Exercício 3 – Estrutura de Dados

Nome: Breno Poggiali de Sousa
Matrícula: 2018054800

Ex. 1)

Método	Breve descrição	Vantagens	Desvantagens	Complexidade (caso médio)
Bolha	Passa no arquivo e troca elementos adjacentes que estão fora de ordem, até os registros ficarem Ordenados.	Algoritmo simples Algoritmo estável	Não adaptável em termos de comparações Muitas trocas de itens	$O(n^2)$
Seleção	Seleção do n-ésimo menor (ou maior) elemento da lista e troca esse n-ésimo menor (ou maior) elemento com a n-ésima posição da lista Uma única troca é realizada por vez.	Números de movimentos → O(n) Bom para arquivos muito grandes com número de elementos a ordenar pequenos	Não adaptável (não importa se o arquivo está ordenado). Algoritmo não é estável.	$O(n^2)$
Inserção	Parecido com o que o jogador de baralho faz: Cartas ordenadas da esquerda para a direita. Escolhe a carta e verifica se ela deveria estar antes ou onde está – caso devesse estar antes, ela vai para a posição que deveria no array que a antecede. Isso se repete até o final, da esquerda para direita.	É o mais interessante para números pequenos Para arquivos ordenados → O(n) Implementação simples Estável	Alto custo de movimentações de elementos no vetor	$O(n^2)$
Quick Sort	A idéia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores. Os problemas menores são ordenados independentemente. Os resultados são combinados para produzir a solução final	Mais eficiente em diversas situações Caso médio com O(n log(n)) Necessita apenas de uma pequena pilha como memória auxiliar Pior caso possui uma probabilidade muito pequena de ocorrer	Variação pior caso realiza $O(n^2)$ operações Implementação delicada e difícil Método não é estável.	O(n log(n))
Mergesort	Método dividir para conquistar baseado em merging (intercalação) Combinação de dois vetores ordenados em um vetor maior que também já está ordenado	Seu custo é sempre O(n log(n)) Não varia com entrada É estável Espaço extra de memória necessário é proporcional a n	Requer espaço extra proporcional a n.	O(n log(n))
Heapsort	Estrutura de dados composta de chaves, que suporta duas operações principais: inserção de um novo item e remoção do item com a maior chave A chave de cada item reflete a prioridade em que se deve tratar aquele item	Método elegante e eficiente Não necessita de memória adicional Executa sempre em tempo proporcional a O(n log(n))	O anel interno do algoritmo é bastante complexo quando comparado com o do Quicksort Não é estável	O(n log(n))
Shellsort	Extensão do algoritmo de ordenação por inserção Permite que itens separados por h posições sejam reanjados. Todo h-ésimo item leva a uma sequência ordenada. Quando h=1, o algoritmo é equivalente ao de inserção	Bom para ordenar um número moderado de elementos Bom para aplicações que não podem tolerar variações no tempo esperado de execução Quando encontra um arquivo parcialamente ordenado trabalha menos Implementação simples	O tempo de execução é sensível à ordem inicial da entrada Não é estável	$O(n \log(n)^2)$
Counting sort	A idéia básica é determinar, para cada entrada de x, o número máximo de elementos menor que x. Isso pode ser usado para colocar o elemento x diretamente em sua posição na saída.	É estável. Possui complexidade linear	Necessita de memória auxiliar, logo, não é in-place	O(n)
Bucket sort	Divide o vetor em um número finito de recipientes, onde cada recipiente é ordenado individualmente. Ao final, juntam-se os vetores e tem o vetor ordenado	Em seu caso médio, sua complexidade é linear.	No seu pior caso, sua complexidade é $O(n^2)$	O(n)
Radix sort	Esse algoritmo ordena elementos processando seus bits. Da esquerda para direita (mais para menos significativo), ele vai selecionando os maiores e os menores bits para ordena-los.	É estável. Possui complexidade linear, mas maior do que das últimas 2 variações.	Sua implementação é mais complexa	O(n)

Ex. 2)

A) BOLHA:
[i = 0] 0 1 2 0 5 4 8 0 0 8
[i = 1] 0 1 0 2 4 5 0 0 8 8
[i = 2] 0 0 1 2 4 0 0 5 8 8
[i = 3] 0 0 1 2 0 0 4 5 8 8
[i = 4] 0 0 1 0 0 2 4 5 8 8
[i = 5] 0 0 0 0 1 2 4 5 8 8
[i = 6] 0 0 0 0 1 2 4 5 8 8
[i = 7] 0 0 0 0 1 2 4 5 8 8
[i = 8] 0 0 0 0 1 2 4 5 8 8

C) INSERÇÃO:
[i = 0] 0 2 1 8 0 5 4 8 0 0
[i = 1] 0 1 2 8 0 5 4 8 0 0
[i = 2] 0 1 2 8 0 5 4 8 0 0
[i = 3] 0 0 1 2 8 5 4 8 0 0
[i = 4] 0 0 1 2 5 8 4 8 0 0
[i = 5] 0 0 1 2 4 5 8 8 0 0
[i = 6] 0 0 1 2 4 5 8 8 0 0
[i = 7] 0 0 0 1 2 4 5 8 8 0
[i = 8] 0 0 0 0 1 2 4 5 8 8

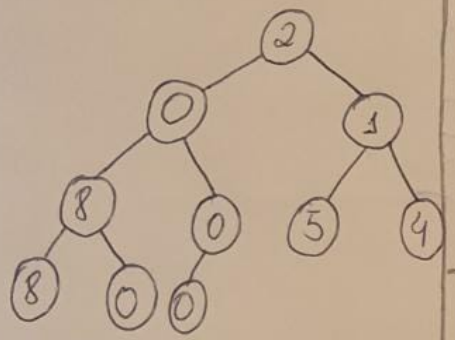
B) SELEÇÃO:
[i = 0] 0 2 1 8 0 5 4 8 0 0
[i = 1] 0 0 1 8 2 5 4 8 0 0
[i = 2] 0 0 0 8 2 5 4 8 1 0
[i = 3] 0 0 0 0 2 5 4 8 1 8
[i = 4] 0 0 0 0 1 5 4 8 2 8
[i = 5] 0 0 0 0 1 2 4 8 5 8
[i = 6] 0 0 0 0 1 2 4 8 5 8
[i = 7] 0 0 0 0 1 2 4 5 8 8
[i = 8] 0 0 0 0 1 2 4 5 8 8

D) QUICKSORT:

pivot = 0; subvector = 0 0 0 8 1 5 4 8 0
pivot = 0; subvector = 0 0
pivot = 4; subvector = 2 1 0 4 8 5
pivot = 1; subvector = 0 1
pivot = 5; subvector = 5 8
pivot = 8; subvector = 8

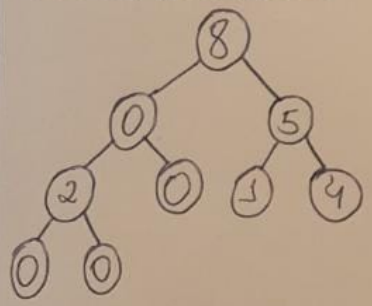
E) Heapsort

Heap inicial:



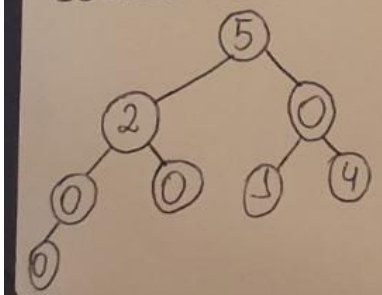
Após 1ª retirada

_ _ _ _ _ _ _ _ _ 8



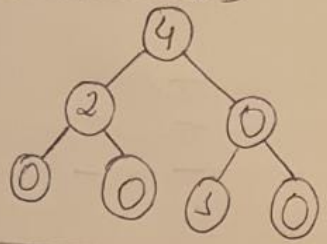
Após 2ª retirada

_ _ _ _ _ _ _ 8 8



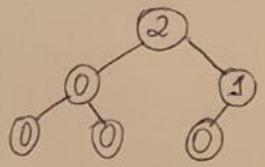
Após 3ª retirada

_ _ _ _ _ _ 5 8 8



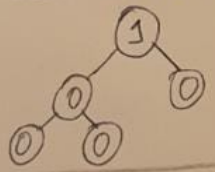
Após 4ª retirada

_ _ _ _ _ 4 5 8 8



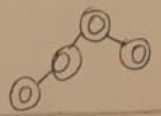
Após 5ª retirada

_ _ _ _ 2 4 5 8 8



Após 6ª retirada

_ _ _ _ 1 2 4 5 8 8



Após 7ª retirada

_ _ _ 0 1 2 4 5 8 8

Após 8ª retirada

_ _ 0 0 1 2 4 5 8 8

Após 9ª retirada

_ 0 0 0 1 2 4 5 8 8

Após 10ª retirada

0 0 0 0 1 2 4 5 8 8