Trabalho Prático 1 – TAD Ordenador Universal

Estrutura de Dados

Matheus Soares dos Santos de Freitas

masanfreitas@outlook.com

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte

2025

ÍNDICE

1 INTRODUÇÃO

Este trabalho apresenta a implementação de uma prova de valor para o Tipo Abstrato de Dados (TAD) Ordenador Universal, um componente projetado para selecionar dinamicamente o algoritmo de ordenação mais eficiente com base nas características do vetor a ser ordenado. A proposta visa explorar critérios simples, porém eficazes, para otimizar o desempenho da ordenação, combinando a lógica de limiares ótimos com o conceito de “quebras” — isto é, as descontinuidades em vetores quase ordenados.

Nesta etapa inicial, o Ordenador Universal foi construído com base no relacionamento entre dois algoritmos: insertion sort e quick sort. A seleção entre eles é feita considerando dois fatores principais: o grau de ordenação do vetor (medido pelo número de quebras) e o seu tamanho, com uso da técnica de mediana de três para mitigar ineficiências no quicksort.

Esta documentação detalha o processo de desenvolvimento do TAD, descreve os critérios adotados para a escolha entre os algoritmos e apresenta os resultados obtidos por meio de análises experimentais.

2 IMPLEMENTAÇÃO

O código-fonte foi estruturado de forma modular, visando organização e facilidade de manutenção. A pasta include contém os arquivos de especificação, incluindo cabeçalhos e templates: estatisticas.hpp, funcoes.hpp, funcoes.tpp, ordenador.hpp e ordenador.tpp. Já a pasta src reúne os arquivos de implementação: estatisticas.cpp e main.cpp. No diretório raiz, há um Makefile responsável por automatizar o processo de compilação. Além disso, o projeto inclui a pasta .vscode, com arquivos de configuração no formato JSON para o ambiente de desenvolvimento Visual Studio Code. Os arquivos objeto gerados (.o) são armazenados em .obj, e a pasta bin guarda o executável final (tp1.out).

2.1 Estruturas de dados

A implementação se apoia em duas classes principais:

- Classe Estatisticas: responsável por registrar dados relevantes durante a execução dos algoritmos de ordenação, como o número de chamadas, comparações e movimentações. Também armazena o número de quebras no vetor e os custos de cada ordenação, permitindo posterior análise de desempenho.
- Classe OrdenadorUniversal: implementada como um template para garantir compatibilidade com diferentes tipos de dados. Contém os métodos responsáveis pela ordenação dos vetores e pela determinação dos limiares ótimos de partição e de quebra.

2.2 Funcionamento do programa

O programa principal, implementado em main.cpp, inicia-se com a leitura de um arquivo de entrada, cujos parâmetros são passados por linha de comando (via argc/argv). Esses parâmetros incluem: semente aleatória, valor arbitrário de limiar de custo, coeficientes da função de custo e o tamanho do vetor a ser ordenado. Após a leitura, os dados são armazenados em um vetor, e um objeto da classe Estatisticas é instanciado para registrar a quantidade de quebras e a semente utilizada, além de imprimir essas informações na tela, junto ao tamanho do vetor.

Na sequência, o programa determina o limiar de partição — valor a partir do qual o uso do quick sort torna-se mais vantajoso que o insertion sort. Esse processo é realizado pelo método **determinaLimiarParticao()** da classe OrdenadorUniversal, que realiza uma busca refinada em torno dos limiares candidatos, com base no custo total observado.

De forma análoga, o método **determinaLimiarQuebra()** é utilizado para encontrar o limite ideal de quebras — ou seja, o número de descontinuidades no vetor abaixo do qual o insertion sort é preferível.

Durante essas buscas, o módulo de funções auxilia com métodos de suporte, como:

- **menorElemento()**: encontra o menor custo (ou a menor diferença de custo) em faixas de valores candidatos;
- **median()**: calcula a mediana de três elementos para o quicksort otimizado;
- **swap()**: realiza trocas de elementos no vetor.

2.3 Configuração de testes

O programa foi desenvolvido em C++, utilizando o compilador G++ da GNU Compiler Collection. Embora o sistema operacional do computador seja o Windows, a execução do código foi feita por meio do Windows Subsystem for Linux (WSL), garantindo um ambiente compatível com sistemas baseados em Unix.

As especificações da máquina de testes são as seguintes:

- Processador: Intel(R) Core(TM), 1 socket, 6 núcleos, 6 threads, com virtualização habilitada e frequência base de 2,90 GHz;
- Cache: L1 (384 KB), L2 (1,5 MB) e L3 (9 MB);
- Memória RAM: 9,2 GB em uso, com 2,7 GB disponíveis no momento da execução dos testes.

3 INSTRUÇÕES DE COMPILAÇÃO E EXECUÇÃO

Para compilar e executar o projeto, siga os passos abaixo:

1. Acesse o diretório do projeto, geralmente localizado em **~/TP1**.
2. Abra um terminal dentro desse diretório e utilize o comando **make all** para compilar todos os arquivos.
3. Para rodar o programa principal com um arquivo de entrada, execute o comando **./bin/tp1.out arquivo.txt**, considerando que o arquivo esteja na mesma pasta do executável.

4 ANÁLISE DE COMPLEXIDADE

As análises serão divididas entre 3 importantes blocos: as funções e métodos de domínio do Ordenador Universal (ordenador.tpp), do módulo de funções (funcoes.cpp) e do programa principal (main.cpp):

ordenador.tpp:

Função ordenadorUniversal(): Decide se usa InsertionSort ou QuickSort baseado em um limiar de quebras e no tamanho da partição. Sua complexidade depende dos algoritmos escolhidos (InsertionSort ou QuickSort).

- ➢ Complexidade de tempo: Se usa InsertionSort, Complexidade $O(n^2)$ no pior caso. Se usa QuickSort, a complexidade média será $O(n \log n)$, e, no pior caso, $O(n^2)$ (mas a partição de 3 e a escolha da mediana devem melhorar para o caso médio).
- ➢ Complexidade de espaço: Faz uso de pilha na recursão do QuickSort; assim, no pior caso (recursão desbalanceada), pode ser $O(n)$, mas geralmente é $O(\log n)$.

Função insertionSort(): Faz ordenação por inserção clássica.

- ➢ Complexidade de tempo: Como essa função é usada para subarrays pequenos (tamanho menor que o limiar de partição), o impacto prático no desempenho costuma ser pequeno, mas, no geral, ainda será $O(n^2)$ para vetores inversamente ordenados e $O(n)$ para vetores quase ordenados.
- ➢ Complexidade de espaço: A ordenação é feita in-place, isto é, sem o uso de memória extra significativa, acarretando custo constante de memória O(1).

Função quickSort(): Implementa QuickSort com partição de 3 e chamada recursiva e, para partições menores que um limiar, troca para InsertionSort.

- ➢ Complexidade de tempo: Em média, será $O(n \log n)$, devido à divisão equilibrada. No entanto, quando a partição for altamente desbalanceada, será $O(n^2)$.
- ➢ Complexidade de espaço: Sua versão recursiva faz uso de pilha; assim, no pior caso (recursão desbalanceada), pode ser $O(n)$, mas geralmente é $O(\log n)$.

Função partition3(): Executa a partição de 3 (divide o vetor em três partes: menor que pivô, igual ao pivô e maior que pivô).

- ➢ Complexidade de tempo: $O(n)$ para particionar o array inteiro uma vez.
- ➢ Complexidade de espaço: Faz particionamento in-place, isto é, sem uso extra significativo de memória. Dessa forma, o custo é constante $O(1)$.

Função embaralhar(): Embaralha o vetor trocando pares aleatórios com *numShuffle* trocas.

- ➢ Complexidade de tempo: O(numShuffle), normalmente proporcional a um valor menor que n.
- ➢ Complexidade de espaço: A troca de elementos ocorre de forma in-place, com memória extra constante, justificando o custo $O(1)$.

Função determinaLimiarParticao(): Procura empiricamente o limiar ótimo para a partição (mínimo tamanho para trocar para InsertionSort), através de várias ordenações em subvetores com tamanhos variados (5 pontos na faixa, repetidamente reduzindo a faixa). Em cada iteração, são executadas as ordenações e o cálculo de seus respectivos custos.

- ➢ Complexidade de tempo: Muito cara — em ordem de $O(k*n \log n)$, onde k é o número de iterações e ordenações feitas para testar limiares diferentes. Esse é um processo de calibração, não parte da ordenação em si.
- ➢ Complexidade de espaço: Pelo menos $O(n)$, pois mantém várias cópias do vetor para testes. Além disso, usa memória extra para criar subarrays e embaralhá-los.

Função determinaLimiarQuebra(): Procura empiricamente o limiar de quebras para troca entre InsertionSort e QuickSort, baseado no custo diferencial entre os dois métodos. Semelhante à função anterior, executa múltiplas ordenações e cálculo de custo, com o diferencial dos embaralhamentos.

- ➢ Complexidade de espaço: Também bastante custosa computacionalmente — da ordem de múltiplas execuções de ordenações em $O(n \log n)$ ou $O(n^2)$, conforme tamanho do vetor e limiares testados.
- ➢ Complexidade de tempo: Pelo menos $O(n)$, similarmente à determinaLimiarParticao(), devido a múltiplas cópias temporárias.

Função calculaNovaFaixa(): Ajusta os limites de busca para limiares na função de calibração.

- ➢ Complexidade de tempo: O custo é constante $O(1)$, pois os cálculos realizados se resumem a aritmética, sem percorrimento de vetores.
- ➢ Complexidade de espaço: $O(1)$, em razão do uso constante de memória para as variáveis auxiliares.

funcoes.cpp:

Função median(): Retorna a mediana entre três elementos.

- ➢ Complexidade de tempo: Constante $O(1)$, pois faz uma série de comparações simples entre os valores.
- ➢ Complexidade de espaço: Da mesma forma, constante $O(1)$, pois usa apenas variáveis locais e retorna um valor.

Função swap(): Realiza a troca de posições entre dois elementos de um vetor qualquer.

- ➢ Complexidade de tempo: Custo constante $O(1)$, pois realiza apenas 3 operações de acesso ao vetor (leitura/escrita) e chamadas constantes para incremento das movimentações contabilizadas.
- ➢ Complexidade de espaço: O custo é constante $O(1)$, já que a função apenas faz uso de uma variável temporária local (*temp*).

Função menorElemento(): Retorna o índice do menor elemento de um vetor genérico.

- ➢ Complexidade de tempo: Como percorre todo o vetor para encontrar o menor elemento, a complexidade é $O(n)$, onde n=*tam*.
- ➢ Complexidade de espaço: $O(1)$, pois faz uso de apenas duas variáveis locais (*menor* e *idmenor*).

main.cpp:

O programa principal se baseia em etapas, com a leitura dos n elementos do arquivo e a alocação dinâmica destes num vetor tendo custo O(n) tanto em tempo quanto em complexidade. A contagem de quebras (**countBreaks()**) é um método de Estatisticas chamado no código e com custo de tempo $O(n)$ - percorre o vetor uma única vez - e de espaço $O(1)$. As funções **determinaLimiarParticao()** e **determinaLimiarQuebra()**, conforme anteriormente elucidado, possuem custo $O(k*n \log n)$. Assim, a respeito de custo em tempo, o termo $O(k*n \log n)$ domina. Já o uso total de memória está compreendido em $O(n)$.

5 ANÁLISE EXPERIMENTAL

A análise experimental avaliou o comportamento do TAD Ordenador Universal sob diferentes configurações de vetores, considerando a variação no tamanho, nas chaves, no payload e no grau de ordenação (ordenado, parcialmente ordenado ou inversamente ordenado, conforme o número de quebras). Os experimentos foram realizados em C++, e os resultados foram registrados em arquivos .csv. A análise dos dados e a geração dos gráficos foram feitas em Python, por meio de notebooks Jupyter.

5.1 Premissas

A análise de desempenho dos algoritmos de ordenação foi baseada na premissa de que o tempo de execução pode ser modelado como função Custo() que consiste numa combinação linear de fatores como o número de chamadas recursivas, movimentações e comparações, isto é,

$$\textbf{tempo} \approx \textbf{a}\cdot\textbf{calls} + \textbf{b}\cdot\textbf{moves} + \textbf{c}\cdot\textbf{comp}$$

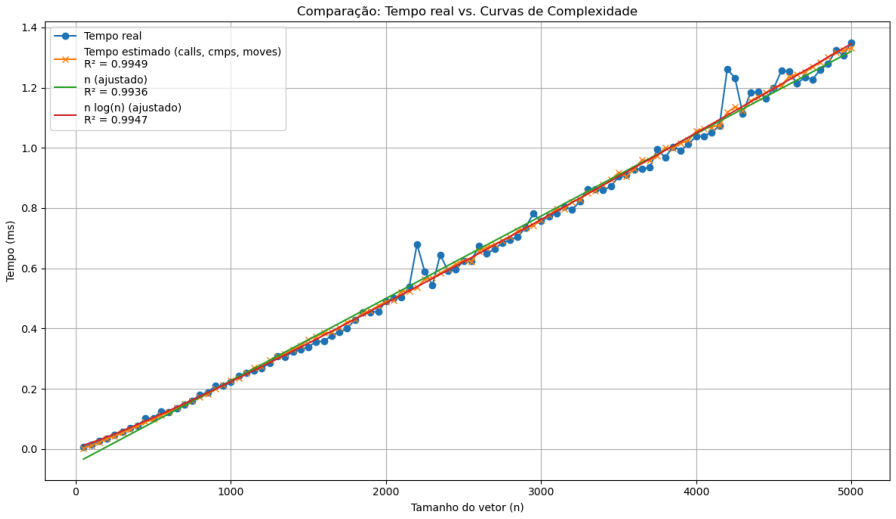

Figura 1: Comparação entre a função Custo() e o tempo real de execução do programa

A análise dos resultados, com um coeficiente de determinação $R^2 \approx 0{,}9949$, evidencia a eficácia da função Custo() em modelar o tempo de execução do programa. Observa-se ainda que a curva n*logn apresenta excelente aderência aos dados experimentais, reforçando a hipótese de que o algoritmo possui complexidade temporal na ordem de *O(n*logn)*.

5.2 Correlação: Payload X Tempo de Execução do TAD

A variação do *payload* — a informação útil armazenada em cada estrutura — também demonstra uma influência moderada e negativa sobre o tempo de execução, conforme ilustrado na Figura 2. Nesse contexto, uma regressão polinomial de segundo grau mostra-se adequada para descrever o comportamento dos dados no intervalo analisado. Contudo, o número limitado de amostras representa uma limitação que impede conclusões definitivas.

Uma hipótese plausível é que *payloads* maiores reduzam a sensibilidade do algoritmo às diferenças entre as chaves, concentrando o custo computacional nas comparações de

chave em si e minimizando acessos adicionais à memória ou movimentações desnecessárias dos dados auxiliares.

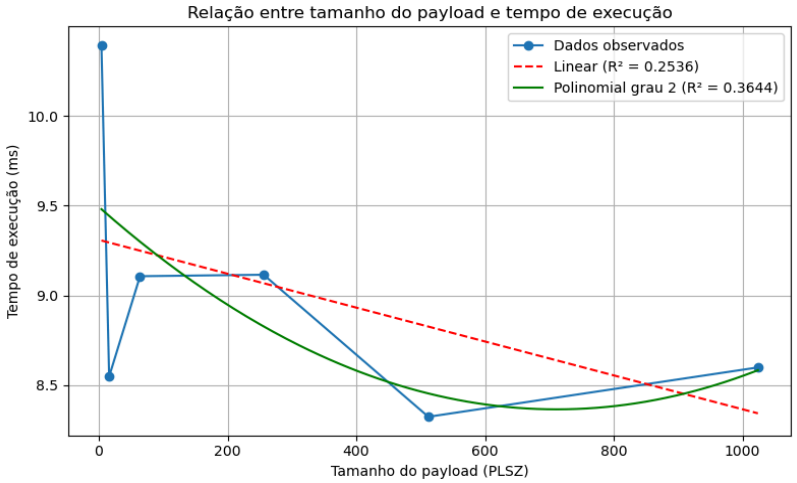

Figura 2: Relação entre a informação útil do vetor e o tempo de execução do programa

5.3 Correlação: Tamanho das chaves X Tempo de Execução do TAD

Observou-se uma correlação forte e negativa entre o tempo de execução do TAD e o tamanho das chaves no vetor de estruturas. Em outras palavras, o tempo de execução do Ordenador Universal tende a ser maior quando as chaves possuem tamanho reduzido. Isso sugere a hipótese de que tamanhos menores de chave podem intensificar colisões ou comparações mais custosas em termos relativos, devido à menor discriminação entre elementos, o que impacta negativamente o desempenho do algoritmo.

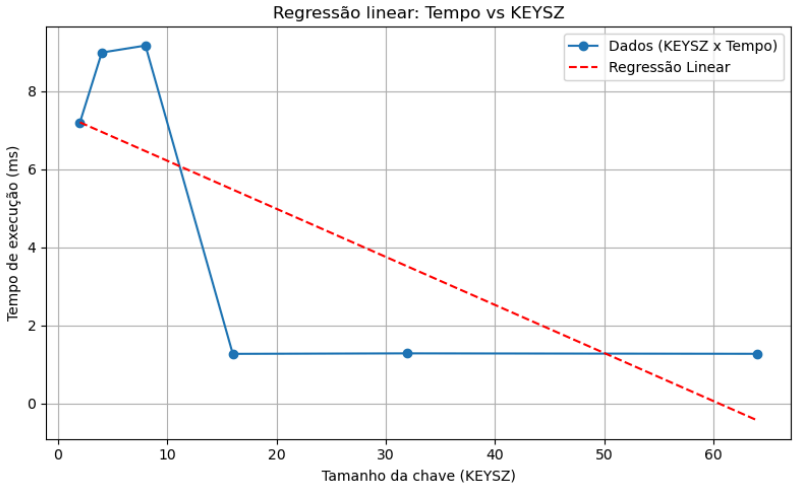

Figura 3: Relação entre o tempo de execução do TAD e o tamanho das chaves no vetor

5.4 Correlação: Número de quebras X Tempo de Execução do TAD

Observou-se uma correlação positiva entre o tempo de execução do programa e o número de quebras no vetor. Isso significa que, à medida que o vetor se torna menos ordenado — ou seja, quanto maior o número de segmentos fora de ordem — o tempo necessário para ordená-lo aumenta. Tal padrão pode ser observado na Figura 4.

Essa relação pode ser explicada pelo fato de que algoritmos de ordenação costumam aproveitar regiões já ordenadas para reduzir o número de comparações e movimentações. Quando há muitas quebras, o vetor se aproxima de uma configuração aleatória, exigindo mais trabalho do algoritmo para restaurar a ordenação global.

Uma hipótese plausível, portanto, é que o número de quebras funcione como um indicador de desordem interna no vetor, afetando diretamente a eficiência de estratégias de ordenação adaptativas, que tendem a performar melhor em dados parcialmente ordenados.

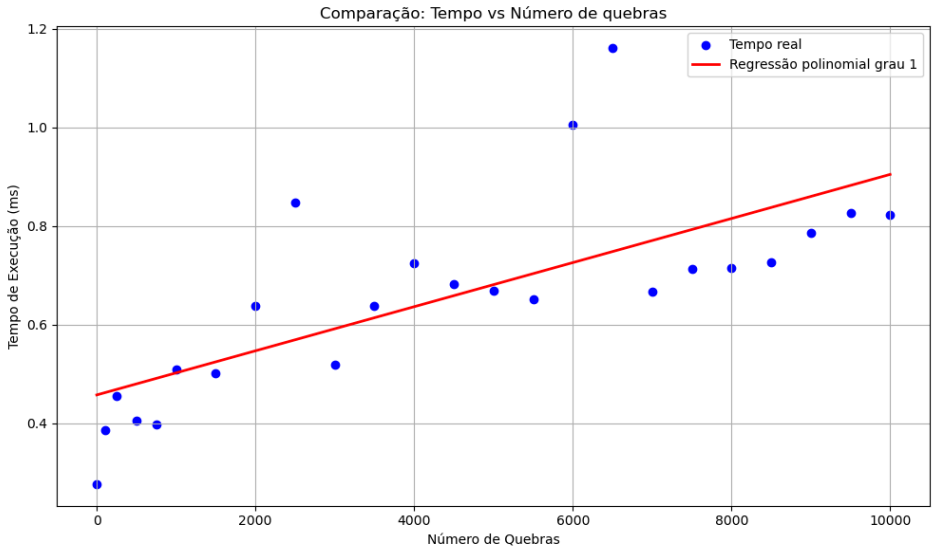

Figura 4: Correlação entre a ordenação inicial do vetor e o tempo de execução do TAD

5.5 Eficiência do TAD: Ordenador Universal X InsertionSort e QuickSort

Por fim, avaliou-se o desempenho do Ordenador Universal em comparação com os algoritmos QuickSort e InsertionSort em suas versões puras. Dada a disparidade nas ordens de grandeza dos tempos de execução, o gráfico correspondente foi construído em escala logarítmica. Os resultados podem ser visualizados na Figura 5.

As conclusões indicam que, como esperado, o InsertionSort torna-se rapidamente ineficiente à medida que o número de quebras no vetor aumenta, sendo viável apenas em vetores quase ordenados. O Ordenador Universal, por sua vez, apresenta ótimo desempenho nesse cenário de quase ordenação, superando os demais algoritmos.

No entanto, à medida que o vetor se torna mais desordenado, observa-se que o QuickSort passa a ter melhor desempenho do que o Ordenador Universal, embora ambos se mantenham próximos em termos absolutos.

Uma possível hipótese para esse comportamento é que os mecanismos de decisão do Ordenador Universal — que incluem verificações para detectar ordenação parcial e seleção de pivôs por mediana — introduzem uma sobrecarga computacional que deixa de ser compensada em vetores com alta desordem. Nesses casos, o QuickSort, com sua estrutura mais direta e adaptabilidade, acaba se beneficiando de sua simplicidade e eficiência média, superando o TAD em tempo de execução.

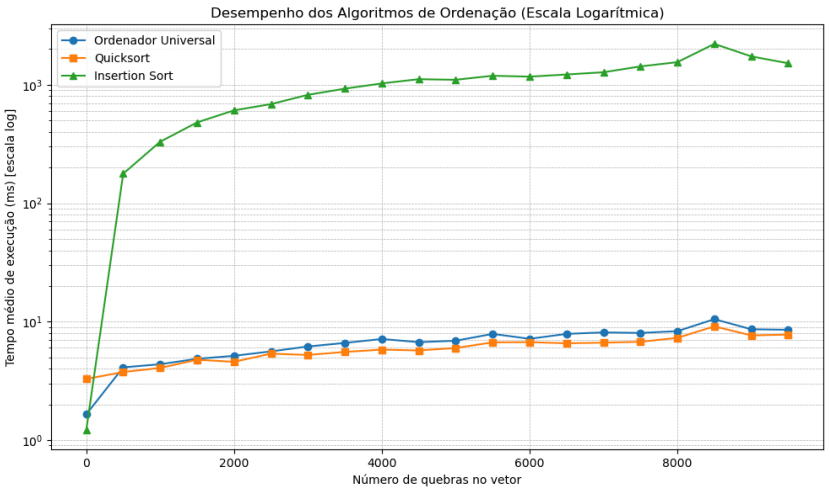

Figura 5: Desempenho do TAD contra outros algoritmos de ordenação

6 CONCLUSÃO

Este trabalho explorou a possibilidade de otimizar algoritmos de ordenação por meio da implementação de um Ordenador Universal, concebido como um TAD (Tipo Abstrato de Dados) com o objetivo de reduzir os custos de execução na ordenação de vetores. A partir dos experimentos realizados, foi possível demonstrar a eficácia do TAD, especialmente na ordenação de vetores quase ordenados, onde apresentou desempenho superior. À medida que a desordem no vetor aumentou, o custo de execução do TAD se mostrou comparável ao do QuickSort, confirmando sua adaptabilidade.

Ao longo do desenvolvimento, tive a oportunidade de aplicar na prática conceitos fundamentais como algoritmos de ordenação, análise de complexidade e custo computacional. Os principais desafios enfrentados estavam ligados à compreensão dos conceitos teóricos subjacentes — como os limiares de quebra e partição — e à internalização da lógica de execução dos algoritmos envolvidos.

7 BIBLIOGRAFIA

Referências

[1] Wikipedia Contribuitors. *Insertion sort*. Disponível em:
<https://pt.wikipedia.org/wiki/Insertion_sort>.

[2] Wikipedia Contribuitors. *Quicksort*. Disponível em:
<https://pt.wikipedia.org/wiki/Quicksort>.

[3] Especificação do Trabalho Prático 1 - TAD Ordenador Universal, DCC/ICEx/UFMG

[4] Anisio Lacerda, Wagner Meira Jr., Washington Cunha. Estruturas de Dados, Slides da disciplina DCC205 - Estruturas de Dados, Departamento de Ci^encia da Computa¸c˜ao, ICEx/UFMG.