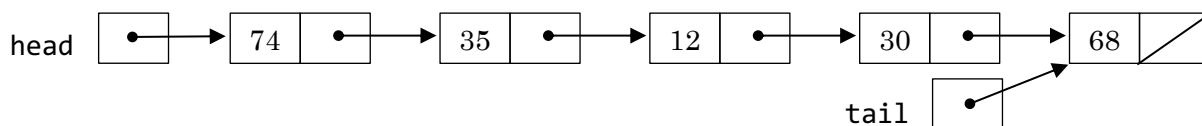


1. 下記のプログラムは、**単方向連結リスト**にデータの末尾を常に指すフィールド **tail** を加えて第 8 回の 3. と同様のキューを実現したものである。適切に空欄を埋めてプログラムを完成させ、動作を確認せよ。



```

/* Node.java */
public class Node { // リスト要素 (ノード)

    // 同一パッケージからのみアクセス可能
    int data; // 格納データ
    Node next; // 次のノードへのリンク

    Node(int data) {

    }

}

```

```

/* Queue.java */
public class Queue {

    // 連結リストの先頭と末尾のノードを指す
    private Node head, tail;

    public Queue() {
        head = tail = null;
    }

    // リストの末尾にノードを加える
    public void enqueue(int data) {
        Node n = new Node(data);

        // もしリストが空だった場合
        if (head == null) {
            head = tail =
        }
        else { // そうでない場合
            = n;

            tail =
        }
    }
}

```

```

// リストの先頭からノードを取り出す
public int dequeue() throws Exception {
    // データがない場合をチェック
    if (head == ) {

        throw new Exception("empty!");
    }
    // data を取り出し、新しい head を設定
    int data = head.data;
    head =

    // リストが空になった場合
    if (head == null)
        tail =

    return data;
}

public void printAll() {
    System.out.print("[ ");

    // リストの中のノードを 1 つずつたどる
    for (Node n = head;
        n != null; n = n.next) {

    }
    System.out.println("]");
}
}

```

```

/* Main.java */
/* メインプログラムは第 8 回の 4. のものを
   そのまま利用する。ただし new Queue(8)
   は new Queue() に書き換える。
*/

```

2. 1.のプログラムにおいて、**enqueue** と **dequeue** のリンクのつなぎ替えの処理を図示せよ。さらに、キュー内のデータ数 **n** に対する **enqueue** と **dequeue** の計算量（オーダー）および **tail** の役割を考察せよ。

3. 第9回の2.のListクラスに、引数のノードpの次に新しいノードを挿入するメソッドinsertNextと、同様に、pの次のノードを削除するメソッドremoveNextを追加し、mainも変更して動作を確認せよ。

```
// ノードpの“次”に新ノードnを挿入
public void insertNext(Node p, Node n) {
    // pがnullのときは先頭に挿入
    if (p == null) {
        n.next = head;
        head = n;
    } else {
        n.next =
        p.next =
    }
}
```

```
// ノードpの“次”のノードを削除
public void removeNext(Node p) {
    // pがnullのときは先頭を削除
    if (p == null) {
        if (head != null)
            head = head.next;
    } else if (p.next != null) {
        p.next =
    }
}
```

4. さらに、ジェネリクスを用いて第9回の2.と5.の連結リストを定義し直す。下記のクラス定義の最初のclass Node<E>は、Eに任意のクラス名が当てはめられてくることを示し、フィールドおよびメソッドの定義でEを使うことができる。ジェネリッククラスの利用方法は、第6回で学んだ通りである。

```
/* Node.java */
public class Node<E> {
    public E data;
    public Node<E> next;

    public Node(E data) {
        this.data = data;
        next = null;
    }
}
```

```
/* List.java */
public class List<E> {
    public Node<E> head;

    public List() {
        head = null;
    }

    public void printAll() {
        // 第9回の2.を参考に作成せよ
    }
}
```

```
public Node<E> search(E data) {
    // 第9回の5.を参考に作成せよ
}

// insertNext と removeNext も追加するとよい
}
```

```
/* Main.java */
public class Main {
    public static void main(String[] args) {
        List<String> list = new List<String>();
        list.head = new Node<String>("ABC");

        // 適当にリストを構築して表示させてみよ

        list.printAll();
        String key = new java.util.Scanner(
            System.in).next();
        Node<String> n = list.search(key);
        if (n != null)
            System.out.println(n.data);
    }
}
```

5. Java では java.util パッケージにおいて、双方向連結リストのクラス LinkedList<E>が提供されている。このクラスでは、例えば下の表のような操作が可能である（詳しくは Java の公式リファレンスを参照）。これを参考に、LinkedList<String>のインスタンスにいくつかの文字列を登録（追加）し、辞書順（アルファベット順・あいうえお順）に並べ替えてから、全要素を表示するプログラムを作成せよ。

先頭への追加	list.addFirst(e) list.push(e)	末尾への追加	list.addLast(e) list.add(e)
先頭からの取り出し	e = list.removeFirst() e = list.pop()	末尾からの取り出し	e = list.removeLast() e = list.poll()
先頭からの線形探索	i = list.indexOf(key)	末尾からの線形探索	i = list.lastIndexOf(key)
マージソート	Collections.sort(list)	全要素へのアクセス	for (E e : list)

6. 【発展】逆方向にもリンクをたどれる**双方向連結リスト**を自作してみよう。下記のコードの空欄を適切に埋めて双方向連結リストを表すクラス完成させ、(1)～(8)の手順で動作を確認するプログラムを作成せよ。
- (1) 空のリストを生成する。
  - (2) キーボードから文字列を 5 つ読み込み、リストの先頭から入力順に並ぶように insert で追加する。
  - (3) 先頭から順にリストの全要素を表示し、入力順になっていることを確認する。
  - (4) 文字列を読み込み、リストから要素 X を探索する。ただし、X は存在するものを入力する。
  - (5) 要素 X からリストの末尾までの要素を表示する。
  - (6) 要素 X をリストから削除して、リストの全要素を先頭から順に表示する。
  - (7) 別の文字列を読み込み、要素 Y を探索する。ただし、Y は存在するものを入力する。
  - (8) 要素 Y の直前の位置に要素 X を再挿入して、要素 Y から先頭まで逆順に表示する。

```
/* Node.java */
public class Node {
    String data;
    Node prev; // 前のノード
    Node next; // 次のノード

    public Node(String data) {
        this.data = data;
        prev = next = null;
    }
}
```

```
/* List.java */
public class List {

    public Node head;

    // 指定ノード p の直後に新ノード n を挿入
    public void insert(Node p, Node n) {
        if (p != null) {
            n.prev =
            n.next =
            p.next =

        } else {
            // p が null の場合は先頭に挿入
            n.prev = null;
            n.next = head;
            head = n;
        }
        if (n.next != null) {
            n.next.prev =

        }
    }
}
```

```
// ノード n の削除
public void remove(Node n) {
    if (n == head) {
        head = n.next;
    }
    if (n.prev != null) {
        = n.next;

    }
    if (n.next != null) {
        = n.prev;

    }
}

// ノード n から後ろをすべて表示
public void printToTail(Node n) {
    while (n != null) {
        System.out.print(n.data + " ");
        n = n.next;
    }
    System.out.println();
}

// ノード n より前をすべて（逆順に）表示
public void printToHead(Node n) {

    // printToTail を参考にして作成する

}

// データの探索
public Node search(String data) {

    // 先頭から順に探索する（なければ null）

}
}
```

7. 【発展】連結リストの要素のソートでは、挿入ソートまたはマージソートが用いられることが多い。連結リストの場合、配列に対するソートとは異なり、値のコピーは行わずにノードのつなぎ換えで並べ替えを進めることになる。第9回の2.のListクラスに下記のマージソートのメソッドを追加し（空欄部分は適切に補い）、動作を確認せよ。また、連結リストのソートでマージソートが用いられる理由を考察せよ。

```
// リストの中央位置を求める
protected Node midpoint() {
    Node p = head, q = head;
    for (;;) {
        // p を 2 つ進めると同時に
        p = p.next;
        if (p == null) break;
        p = p.next;
        if (p == null) break;

        // q は 1 つ進めていく
        q =

    }
    // p が末尾に到達すると q は中央位置を指す
    return q;
}

// マージソート
public void sort() {
    // 要素がないか 1 つだけならなにもしない
    if (head == null || head.next == null)
        return;

    // 分割のためにリストの中央位置を求める
    Node mid = midpoint();

    // 後半のリストは, mid の次から末尾まで
    List secondHalf = new List();
    secondHalf.head = mid;

    // 前半のリストは, 先頭から mid まで
    List firstHalf = new List();
    firstHalf.head =

    mid.next = null; // 後半への連結を切る

    // 前半と後半を, 再帰的にソート済みにする
    firstHalf.

    secondHalf.
```

```
// 前半と後半のリストからマージしていく
// 末尾につなげていくので tail を使う
head = null;
Node tail = null;

Node p = firstHalf.head;
Node q = secondHalf.head;

while (p != null || q != null) {

    // 次に末尾につなげる n を求める
    Node n = null;
    if (p == null) {

        n =

        q = q.next;
    } else if (q ==

    ) {

        n =

        p = p.next;
    } else if (p.data <= q.data) {
        n =

        p = p.next;
    } else {
        n =

        q = q.next;
    }

    // 求めた n をリストの末尾につなげる
    if (head == null) {
        head = n;
    } else {
        = n;

    }
    tail =
}
// 最後に末尾の連結先を null に設定
tail.next = null;
}
```