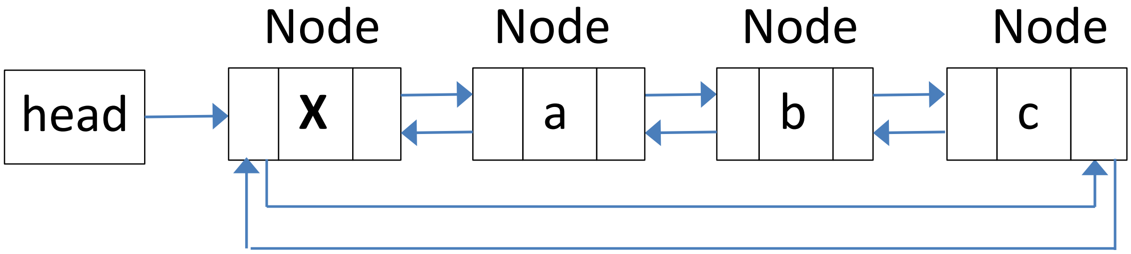
Circular doubly-linked list



Circular doubly-linked list with header node



Here we'll use a circular doubly-linked list with header node



To keep it simpler, we won't check for any error conditions

```
class Node {
        ElementType  data;
        Node prev, next;
        Node (ElementType x, Node p, Node q) {
                data = x;  prev = p;  next = q;
        }
 }
 class Position (Node) {
 }
 class List {
        Node head;
        List( ) {
                head = new Node( );
                head.prev = head;
                head.next = head;
        }
        void insertFirst (ElementType  x)
                { insertBefore (first( ), x); }
        void insertLast (ElementType  x)
                { insertAfter (last( ), x); }
```

```
void insertAfter (Position p, ElementType  x) {
        Node q = new Node (x, p, p.next);
        p.next.prev = q;
        p.next = q;
}
void insertBefore (Position p, ElementType  x) {
        Node q = new Node (x, p.prev, p);
        p.prev.next = q;
        p.prev = q;
}
void insertAtRank (int rank, ElementType  x)
        { insertBefore (toPosition(rank), x); }
void remove (Position p) {
        p.prev.next = p.next;
        p.next.prev = p.prev;
        p.prev = p.next = null;
}
void removeAtRank (int rank)
        { remove (toPosition(rank)); }
ElementType  element (Position p)
        { return p.data; }
ElementType  elementAtRank (int rank)
        { return toPosition(rank).data; }
int  toRank (Position p) {
        int r=0;
        for (Node q = first( );  q != p;  q = q.next)  r += 1;
        return r;
}
```

```
    Position  toPosition (int rank) {
        Node q = first( );
        for (int r=0;  r != rank;  r += 1)  q = q.next;
        return q;
    }
    Position first( ) { return head.next; }
    Position last( ) { return head.prev; }
    Position after (Position p) { return p.next; }
    Position before (Position p) { return p.prev; }
    boolean isEmpty( ) { return head.next == head; }
}
```

Running times for the above data structure:

$\theta(1)$:	$\theta(n)$:	Not written:
insertFirst (x) insertLast (x) insertAfter (p, x) insertBefore (p, x) remove (p) element (p) first( ) last( ) after (p) before (p) isEmpty( )	insertAtRank (r, x) removeAtRank (r) elementAtRank (r) toRank (p) toPosition (r)	insert (x) remove (x) removeFirst( ) removeLast( ) replace (p, x) replaceAtRank (r, x) find (x) findPosition (x) findRank (x) size( )