6. Qual a vantagem das listas sobre os vetores em termos de consumo de memória? Exemplifique.

A principal vantagem das listas encadeadas é a **flexibilidade do uso da memória** através da alocação dinâmica.

- **Vetor (com alocação estática):** Você precisa pré-alocar um tamanho máximo fixo em tempo de compilação. Se você aloca um vetor para 1.000 contatos, mas usa apenas 50, os 950 espaços restantes representam um desperdício de memória.
- **Lista Encadeada:** A memória é alocada nó por nó, em tempo de execução. Se você precisa armazenar 50 contatos, você aloca memória dinamicamente apenas para esses 50 nós. Se precisar de mais um, aloca espaço para mais um. Isso evita o desperdício de memória inicial, pois a estrutura cresce e diminui sob demanda.

7. O que é uma lista simplesmente encadeada? Apresente um diagrama para ilustrar essa estrutura de dados.

Uma lista simplesmente encadeada (ou lista ligada) é uma estrutura de dados linear composta por uma sequência de **nós**. Cada nó é uma estrutura que contém duas informações:

1. O **dado** (a informação a ser armazenada).
2. Um **ponteiro** (prox) que aponta para o **próximo** elemento da lista.

A lista só pode ser percorrida em uma direção (do início ao fim). O último nó da lista aponta para um valor nulo (NULL), indicando o final da sequência.

- Ilustração (Diagrama Textual):
  [Início] -> [Nó A | prox_B] -> [Nó B | prox_C] -> [Nó C | NULL]

8. O que é uma lista duplamente encadeada? Apresente um diagrama para ilustrar essa estrutura de dados.
(Esta resposta também se aplica à questão 9, que está duplicada no seu documento).
Uma lista duplamente encadeada é uma evolução da lista simples onde cada nó armazena **três informações**:

1. O **dado** (a informação).
2. Um ponteiro (prox) que aponta para o **próximo** nó.
3. Um ponteiro (ant) que aponta para o nó **anterior**.

Essa estrutura de ponteiro duplo permite que a lista seja percorrida em **ambas as direções** (para frente e para trás), facilitando operações como inserção ou remoção em posições arbitrárias.

- Ilustração (Diagrama Textual):
  [Início] -> [Nó A: NULL | Dado A | prox_B] <-> [Nó B: ant_A | Dado B | prox_C] <-> [Nó C: ant_B | Dado C | NULL]

10. Explique o funcionamento do algoritmo de busca binária e sequencial.

- **Busca Sequencial (Linear):** É um método de busca simples que não exige que os dados estejam ordenados. O algoritmo percorre o vetor **elemento por elemento**, começando da primeira posição (índice 0). Em cada posição, ele compara o elemento com o valor procurado. A busca termina se o elemento for encontrado ou se o final do vetor for atingido (indicando que o valor não existe).

- **Busca Binária:** É um algoritmo de busca muito mais eficiente, mas que **exige que o vetor esteja previamente ordenado**. Ele funciona pela estratégia de "dividir para conquistar":
  1. O algoritmo compara o valor procurado com o elemento no **meio** do vetor.
  2. Se forem iguais, a busca termina.
  3. Se o valor procurado for **menor** que o do meio, o algoritmo descarta toda a metade direita (superior) do vetor e repete o processo, buscando apenas na metade esquerda.
  4. Se o valor procurado for maior, ele descarta a metade esquerda (inferior) e busca na metade direita.
     A cada passo, o espaço de busca é reduzido pela metade, tornando-o muito rápido para grandes volumes de dados.

11. Explique o funcionamento dos seguintes algoritmos de ordenação: Insertion sort, Selection sort, Merge sort, Count sort, Quicksort.

- **Insertion Sort (Ordenação por Inserção):** Constrói a lista ordenada final um elemento de cada vez. Ele percorre o vetor e, para cada elemento, o "insere" na posição correta dentro da porção do vetor que já está ordenada (à sua esquerda). É eficiente para listas pequenas ou quase ordenadas.
- **Selection Sort (Ordenação por Seleção):** É um algoritmo simples que, a cada passo, "seleciona" o menor elemento da porção ainda não ordenada do vetor. Em seguida, ele troca esse menor elemento com o elemento na primeira posição da porção não ordenada. O processo se repete, movendo a fronteira ordenada um elemento de cada vez.
- **Merge Sort (Ordenação por Intercalação):** Utiliza a estratégia "dividir para conquistar". Ele divide recursivamente o vetor em metades até que restem apenas sub-vetores de um único elemento (que são considerados ordenados). Em seguida, ele "intercala" (merge) esses sub-vetores, combinando-os de volta em vetores maiores, de forma ordenada, até que o vetor original esteja completo e ordenado.
- **Counting Sort (Ordenação por Contagem):** É um algoritmo não comparativo (não compara elementos entre si). Ele é usado para ordenar inteiros dentro de um intervalo conhecido. Seu funcionamento baseia-se em **contar** a frequência (o número de ocorrências) de cada valor no vetor de entrada. Com essas contagens, ele calcula a posição final de cada elemento no vetor ordenado.
- **Quicksort (Ordenação Rápida):** Também é um algoritmo "dividir para conquistar". Ele funciona escolhendo um elemento do vetor como **"pivô"**. Em seguida, ele **particiona** o restante do vetor, movendo todos os elementos menores que o pivô para a sua esquerda e todos os maiores para a sua direita. Finalmente, o algoritmo é chamado recursivamente para ordenar as duas sub-listas (a da esquerda e a da direita).