**RESUMO**

**TADS**

**Tipos Abstratos de dados** visam desvincular o tipo de dado(estrutura de dados e operacoes que as manipulam) de sua implementacao. Quando definimos um Tipo Abstrato de Dados estamos preocupados com O QUE ele faz(especificacao) e nao COMO ele faz(implementacao). Basicamente escondemos os dados e detalhes do usuario, fornecendo apenas uma interface publica(metodos/operacoes) para manipula-los.

No arquivo de especificacao(.h) colocamos a assinatura e a semantica, ja no arquivo de implementacao(.c) colocamos a estrutura de dados e os algoritmos.

A vantagem de usar TAD seria a reutilizacao, facilidade de manutencao ja que mudancas no arquivo de implementacao nao afetam o codigo fonte dos programas que utilizam

Anotacoes das Aulas

Todos os TADS precisam de um contador pra controlar o tamanho, diminuindo o custo e aumentando a eficiencia.

Se uma funcao for excencial pro programa, nao podemos retornar ao programa, eh necessario finaliza-lo

Utilizamos um prefixo “is” no nome de toda funcao bool por convensao

Ao cortar/dividir tads voce pode acabar perdendo referencias

Na funcao DESTROY limpamos o ponteiro e depois a estrutura inteira, pois se destruisse apenas a estrutura ainda ficariam casas as quais nao seria desalocadas

CALLOC vs MALLOC:

Ambos serve para alocar memoria durante a execucao do programa fazendo o pedido de memoria ao computador e retornando um ponteiro com o endereco de onde comeca o espaco de memoria alocado, no entanto, o calloc inicializa todos os bits dos espacos alocados com zero, enquando o malloc utiliza o lixo da memoria. Porem o calloc eh menos performavel comparado ao malloc.

**BUUBLE SORT**

Eh um algoritmo de ordenacao que se baseia na ideia de comparar repetidamente pares de elementos adjacentes, em seguida, trocar suas posicoes caso um seja maior que o outro.

Eh um algoritmo de ordenacao de facil entendimento e implementacao, no entanto, nao eh um algoritmo eficaz pois sua eficiencia diminui radicalmente conforme aumentamos o numero de elementos do array.

Seu melhor caso eh quando os elementos ja estao ordenados

Ja o pior caso eh quando os elementos estao ordenados em ordem inversa

**SELECTION SORT**

Eh um algoritmo que divide o array em duas partes :

A parte Ordenada que fica a esquerda do elemento analisado e a parte que ainda Nao Foi Ordenada que fica a direita desse elemento. Pra cada elemento do array ele procura o menor valor a sua direita e troca os dois valores de lugar.

Por esse algoritmo SEMPRE comparar um elemento com os outros em cada iteracao procurando o menor, Nao Existe um Melhor Caso, pq mesmo o vetor estando ordenado ou inverso, sempre vai ser executado os dois lacos do algoritmo. Tambem sendo muito lento pra vetores grandes

No entanto ele ocupa menos memoria pois nao necessita de um vetor auxiliar pra realizar a ordenacao sendo extremamente rapido em vetores pequenos

**INSERTION SORT**

Esse algoritmo percorre o array e pra cada posicao X, ele verifica se o valor esta na posicao correta. Isso eh feito andando pro comeco do array partindo da posicao X e movimentando uma posicao pra frente os valores que sao maiores do que o valor da posicao X. Dessa forma vamos ter uma posicao livre pra inserir o valor da posicao X em seu devido lugar

Melhor Caso com os elementos ja ordenados

Pior caso os elementos ordenados em ordem inversa tendo em vista q eh feito uma analise de elemento em elemento antes de inserir o elemento em uma posicao

**MERGE SORT**

O merge divide o array, recursivamente, em duas partes, ate que cada posicao dele seja considerada um array de um unico elemento. Em seguida ele combina dois arrays de forma a obter um array maior e ordenado. Essa combinacao dos arrays eh feita intercalando seus elementos de forma crescente. Esse processo se repete ate que exista apenas um array

A eficiencia desse algoritmo nao depende da ordem dos elementos, ele possui um gasto extra de espaco de memoria em relacao aos demais metodos de ordenacao. Isso acontece pq ele cria uma copia do array pra cada chamada recursiva.

No Pior Caso o merge realiza cerca de 39% menos comparacoes do que o quick faz no seu caso medio.

Ja o Melhor Caso, o merge realiza cerca de metade do numero de iteracoes do seu pior caso

**QUICK SORT**

Esse algoritmo, primeiramente, rearranja os valores do array de modo que os valores menores do que o valor do PIVO fiquem na parte esquerda do array, enquanto os valores maiores que o PIVO ficam na parte direita. Em seguida o algoritmo eh aplicado recursivamente a cada particao. O processo se repete ate que cada particao contenha um unico elemento

Melhor caso eh quando as particoes possuem sempre o mesmo tamanho, ou seja, estao sempre balanceada

Pior Caso ocorre quando a funcao de particao calcula um pivo que divide o array de N elementos em dois: uma particao com N - 1 elementos e outra com 0 elementos. Nesse caso teremos particionamento nao balanceado. Quando isso acontece a cada nivel da recursao temos um tempo de execucao O(n^2)