CONTEÚDO DE AULA - ESTRUTURA DE DADOS I

Aula 08.25
[Apresentação da disciplina]

Página central de conteúdo da disciplina:
SSC-603-2020(FOsorio)

A matéria estrutura de dados é a base de conceitos para as matérias de computação. Ela tem como objetivo fornecer os saberes e ferramentas necessários ao aluno para que ele faça programas e sistemas que respondam de maneira rápida e eficiente, mesmo estando carregado com uma grande quantidade de dados.

Aula 08.27
[Revisão (ponteiros) + TAD - Tipos Abstratos de Dados]

→Os **TAD (Tipos Abstratos de dados)** podem também serem chamados de **ADT (Abstract Data Types)**. Eles são, basicamente, structs que ordenam dados aliadas a um conjunto de funções que manipulam esses dados, com intuitos diversos.

→Devemos ter cuidado com a compatibilidade entre o sistema operacional (32 ou 64 bits) e a natureza do compilador.

~Curiosidade~
A pasta "Arquivos de programas" (ou "Program Files"), do Windows, é convenientemente separada em programas 32 e 64 bits. Em um S.O. 64 bits, a pasta "Arquivos de programas" contém programas 64 bits e a "Arquivos de programas (x86)" contém programas 32 bits.

→O professor retomou algumas manipulações de ponteiros, usando * ou não, que já foram descritas no meu Drive de ICC I (https://docs.google.com/document/d/1IPYilK7ZeYaiNOPnlAAk0tDfglkRBI_6bUtnw3ZE9r8/edit).

→Normalmente, uma variável possui 3 atributos:

- Nome;
- Dado (número, no caso de int, caractere, no caso de char etc.);
- Endereço (na memória).

→Ao declarar um vetor e atribuí-lo a um ponteiro, sem especificar índice, o ponteiro apontará para a primeira posição dele. Porém, pode-se utilizar uma manipulação de soma:

- int vetor[10];

*(vetor+5) = 15;

Nesse caso, o endereço "vetor", sem índice, apontaria naturalmente para v[0]. Porém, como somamos 5 posições ao endereço, avançamos 5 inteiros na memória, o que seria equivalente ao v[5] (sexta posição).

→**Segmentation fault** é uma extrapolação de segmento. Esse segmento, por sua vez, seria o seu programa. Geralmente, quando temos um segmentation fault, quer dizer que houve algum erro de manipulação e o programa tentou invadir a memória de outro programa, o que faz com que ele seja abortado instantaneamente.

→Criar uma variável que não seja ponteiro aloca memória automaticamente. No entanto, ao criar um ponteiro, a memória não é alocada automaticamente. Ele pode apontar para uma variável única, para múltiplas variáveis e etc. Nesse sentido, ao usarmos ponteiros, é interessante que sejam usados calloc ou malloc junto com eles.

→O **scanf** pode ser usado com strings (scanf("%s", *nome_da_string*). Porém, é menos seguro do que o **fgets**, uma vez que o scanf não tem limitação de tamanho e o usuário pode invadir a memória de outras variáveis por meio dele.

Aula 09.02
[Revisão (ponteiros) + TAD - Tipos Abstratos de Dados]

→O professor iniciou a aula dando ênfase na importância de saber trabalhar com **typedef**, comando que nos auxiliará na organização de tabelas de dados.

~Relembrando~

Ao usarmos **ponteiros para structs**, acessamos algum parâmetro da struct que o ponteiro aponta por meio da "->". Exemplo:

- typedef struct {
  int nro;
  char letra;
  } tipoDado;
  tipoDado Tabela[100];
  Tabela[*índice*]->nro = 789;

→Os Tipos Abstratos de Dados (TADs) são a união do uso de estruturas para organizar dados que se relacionam com rotinas, que são funções que manipulam esses dados como desejado. Exemplos de TADs: structs, unions etc.

→**Rotinas:** são funções que manipulam os tipos abstratos de dados. Exemplo: uma função que inicia uma tabela, declarando-a, alocando-a (com malloc, calloc etc.), funções que fazem atualizações, inserem e removem dados e assim por diante.

Aula 09.08
[Lista Linear Simples (Lista encadeada)]

→**Linked Lists (listas encadeadas):** são alocadas dinamicamente na memória. Cada item não necessariamente é colocado de maneira justaposta na memória, o que traz a necessidade de cada item apontar para outros itens. As listas podem apontar para o item posterior ou anterior e posterior, e isso é o que define o tipo da lista.

→**Listas simples:** cada item da lista (situado em um endereço específico) contém o endereço do próximo item. Existem diversos tipos de listas simples. Dentre eles:

- Lista encadeada simples;
- Lista encadeada simples com nó descritor;
- Lista encadeada simples circular;
- Lista encadeada simples ordenada.

→**Listas simples (LS ou LDES):** possui um ponteiro que aponta apenas para o início da lista.

→**Listas simples com nó descritor (LSND):** possui um ponteiros que apontam para o início e para o final da lista, além de um nó descritor, que contém a quantidade de nodos da lista.

→**Listas simples circular (LSC):** é assim chamada porque o último elemento da lista aponta para o primeiro.

→**Listas duplas (LD ou LDED):** cada item - ou nodo - da lista (situado em um endereço específico) contém o endereço do próximo item **e do anterior.** Exige mais memória do que a lista simples, porém é possível andar “para frente e para trás” dentro dela. Exemplos de listas duplas:

- Listas encadeadas duplas;
- Listas duplas circulares.

→**É interessante usar typedef para definir algum tipo de dado**, para facilitar o processo no caso de ter que mudar o tipo de variável que o armazena. Por exemplo: ao trocar o tipo de uma variável que armazena alguma informação na lista, os parâmetros de todas as funções teriam de ser modificados, o que não é necessário ao se usar o typedef.

***li (lista início):** nome geralmente dado ao ponteiro que guarda o endereço do primeiro item da lista, ou seja, do começo da lista.

TAD listas encadeadas:

- **Fila:** First in - Last out (FILO / Queue);
- **Pilha:** Last in - First out (LIFO / Stack);
- **DEQUE:**
    - **Double Ended Queue**;
    - **Double Sided Queue:** fila de dois lados;
- **Priority Queue (PQ).**

→**Um nodo de uma lista é composto pelo dado e pelo endereço para o próximo item** (ou anterior e próximo, no caso das listas duplas).

→Os últimos nós de listas não circulares sempre apontam para nulo (o próximo endereço é NULL). Ao inserir um novo nó no início da lista, ele apontará para **li.** Ao inserir no final, o nó que antes apontava para NULL (último nó) apontará para o nó inserido.

Aula 09.10
[Lista Linear Simples (Lista encadeada)]

→**Memory leak:** vazamento de memória. Ocorre quando são alocados espaços de memória e eles são perdidos de alguma forma (por exemplo: aloca-se um bloco de memória e seu endereço é perdido. Então, ocorreu um vazamento de memória).

Aula 09.15
[LDES - Fila e pilha]

→**Fila:** funciona como uma fila de banco. Ou seja, é do tipo FIFO (First in, first out). O primeiro a chegar na fila é o primeiro a sair. Para tanto, ela possui um sentido para determinar onde os itens entrarão e onde sairão.

→**Pilha (stack):** funciona como qualquer pilha feita. Isto é, é do tipo LIFO (Last in, first out). O último a entrar é o primeiro a sair. **Pilhas geralmente não precisam de nó descritor**, porque não é necessário se preocupar com os elementos que estão no final da pilha (na base), uma vez que **todos os elementos são inseridos e também removidos pelo topo!**

→**Deque (Double Ended Queue):** opera tanto de maneira FIFO como de maneira LIFO. Os itens podem entrar/sair tanto pelo início, quanto pelo final.
inserir item na pilha/fila.

→Operações básicas de uma pilha:

- **Push:** inserir item na pilha;
- **Pop:** remover item da pilha;
- **Peek:** obter informação sobre o item que está no topo da pilha (“observá-lo”).

→É interessante que as listas tenham um nó descritor, para que tenham não somente um ponteiro que aponta para o início da lista, mas também para o final. Isso poupa trabalho da máquina quando ela precisa manejar itens que encontram-se nos extremos das listas.

→Head (também chamada de Front): “cabeça” ou “dianteira” de uma lista.

→Tail (também chamada de Rear): “cauda” ou “traseira” de uma lista.

Aula 09.17
[Lista Linear Dinâmica com Encadeamento Duplo ou Duplamente Encadeada (LD, LDED ou LDDE)]

→Nesse tipo de lista, os nodos não possuem ponteiros somente para o próximo nodo, mas também para o anterior.

Aula 09.22
[Séries de dados]

→Formato de arquivo **.csv (Comma-separated Values ou então Valores Separados por Vírgulas):** como o nome bem diz, nesse tipo de arquivo, os campos de dados são separados por vírgulas.

- **Separadores .csv (USA):** ‘,’ e ‘.’;
- **Separadores .csv (BR):** ‘;’ e ‘,’.

→Os dados geralmente são organizados em structs. Cada campo se refere a um dado da struct, e eles podem ser diferentes entre si (um campo de número inteiro, outro de float, outro de string etc.). Esses campos, no arquivo .csv, são separados por vírgula. Cada struct, por sua vez (dados agrupados em uma estrutura comum), é separado pela quebra de linhas.

→Ponteiros para void (void *ponteiro;): podem ser utilizados para apontar para qualquer tipo de variável (int, char etc.). Com isso, os tipos de dados usados nas structs se tornam ainda mais abstratos, podendo ser modificados sem grandes alterações na estrutura do programa.

→**LDED void:** é uma lista duplamente encadeada que no lugar de seu dado, possui um ponteiro para void que aponta para o dado. Com isso, os dados empregados na lista se tornam ainda mais abstratos e podem ser modificados com grande facilidade.

Aula 09.24
[Merge Sort]

→A abstração é importante para que as rotinas de um TAD possam ser reaproveitadas para diferentes tipos de dados.

→Merge Sort: é a ordenação de um conjunto de dados conforme algum critério específico. Sua ordenação consiste em montar uma pilha ordenada comparando os elementos do topo de diferentes "subpilhas" (sempre mais de uma e, para tanto, os elementos devem estar ordenados dentro sua respectiva "subpilha"), usando o **peek**, repetidas vezes até que todos os elementos sejam ordenados. Com isso, é descoberto qual é o elemento certo a ser pego para manter a ordem desejada. É muito mais ágil para o computador do que o Bubble Sorte, por não ter que fazer uma comparação de todos contra todos.

→Bubble Sort: é um tipo de ordenação no qual um elemento é sempre comparado com **todos** os outros. Pode ser chamado de "todos contra todos" ou "n x n". Por conta das inúmeras comparações feitas, é muito mais custoso ao computador do que um Merge Sort.

→É possível usar o fscanf para ler dados da maneira como foram escritos no arquivo texto. Exemplo:

- Arquivo .txt:
    - 11 45.1353
      12 31.3423
      13 19.3919
- fscanf:
    - fscaf(*nome_arquivo*, "%d %lf", *nome_int*, *nome_float*);
    - O escrito acima pode ser adaptado para ler no modo .csv, com a separação contendo vírgulas além do espaço e outras maneiras.

Aula 09.29
[Ordenação]

→No momento da comparação de strings, usando o strcmp(), deve-se tomar cuidado com letras maiúsculas e minúsculas, uma vez que as letras maiúsculas vêm antes das minúsculas na tabela ASCII e isso pode gerar erros.

→strcmp(): além dessa função dizer se as strings são iguais, caso não sejam, ela indica qual string vem antes usando a ordem da tabela ASCII (por isso o cuidado necessário ao comparar letras maiúsculas e minúsculas, visando uma ordenação alfabética).

- Se strcmp(str1, str2) == 0 → as strings são iguais.
- Se strcmp(str1, str2) < 0 → str1 vem "antes" da str2 (**caso não haja comparações de letras maiúsculas com letras minúsculas, str1 vem antes da str2 na ordem alfabética**);
- Se strcmp(str1, str2) > 0 → str1 vem "depois" da str2 (novamente, **caso não haja comparações de letras maiúsculas com letras minúsculas, str1 vem depois da str2 na ordem alfabética**).

→O campo usado como referência para fazer a ordenação é chamado de **chave.**

→Ao escolher qual campo será a chave da ordenação, devemos tomar cuidado em escolher uma chave que não possa ser repetida em dois itens diferentes.

→Para realizar uma ordenação por datas, é interessante realizar uma concatenação da forma "AnoMêsDia". Dessa forma, são gerados números ordenáveis e por meio deles pode-se fazer a ordenação das datas, inclusive por comparação de strings.

Aula 10.01
[Provas anteriores]

Nesta aula, o professor falou sobre como seria a prova e mostrou algumas provas de anos anteriores para a turma. Essas provas e suas respectivas resoluções podem ser encontradas na Wiki, que redireciona para um Drive que as contém.

Aula 10.08
[Matrizes Esparsas usando LDED]

→Em listas esparsas, além dos nós carregarem dados, carregam também um índice, normalmente inteiro.

→Uma matriz esparsa é uma lista esparsa na qual existe mais de um índice. Exemplos:

- Uma matriz de horários, que contém hora, minuto, segundo etc.;
- Um programa como o Excel, que trabalha com linhas e colunas.

Aula 10.13
[Análise de complexidade]

→Existem problemas solucionáveis, e não-solucionáveis. Da mesma forma, existem problemas computáveis e não computáveis. O que decorre disso, é que existem problemas não-solucionáveis que podem ser parcialmente computáveis e, com isso, parcialmente solucionável.

→Problema intratável é aquele para o qual não existe um algoritmo que solucione suas demandas em tempo hábil (relativo de acordo com o problema em questão).

→O **tempo** e o **espaço** devem ser dois fatores muito relevantes ao se pensar em um algoritmo. O tempo diz respeito ao próprio tempo de processamento, enquanto o espaço se refere à quantidade de memória necessária para o funcionamento desse algoritmo.

→Níveis de complexidade dos problemas:
→Incomputáveis (uncomputable, são os **mais complexos** de todos);
→Computáveis (computable);
→ EXP (exponential, ou exponenciais);
→P (polynomial ou polinomiais, são os **mais simples**).

→A eficiência de um algoritmo está relacionada a algumas características, sendo elas:

- a quantidade de espaço de armazenamento que ele necessita;
- a quantidade de tráfego que gera em uma rede de computadores;
- a quantidade de dados que precisa ser movida **do disco** ou **para o disco**;

→Existem dois tipos de análise da complexidade entre dois programas:

- Empírica: comparação entre programas;
- Matemática: estudo das propriedades do algoritmo.

Como calcular o tempo decorrido na execução de um programa:

```
#include <stdio.h>
#include <time.h>

int main (void) {
  time_t t1, t2, total;

  t1 = time(NULL); // retorna hora atual do sistema

  /* algoritmo */

  t2 = time(NULL);
  total = difftime(t2,t1); // retorna a diferenca t2-t1
  printf("\n\nTotal: %ld seg.\n", total);
  return 0;
 }
```

→Na análise de complexidade, as constantes se tornam irrelevantes quando pensamos em números grotescos. Portanto, elas são descartadas, importando para nós somente a **ordem** de complexidade dos algoritmos.

→Quanto às ordens de complexidade, de antemão, é interessante notar que:

$$n! > a^n > n^3 > n^2 > n \log_b n > n > \log_b n > 1$$

Em nomes:
fatorial > *exponencial* > *polinomial* > *cúbica* > *quadrática* > *linear-log* > *linear* > *logarítmica* > *constante*

→Nas análises de complexidade, consideram:

- Melhor caso (Big Omega, Ω): caso em que todos os dados já estão ordenados, não é necessária nenhuma manipulação. Esse caso é muito pouco usado para fins de comparação da efetividade de dois algoritmos, visto que é praticamente impossível encontrar um conjunto de dados que já está perfeitamente organizado. Portanto, não é um bom caso para análise.
- Caso médio (Big Theta, θ): seria o melhor caso para análise, porém é muito difícil determinar qual seria o caso médio, o que inviabiliza essa análise.
- Pior caso (Big O, O): considera a ordem de complexidade mais alta possível para o algoritmo em questão, o que o torna uma melhor análise, visto que não há surpresas, já se conhece o maior tempo possível que o algoritmo pode levar. É bem mais simples de ser encontrado do que o Big Theta, e por isso é a análise recomendada pela maioria dos livros didáticos.

Aula 10.15
[Estrutura de dados não linear: Árvores]

→Uma árvore é um tipo especial de grafo.

→Um grafo é um conjunto de vértices e arestas que conectam elementos de maneiras variadas.

→A **raiz** (ou **root**) da árvore é o elemento que está representado mais acima. Em uma árvore binária, a raiz dá origem a dois ramos (esquerdo ou direito), e cada ramo, por sua vez, também só se subdivide em no máximo dois ramos. Em uma árvore ternária, a raiz se subdivide em três ramos, e assim por diante.

→Nas árvores hierárquicas existem elementos **pai** e elementos **filhos**, e os pais estão hierarquicamente acima dos filhos. Em uma **árvore binária**, cada **elemento pai tem no máximo dois filhos,** o **filho da esquerda e filho da direita**.

→As árvores simplificam de certa forma a manipulação dos dados, visto que você escolhe diferentes caminhos a percorrer. Ao escolher um caminho, os dados de outros caminhos não são acessados ou manipulados, o que poupa tempo e fornece uma maior eficiência para certas estruturas de dados.

→As árvores binárias possuem nós, assim como as listas dinâmicas. Porém, ao invés de os nós conterem ponteiro para o próximo elemento, ou para o próximo e o anterior, possui ponteiros que apontam para o filho da esquerda e para o filho da direita. Também contém o dado do nó, como nas listas dinâmicas.

→**Nó folha** ou **nó terminal:** um nó que não possui filhos.

→**Nó interno** ou **nó não terminal:** um nó que possui filhos.

→**Caminho:** liga um nó ao outro.

→**Altura:** quantidade de **níveis** de uma árvore. A cada passagem de nó pai para filho, é aumentado um nível. A raiz, por convenção, está no nível 0, seus filhos estão no nível 1, e assim por diante.

- Uma árvore de altura h possui $2^h - 1$ nós;
- Um nível n possui exatamente no máximo 2^n nós.

→Uma árvore cheia é aquela que possui o máximo de elementos que seus níveis comportam. Por exemplo: uma árvore de 3 níveis, que possui 2^n nós em cada nível, sendo *n* o índice do nível.

→O resultado de se usar uma árvore binária, é que um dado pode ser achado em um conjunto de bilhões, trilhões de dados em poucos passos.

→Um modo de se fazer uma árvore ordenada, por exemplo, é colocando em um filho da esquerda sempre um valor menor, e no filho da direita sempre um valor maior. Árvores ordenadas facilitam ainda mais as buscas de dados.

→Uma árvore binária balanceada é uma árvore ordenada, na qual a raiz é colocada de maneira que os dois lados da árvore fiquem equilibrados, isto é, um lado não fica com uma quantidade de itens muito maior do que o outro.

Aula 10.20
[Comparando TADs: Lista Ordenada x Árvore Ordenada]

→Uma busca binária - busca por um dado dentro de uma árvore binária - possui complexidade *log n*, ou seja, menor que a linear, o que torna essa estrutura de dados extremamente eficiente para se organizar dados e para buscá-los posteriormente.

→Árvore binária:
- **Search:** Big Ω → O(*log n*); Big O → O(*n*);
- **Insert:** Big Ω → O(*log n*); Big O → O(*n*);
- **Delete:** Big Ω → O(*log n*); Big O → O(*n*).

→Árvore binária ordenada e balanceada (AVL tree):
- **Search:** Big Ω → O(*log n*); Big O → O(*log n*);
- **Insert:** Big Ω → O(*log n*); Big O → O(*log n*);
- **Delete:** Big Ω → O(*log n*); Big O → O(*log n*).

→**A altura** de uma árvore nos dá o **número máximo de passos** necessário para atingir **qualquer** dado contido na árvore.

Aula 10.22
[Árvores binárias ordenadas (ABO/ABB ou BST - Binary Search Tree)]

→Como dito anteriormente, uma maneira de se confeccionar uma árvore binária ordenada de **inteiros**, é colocar todos os valores MAIORES do que o valor pai a direita, e todos MENORES a esquerda, ou o contrário.

Aula 11.10
[Recursividade - Funções recursivas]

→A recursividade, em ciência da computação, é a definição de uma sub-rotina (função ou método) que pode invocar a si mesma.

→Como resolver a função matemática fatorial por recursividade:

- ```
  int fat (n) {
        if ((n==1) || (n==0)) return 1;
        else return fat(n-1)*n;
  }
  ```

→A recursividade pode ajudar a resolver problemas de uma maneira bastante simplificada, porém o seu problema é controlar a quantidade de chamadas da função. Isso porque as variáveis que **todas** as chamadas da função são depositadas na pilha, até que sejam dados os **return** da função. Portanto, é importantíssimo se preocupar em usar a recursividade de maneira adequada para não estourar a pilha, isto é, se houverem apenas chamadas, sem retornos da função, a pilha estourará.

→Algoritmo de Flood-fill: preenchimento por inundação. Por exemplo, quando é usada a ferramenta “balde de tinta”, do Paint: todo pixel da mesma cor do pixel clicado é pintado com a cor escolhida, até que o algoritmo encontre um pixel que tenha cor diferente, aonde ele para.

→Para montarmos uma boa recursão, é preciso ter uma boa função de corte. Uma boa função de corte, por exemplo, não permite ao programa passar em partes do problema que a função já passou.
  ```

Aula 11.12
[Recursividade em funções como Exibe ArvBin, Procura ArvBin]

→Para fazer uma impressão da árvore, ou busca exaustiva (por todos os itens da árvore), em ordem (**Inorder**), é usada a recursividade para ir todos os nodos para a esquerda e imprimir repetidamente os nodos da esquerda, o nodo pai, e os nodos da direita. Assim, caso seja uma árvore ordenada, de números, por exemplo, os números seriam impressos em ordem crescente.

→Para liberar os nodos da árvore também é utilizada a recursividade, porém aqui é empregada a pós ordem (**Postorder**).

→Em uma consulta, em uma árvore binária, não é usada a recursividade. É feito um laço que faz com que, a cada passo dado na árvore, uma enorme quantidade de nodos seja desprezada, o que otimiza o algoritmo em termos de velocidade, mas não varre a árvore toda.

→Os comandos antecedidos por "system", são provenientes da biblioteca stdlib e elas permitem inserir comandos no prompt de quem está executando o programa. Por exemplo, o comando system("Pause"); faz com que o programa pause, no Windows.

→Quando é necessária fazer uma remoção do nodo que está na raiz, em uma árvore ordenada, nos deparamos com certa complicação. Porém, se escolhermos o nó adequado, isto é, o **predecessor** ou o **sucessor** da raiz (nó imediatamente atrás ou à frente na ordem), a raiz pode ser substituída por ele sem prejuízo nenhum no código.

→O **predecessor** é o maior valor da árvore da esquerda.

- "Esquerda e tudo à direita!"

→O **sucessor** é o menor valor da árvore da direita.

- "Direita e tudo à esquerda!"

Aula 11.17
[Árvores binárias ordenadas e balanceadas (AVL)]

→O primeiro passo do algoritmo é buscar saber se a árvore está balanceada ou não, e se está desequilibrada, saber para qual lado. Quando a árvore não está cheia, ela pode ter um desbalanceamento de uma folha, e esse é tolerado. Passando desse desbalanço, o algoritmo busca acertá-la. A diferença entre as alturas das subárvores da esquerda e da direita é chamada fator de balanceamento ou **fb**.

→Se uma árvore não está balanceada, o Big O da busca, inserção e remoção é O(*N*). Em uma árvore balanceada, por sua vez, o Big O dessas rotinas é O(*log N*).

→Existem mais tipos de árvores binárias ordenadas e balanceadas além do AVL, como o Red-Black, porém eles não serão abordados nessa disciplina.

→ A AVL Tree foi criada por Adelson-Velskii e Landis, que dão origem ao seu nome. Para balancear a árvore, ela faz **rotações** simples ou duplas, que ocorrem a cada inserção ou remoção. Portanto, uma árvore binária balanceada é balanceada a **CADA** inserção ou remoção, o que indica que essa manutenção precisa ocorrer **necessariamente** a cada mudança na árvore. Não adianta querer balancear uma árvore depois de muitas inserções, porque o algoritmo não foi feito para isso.

→Como dito anteriormente:

fb = AE - AD

→**Fator de balanceamento = Altura da esquerda - Altura da direita**.

→Caso uma das subárvores de um nó não exista, então a altura dessa subárvore será igual a -1.

→Se o fator de balanceamento for maior do que 1 ou menor do que -1 em algum nó, significa que a árvore está desbalanceada **naquele nó em específico**.

→**Rotações:** são as operações básicas usadas para balancear uma árvore AVL. Existem rotações simples e rotações duplas. As duplas, são basicamente duas rotações simples feitas sucessivamente.

- **LR (ou RR, conforme o Backes, left rotation):** rotação à esquerda;
- **RR (ou LL, conforme o Backes, right rotation):** rotação à direita;
- **LR (rotação dupla left-right):** primeiro faz uma rotação à esquerda, e depois outra a direita;

- **RL (rotação dupla right-left):** primeiro faz uma rotação à direita, e depois outra à esquerda.

Aula 11.19
[Árvores binárias ordenadas e balanceadas (AVL)]

→Remover um nó de uma AVL também não é uma tarefa fácil. Ele é removido da mesma maneira que na árvore binária ordenada (no sentido de pegar o antecessor ou o predecessor para substituí-lo), porém a árvore deve ser balanceada após a remoção. Remover um nó da árvore da esquerda é a mesma coisa que inserir um nó na árvore da direita.

Aula 11.19
[Árvores binárias ordenadas e balanceadas (Red-Black)]

→Os nodos de uma árvore Red/Black se dividem entre vermelhos e pretos. A raiz geralmente é **PRETA.** Novos nodos inseridos são **vermelhos**.

→Assim como na árvore AVL, para corrigir os desbalanços, a Red/Black utiliza rotações. Ademais, quando há rotação na Red/Black, há também uma **mudança de cores**.

→A ideia da Red/Black é que cada nível da árvore tenha uma cor, e com isso que ela consegue medir o desbalanço. Se um nó é vermelho, os filhos dele são pretos. Porém, todos os nós entrantes são vermelhos.

→A árvore Red/Black é também chamada árvore RB e seu algoritmo foi criado por Rudolf Bayer e aperfeiçoado por Robert Sedgewick.

→A árvore AVL é mais eficiente nas operações de busca, enquanto a Red/Black é mais eficiente em operações de inserção/remoção. No entanto, ordem de complexidade dos dois algoritmos é *log(N)*.

Aula 12.01
[Árvores genéricas ou generalizadas]

→Um exemplo de árvore genérica é uma árvore genealógica. Nelas não existem limitação de nodos filhos e, portanto **não são binárias**.

→Elas podem funcionar como grafos. Nesse caso, nodos “irmãos” podem estar ligados entre si, podendo formar até uma lista de “irmãos”. Portanto, obtêm-se algo parecido com uma “lista de listas”, visto que cada nível da árvore começa a ser uma lista.