ADC 4

Pedro Henrique Lima Corrêa – 230023350

Engenharia de Computação – 3º Semestre

1. Nesse caso pode ser utilizada uma estrutura de dados chamada de Tabela *Hash*(ou tabela de espalhamento). Que pode ser implementada da seguinte forma: Como já se sabe a quantidade fixa de convidados, cria-se um vetor de listas ligadas. Cada convidado recebe um código, utiliza-se esse código para determinar a posição no vetor que esse convidado deve ser inserido, por meio de uma função *Hash*(que usualmente retorna o resto da divisão inteira entre o código e o tamanho do vetor). Após determinar a posição, insere-se um nó naquela posição. O primeiro nó representa o convidado, para cada amigo do convidado utiliza-se o mesmo código, então ocorre o que é chamado de colisão ao tentar inserir o amigo numa posição já ocupada pelo convidado, para resolver esse problema, insere-se um novo nó que representa o amigo do convidado, apontado pelo nó anterior. Assim para cada posição haverá uma lista de pelo menos um nó, que representa o convidado, e os nós seguintes que representam os amigos desse convidado. Assim ao buscar pelo código de um convidado, encontra-se a lista ligada que representa esse convidado e seus demais amigos.

2. Uma lista simplesmente ligada é uma estrutura de dados dinâmica(isto é, pode aumentar ou diminuir de tamanho), constituída de nós. Cada nó possui pelo menos dois campos: um para o valor armazenado nesse nó, e um ponteiro para o próximo nó. Cada nó aponta para a próxima posição da lista. É possível inserir e remover nós em qualquer posição na lista, porém, geralmente para esse fim, é usado um apontador para o final da lista. A lista ligada é a base para as outras estruturas de dados.
   Uma pilha funciona como uma lista ligada: constituída de nós, cada nó apontando para o próximo. A diferença está nas regras de inserção e remoção. Uma pilha deve obedecer a regra LIFO(*last-in first-out*), ou seja, o primeiro elemento a entrar é o último elemento a sair. Para isso, utiliza-se um ponteiro para o topo da pilha, cada inserção e remoção é feita na posição topo, apontada pelo ponteiro.
   Do mesmo modo, a fila é baseada na lista ligada, porém esta é do tipo FIFO(*first-in first-out*), o primeiro a entrar é o primeiro a sair, da mesma forma que uma fila real funciona. Utiliza-se um ponteiro para o fim e um ponteiro para o início. As inserções são feitas no fim da lista, e as remoções no começo.

3. O custo de tempo de uma busca binária em um vetor ordenado e em uma árvore de busca binária balanceada, é de O(log n), em que n é número de nós. A busca binária em um vetor ordenado funciona da seguinte forma: procura-se o valor desejado no meio do vetor, caso o valor desejado não for encontrado divide-se o vetor na metade, se o valor do meio for maior que o valor desejado, repete-se o procedimento para a metade menor, caso contrário, para a metade maior. Assim a

cada iteração, metade do vetor é desconsiderado, tornando esse processo muito mais eficiente que a busca sequencial.
Naturalmente, a busca em uma árvore de busca binária, ocorre da mesma forma. Cada nó da árvore pode apontar para a esquerda ou para a direita, sendo o valor da esquerda menor que o valor atual, e o valor da direita maior. Dessa forma, é realizado o passeio pela árvore: Caso o valor atual seja maior que o valor desejado, passeia para o nó à esquerda, caso contrário, para o nó à direita. Assim a cada iteração, metade dos nós da árvore são desconsiderados, sendo esse um processo de busca eficiente.

4. No caso apresentado, para implementar uma Tabela *Hash*, cada dado de entrada deve produzir uma chave por meio de uma função *Hash*, nessa tabela estariam armazenadas as respostas. Assim ao inserir um dado, a função é chamada, retornando uma chave, que representa o índice da tabela em que a resposta está armazenada, sendo então possível, realizar um acesso direto que teria custo de O(1). Sendo assim, a resposta seria praticamente imediata.

5. Um algoritmo eficiente de ordenação, por exemplo, é o *Quick Sort*, que funciona da seguinte forma: Um elemento da lista é escolhido como pivô, a lista é reorganizada para que todos os elementos anteriores ao pivô sejam menores ou iguais a ele, e todos os elementos à sua frente, sejam maiores, esse processo é chamado de particionamento. Após isso, recursivamente, as sub-listas abaixo e acima do pivô são ordenadas. O *Quick Sort* possui complexidade de pior caso $O(n^2)$. Apesar disso, na prática, ele é considerado o algoritmo de ordenação mais rápido, sendo a complexidade de melhor e médio caso O(n log n). Ao ser implementado, são utilizadas estratégias para minimizar as chances de ocorrência do pior caso.
   Um exemplo de algoritmo de ordenação ineficiente é o *Selection Sort*. Possui complexidade de $O(n^2)$ em todos os casos, pois esse algoritmo sempre percorre a lista n^2 vezes, sendo n o número de elementos na lista. Funciona da seguinte forma: Percorre a lista e seleciona o menor item e o coloca na primeira posição, percorre a lista novamente, seleciona o segundo menor item e o põe na segunda posição, e assim sucessivamente. Geralmente é utilizado em listas pequenas, em situações em que a eficiência de tempo não é tão importante. Porém se fosse utilizado para ordenar uma lista muito grande, levaria muito tempo, tornando o programa em que fosse implementado extremamente lento.