Obs. Nos exercícios com listas ligadas, vamos assumir que temos

```
typedef struct node *link;
struct node { Item item; link next; }
```

1. Neste exercício, assuma que implementamos listas ligadas sem cabeça e sem cauda de lista. Escreva uma função de protótipo

```
link reverta(link h);
```

que recebe um ponteiro para o primeito elemento de uma lista (`NULL` caso esta lista seja vazia) e devolve um ponteiro para o primeiro elemento da *reversão* da lista dada, isto é, uma lista com exatamente os mesmos elementos da lista original, mas com os elementos em ordem reversa. Aqui, você deve apenas alterar os ponteiros dos elementos da lista para conseguir este objetivo; você não deve alocar mais memória para produzir a lista nova.

2. Suponha neste exercício que as nossas listas estão implementadas com cabeças de listas (em particular, a lista 'vazia' contém exatamente um elemento, a saber, a cabeça de lista). Ademais, assuma que `Item` é o tipo `int`. Escreva uma função de protótipo

```
link acerte_prim_el(link h);
```

que recebe um ponteiro para o começo de uma lista e reorganiza os elementos desta lista de forma que o elemento com a menor chave (isto é, com o menor campo `item`) é o primeiro elemento da lista nova. A sua função deve devolver um ponteiro para o começo da nova lista.

3. Escreva uma função de protótipo

```
link ordene(link h);
```

que ordena os elementos de uma lista em ordem crescente das chaves (isto é, campo `item`, que assumimos serem inteiros). Use a função do exercício anterior. Tanto a lista original como a lista resultante devem ter cabeças de lista.

4. [3.36 do Sedgewick] Escreva uma função que rearranja os elementos de uma lista dada, pondo todos os elementos que ocorrem nas posições pares antes de todos os elementos de ocorrem nas posições ímpares. Você deve preservar a ordem relativa dos elementos pares entre si, e a ordem relativa dos elementos ímpares entre si.

5. [3.38 do Sedgewick] Escreva uma função de protótipo

```
link copia(link h);
```

que recebe um ponteiro para o começo de uma lista e devolve um ponteiro para uma nova lista, que é uma cópia da lista dada. A nova lista deve conter uma cópia de cada elemento da lista original; estas cópias devem ocorrer 'na mesma ordem' que na lista original.

6. Assuma que pomos

```
typedef struct { float Re; float Im; } *Complex;
```

Escreva uma função de protótipo

```
Complex COMPLEXmult(Complex z, Complex w);
```

que devolve (um ponteiro) para o produto de `z` e `w`.

7. Suponha novamente que temos definido o tipo `Complex`. Suponha também que temos listas ligadas com `Item` definido como `Complex`. Escreva uma função de protótipo

```
Complex COMPLEXsum(link h);
```

que recebe uma lista ligada representando uma seqüência de números complexos, e que devolve a soma desta lista de números complexos. Assuma que a lista não tem nem cabeça nem cauda de lista.

8. [4.6 do Sedgewick] Este é um exercício sobre pilhas. Na seqüência

```
E A S * Y * Q U E * * * S T * * * I O * N * * *
```

uma letra significa *empilhe* e um asterisco *desempilhe*. Qual é a seqüência de letras devolvida pelos *desempilhe*?

9. [4.7 do Sedgewick] Continuemos usando a notação do exercício anterior. Diga como devemos inserir asteriscos na seqüência `E A S Y` para que os *desempilhe* devolva as seguintes seqüências: (*i*) `E A S Y`, (*ii*) `Y S A E`, (*iii*) `A S Y E`, (*iv*) `A Y S E`, (*v*) `E Y A S`. Note que nem sempre podemos obter a saída desejada; nestes casos, argumente por que a dada saída não pode ser obtida desta forma.

10. [4.9 do Sedgewick] Converta a expressão

```
( 5 * ( ( 9 * 8 ) + ( 7 * ( 4 + 6 ) ) ) )
```

para a notação pósfixa e para a notação infixa.

11. [4.31 do Sedgewick] Este é um exercício sobre filas. Na seqüência

```
E A S * Y * Q U E * * * S T * * * I O * N * * *
```

uma letra significa inserção no fim da fila e um asterisco significa remoção do começo da fila. Qual é a seqüência de letras devolvida por esta seqüência de operações em nossa fila?

12. [O problema das panquecas] Suponha que temos, por exemplo, a seqüencia de números

```
8 2 3 5 1 4 9 7 6 0
```

Imagine que esta seqüência indica uma pilha de panquecas de 10 tamanhos distintos, empilhadas de forma que a menor está no fundo da pilha, a maior é a quarta panqueca a partir do fundo da pilha, a segunda maior está no topo, etc. As únicas operações permitidas são inverter um segmento inicial desta seqüência. Portanto, existem 9 operações não-triviais para a pilha acima; uma delas produz, por exemplo,

```
5 3 2 8 1 4 9 7 6 0
```

Podemos chamar esta operação deste exemplo de 'operação número 4', já que invertemos a ordem das 4 panquecas do topo. O problema é o seguinte: dados um inteiro positivo `n` e uma permutação dos inteiros não-negativos menores do que `n`, determinar uma seqüência de operações (permitidas) que resultam na seqüência

```
0 1 2 ... n - 1
```

Isto é, com as panquecas em ordem crescente, do topo para o fundo. Escreva uma função de protótipo

```
void imprima_solucao(int n, int panq[]);
```

para resolver este problema.

13. [5.11 do Sedgewick] Escreva um programa recursivo que converte uma expressão em notação infixa em notação pósfixa.

14. [5.14 do Sedgewick] Escreva uma função recursiva que remove o último elemento de uma lista ligada.

15. [5.17 do Sedgewick] Escreva uma função recursiva de protótipo

```
Item max_el(link h);
```

que recebe um ponteiro para o começo de uma lista ligada (sem cabeça e sem cauda) e que devolve a maior chave (isto é, campo `item`) presente na lista, onde assumimos que `Item` é o tipo `int`.

16. [5.48 do Sedgewick] Determine o conteúdo dos vetores `maxKnown[]` e `itemKnown[]` que são determinados pelo programa 5.13 (prog5.13.c da página) na chamada `knap(17)`, com os itens

	0	1	2	3	4
item	A	B	C	D	E
size	3	4	7	8	9
valor	4	5	10	11	13

17. [5.55 do Sedgewick] Lembre que os coeficientes binomiais satisfazem a recorrência

$$\binom{N}{k} = \binom{N-1}{k} + \binom{N-1}{k-1}$$

para todo $k \geq 1$ e todo N. Ademais, temos $\binom{N}{0} = \binom{N}{N} = 1$. Escreva uma função recursiva de protótipo

```
long binom(long N, long k);
```

que calcula $\binom{N}{k}$. Use programação dinâmica. Verifique experimentalmente a diferença de eficiência entre as implementações sem e com programação dinâmica.

18. [5.62 do Sedgewick] Existe uma correspondência entre árvores binárias e certas seqüências de bits. Para uma seqüência de bits corresponder a uma AB, devemos ter que o número de `0`s excede de um o número de `1`s e ainda vale que, lendo a seqüência da esquerda para a direita, o número de `0`s nunca excede o número de `1`s. Estas seqüências (que chamaremos de *válidas*) também podem ser caracterizadas da seguinte forma: a seqüência com um único bit `0` é valida; se duas seqüências são válidas então a concatenação delas, antecedida por um bit `1`, é válida.

Desenhe a AB que corresponde à seqüência válida

```
1 1 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 0
```

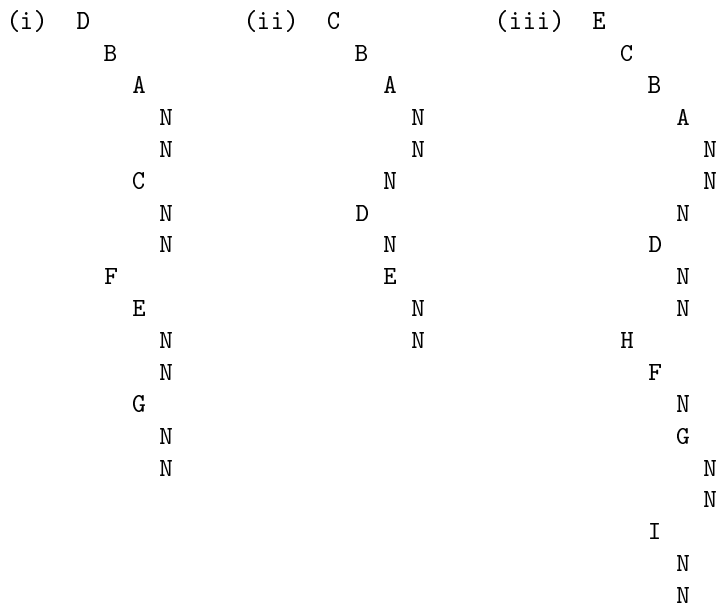
19. [5.63 do Sedgewick, variante] ABs são equivalentes a seqüências bem-formadas de parênteses. Note que seqüências bem-formadas de parênteses podem ser definidas desta forma: a seqüência `( )` é bem-formada. Se duas seqüências `s` e `t` são bem-formadas, então `( s t )` é bem formada.

Desenhe a AB que corresponde à seqüência

```
( ( ( ( ) ( ( ) ( ) ) ) ( ( ( ) ( ) ) ( ) ) ) ( ) )
```

20. [5.79 do Sedgewick] Suponha que percorremos as árvores abaixo em pré-ordem, in-ordem, pós-ordem, e

por níveis, imprimindo o conteúdo dos nós. Quais são as seqüências de caracteres assim obtidas?



Observação. Obtivemos as saídas acima através de algo como

```
printnode(h->token); /* assumimos que printnode() acerta a endentacao */
show(h->l);          /* a subárvore esquerda primeiro, recursivamente */
show(h->r);          /* depois a direita */
```

executado recursivamente.