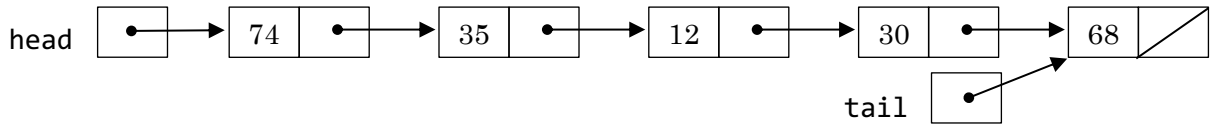
アルゴリズムとデータ構造 2020　第 10 回 演習課題　「連結リストの応用」

1. 下記のプログラムは，第 8 回の 3.と同様のキューを**単方向連結リスト**に末尾を指すフィールド tail を加えて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("]");
   }
}
```

```
/* Program.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，下記のように**ジェネリクス**を用いて第 9 回の 2.の連結リストを定義し直す。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については，第 6 回で学んだ通りである。

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

```
/* Program.java */
public class Program {
   public static void main(String[] args) {

      // new の<>の中は明確なら省略できる
      List<String> list = new List<>();
      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();
      Node<String> n = list.search("DEF");
      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 API の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】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 ==        ) {

            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;
}
```