

UNIVERSIDADE ABERTA DO BRASIL
UNIVERSIDADE FEDERAL DE ALAGOAS
Instituto de Computação
Curso de Bacharelado em Sistemas de
Informação

ALGORITMO E ESTRUTURA DE DADOS II

Prof. Ailton Cruz dos Santos

**UNIVERSIDADE ABERTA DO BRASIL
UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO**

Curso de Bacharelado em Sistemas de Informação
Disciplina:
ALGORITMO E ESTRUTURA DE DADOS II
Prof. Ailton Cruz dos Santos



Este trabalho está licenciado sob uma Licença Creative Commons Atribuição-NãoComercial-Compartilhual 4.0 Internacional. Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc-sa/4.0/>.

Texto da disciplina produzido em 2008 e revisado em 2010, 2012 e 2014

Apresentação

Prezado estudante,

Esta disciplina faz referência direta à disciplina Algoritmo e Estrutura de Dados I quanto a metodologia empregada, texto, implementações em Python etc., tomando os conhecimentos construídos naquela disciplina como indispensáveis. Num primeiro momento, estudamos os fundamentos, desde a iniciação à lógica de programação até as estruturas de dados homogêneas (vetores, matrizes e cadeias de caracteres). Passamos a conhecer problemas que envolviam tais estruturas para resolução. O trabalho agora é avançar um pouco mais nas especificidades das estruturas de dados, conseqüentemente, ampliando as possibilidades em termos de soluções de problemas. O procedimento requerido do estudante ao longo da disciplina permanece o mesmo: seguir cada etapa planejada sempre interagindo com o professor, tutores e colegas.

Prof. Ailton Cruz.

Plano de trabalho

TÍTULO DA DISCIPLINA: Algoritmo e Estrutura de Dados II

CARGA HORÁRIA

Presencial: 06 horas; On-line: 114 horas

EMENTA:

Estruturas de Dados: Listas. Filas. Pilhas. Árvores. Grafos. Algoritmos para manipulação das estruturas de dados estudadas.

OBJETIVOS DA DISCIPLINA

Objetivo geral:

Capacitar o estudante a resolver determinadas classes de problemas que demandam algoritmos de manipulação de dados segundo estruturas avançadas.

Objetivos específicos:

Apresentar as estruturas de dados avançadas: Listas, Filas, Pilhas, Árvores e Grafos;
Identificar e aplicar algoritmos clássicos de manipulação dessas estruturas e desenvolver novos algoritmos, implementando na linguagem Python;
Habilitar o estudante a tomar decisões quanto a escolha das estruturas de dados adequadas à solução de um determinado problema.

METODOLOGIA DE ENSINO

Os estudantes deverão assimilar o conteúdo programático a partir da leitura e análise dos textos da disciplina disponíveis no AVA Moodle organizados por semanas. O acesso à bibliografia complementar, a execução de testes dos algoritmos em laboratório de informática, bem como as interações com o professor e tutores usando ferramentas da plataforma Moodle são elementos constitutivos da metodologia desta disciplina.

METODOLOGIA DE AVALIAÇÃO:

A avaliação do estudante acontece de forma contínua mediante análise das interações no AVA Moodle e tarefas semanais. A participação on-line mínima requerida do estudante é sua atividade no fórum sobre os conteúdos da semana e registro em seu blog de suas seções de estudo, da seguinte maneira. No fórum sobre os conteúdos, o estudante deve interagir com os demais participantes a fim de tirar dúvidas, ora as suas, ora as dos colegas. No blog, o estudante deve registrar que roteiro seguiu no seu estudo, suas estratégias, o que conseguiu estudar, que aspectos foram relevantes, a que conclusões chegou, como está sendo seu contato com a tutoria inclusive se fez contato fora do AVA, etc. Juntas, as participações no fórum dos conteúdos e registro no blog correspondem a 50% da pontuação de cada semana. Esta pontuação é completada com uma tarefa sobre o tema da semana. Obs.: Espera-se que as demais participações, como no fórum de discussões gerais, por exemplo, sejam normalmente detalhes daquilo discutido no fórum dos conteúdos ou registrado no blog. Tais participações podem melhorar a pontuação de interação, mas são desconsideradas se não houver conectividade no blog ou no fórum dos conteúdos. A nota do estudante é composta da maneira seguinte:

-Composição da primeira nota: 60% desta é obtida por pontuação das atividades on-line das três primeiras semanas (2,0 pontos por semana) e 40% por prova online individual ao final da terceira semana.

-Composição da segunda nota: 60% desta é obtida por pontuação das atividades on-line das quatro últimas semanas, de 1,5 ponto por semana e 40% por prova presencial, individual e com consulta, ao final da sétima semana.

CONTRIBUIÇÃO PARA A FORMAÇÃO PROFISSIONAL

Ao final da disciplina, o estudante terá aprimorado sua capacidade de resolver problemas, implementando as soluções no computador, e desenvolvido novas habilidades quanto à combinação de estruturas lógicas e de dados, sendo isto um requisito importante na produção e análise de sistemas de software.

PRÉ-REQUISITOS

Os conceitos desenvolvidos na disciplina Algoritmo e Estruturas de Dados I (comandos e estruturas lógicas básicas, subalgoritmos e estruturas de dados homogêneas).

APLICAÇÃO PRÁTICA DA DISCIPLINA

O conhecimento desenvolvido nesta disciplina complementa a disciplina Algoritmo e Estrutura de Dados I e, aliado a outros específicos ao longo curso, dará ao estudante as condições para produção nas áreas de bancos de dados, redes de computadores, Internet, etc.

CONTEÚDO PROGRAMÁTICO DA DISCIPLINA:

Módulo I - Tipos abstratos de dados (TAD)

- 1.1 - TAD e o Paradigma Imperativo
 - 1.1.1 - Atributos e interface - Registro, Vetor de registros;
 - 1.1.2 - Experimentação;
- 1.2 - TAD e o Paradigma Orientado a Objetos
 - 1.2.1 - Atributos e interface - Classes e objetos;
 - 1.2.2 - Experimentação - Classes predefinidas da linguagem Python;

Módulo II - Estruturas de dados lineares

- 2.1 - Listas
 - 2.1.1 Conceituação - Lista seq. e L. encadeada, L. estática e L. dinâmica;
 - 2.1.2 O TAD lista;
- 2.2 - Pilhas
 - 2.2.1 Conceituação;
 - 2.2.2 O TAD Pilha;
- 2.3 - Filas
 - 2.3.1 Conceituação
 - 2.3.2 O TAD Fila;

Módulo III - Estruturas de dados não-lineares

- 3.1 - Árvores
 - 3.1.1 Conceituação - Propriedades, Caminhamentos;
 - 3.1.2 O TAD Árvore;
- 3.2 - Grafos
 - 3.2.1 Conceituação - Terminologia, Representação de grafos

3.2.2 O TAD Grafo

BIBLIOGRAFIA:

FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de Programação: a Construção de Algoritmos e Estruturas de Dados. 3a ed. São Paulo: Makron Books, 2005. 232 p.

SZWARCFITER, J.L. e MARKENZON, L Estruturas de Dados e Seus Algoritmos. 2ª ed., LTC Editora, 1997.

PILGRIM, M. Mergulhando no Python. Rio de Janeiro: Alta Books, 2004.

BIBLIOGRAFIA COMPLEMENTAR: A bibliografia complementar é composta de uma coleção dinâmica de links com sugestões de leitura, disponibilizada na página da disciplina no Ambiente Virtual de Aprendizagem (AVA) Moodle.

Plano de tutoria

IDENTIFICAÇÃO DA DISCIPLINA
Disciplina: ALGORITMO E ESTRUTURA DE DADOS II Carga horária: 120 horas Professor autor: Ailton Cruz dos Santos Ementa: Estruturas de Dados: Listas. Filas. Pilhas. Árvores. Grafos. Algoritmos para manipulação das estruturas de dados estudadas.
PLANEJAMENTO
MOMENTOS PRESENCIAIS <ul style="list-style-type: none">• PRIMEIRO MOMENTO. Apresentação da disciplina, incluindo sequência do conteúdo, metodologia de ensino, recomendações ao estudante, descrição do processo de avaliação etc. Inclui breve revisão de assuntos da disciplina Algoritmo e Estrutura de Dados I, introdução aos temas da semana corrente (haverá aula em laboratório se o encontro ocorrer a partir da segunda semana).• SEGUNDO MOMENTO. Constará de aula de reforço do conteúdo trabalhado até aquele momento da disciplina, avaliação das estratégias adotadas e dos níveis de participação no curso, e introdução aos temas da semana corrente.
REUNIÕES COM A TUTORIA <p>Para avaliação do andamento da disciplina, reuniões da equipe de professores (professor ministrante e tutores) deverão ocorrer pelo menos três vezes ao longo do curso.</p> <ul style="list-style-type: none">• A primeira, na semana que antecede o início da disciplina. Pauta: Apresentação do material com o conteúdo, metodologia de ensino e avaliação;• A segunda, durante a quarta semana. Pauta: Análise do processo ensino-aprendizagem, dos resultados parciais, e discussão de casos particulares;• Ao final da sétima semana. Pauta: Análise do processo ensino-aprendizagem, da avaliação e dos resultados obtidos;
AVALIAÇÃO DOS ESTUDANTES <p>Primeira nota da disciplina:</p> <ul style="list-style-type: none">• As atividades realizadas durante as três primeiras semanas têm peso de 6,0 pontos. Em cada semana, as interações do estudante no ambiente virtual de aprendizagem (o Moodle), blog e fórum juntas, valem 1,0 ponto, e as atividades dos finais de semana, também. A nota é completada com uma prova individual, online, valendo 4,0 ao final da terceira semana em data a ser marcada. <p>Segunda nota da disciplina:</p> <ul style="list-style-type: none">• As atividades no Moodle da quarta até a sétima semana têm peso 6,0. A prova que consta no calendário como AV2 tem peso 4,0;• Critérios para a AV2 - Será uma prova presencial e individual, envolverá todo conteúdo da disciplina e será com consulta. Os únicos materiais que podem ser consultados são: o texto da disciplina e anotações pessoais que o estudante tenha feito durante seus estudos. O material consultado é pessoal. Não é permitido consulta de materiais de colegas durante a prova.
ATIVIDADES A SEREM DESENVOLVIDAS PELOS ESTUDANTES <p>No início de cada semana:</p> <ul style="list-style-type: none">• Planejar suas atividades da semana, definindo horários e guiando-se pelo tempo mínimo sugerido na página da disciplina no AVA Moodle; <p>Diariamente:</p> <ul style="list-style-type: none">• Seguir os roteiros de estudo disponíveis no texto da disciplina com teoria e prática;• Examinar e repetir os exemplos, refletir sobre a teoria, exemplos e laboratórios. Ao concluir esses passos, proceder a sua autoavaliação realizando os exercícios ao final de cada subunidade, fazendo testes dos algoritmos em computador usando a plataforma Python;• Participar, em todas essas fases acima, do fórum "Sobre os conteúdos": Interagindo

com os colegas e os professores, expondo dúvidas e questionamentos, contribuindo no sentido inclusive de auxílio aos colegas nas resoluções de exercícios, etc. Os temas que extrapolarem o escopo do curso devem ser levados para o fórum “Discussões Gerais”;

- Fazer um registro breve no seu BLOG: Anotar o que conseguiu em cada dia de estudo, que exercícios conseguiu resolver ou não, das suas dificuldades ou observações gerais de desejo fazer sobre os conhecimentos adquiridos;

Ao final de cada semana:

- Resolver um problema simples usando os recursos desenvolvidos na semana respectiva

ATRIBUIÇÕES DO TUTOR ONLINE

- Tomar ciência das atividades a serem desenvolvidas pelos estudantes com antecedência. Deve estar familiarizado com a metodologia da disciplina e a plataforma Python e apto a tirar dúvidas dos estudantes e a sugerir novos exercícios;
- Participar regularmente do “Fórum da tutoria” (não visível para os estudantes) onde são tratadas questões operacionais de acompanhamento e avaliação;
- Participar das interações nos fóruns “Discussões Gerais”, “Sobre os conteúdos” para contato com seus estudantes, como também no “Fórum de notícias”;
- Acessar o perfil de cada estudante, acompanhando os registros nos *blogs* e fóruns: Conferir a coerência de seus registros, comparando com suas participações nos fóruns; Verificar desempenho e a frequência de acesso, alertando em tempo os participantes sobre o encaminhamento de suas atividades; Detectar conceitos equivocados para a imediata correção (tanto nos fóruns quanto nos blogs);
- Providenciar recursos para auxiliar os estudantes em suas dúvidas;
- Estimular o estudante a fazer os exercícios de autoavaliação e a assumir uma atitude autônoma;
- Corrigir as tarefas ao final de cada semana e atribuir a pontuação em blog e fórum;
- Atribuir a pontuação semanal preferencialmente até a quarta-feira da semana seguinte ou, no máximo, até a quinta-feira;
- A pontuação de blog e fórum (em cada um destes itens) é atribuída na faixa de 0 a 0,5: fórum é analisado pela qualidade e abrangência por refletir interesse no aprendizado e integração com os colegas e demais participantes do curso; blog é avaliado pela coerência (não quantidade) - deve refletir que o estudante de fato está dando atenção ao curso (estudando, seguindo a metodologia adotada, interagindo,...), enfim, que tem um *plano individual de estudo* (mesmo que não escreva explicitamente);
- O fórum central é o “Sobre os conteúdos”. Os assuntos que extrapolarem o escopo do curso devem ser levados para “Discussões Gerais”. Não serão pontuadas as participações no fórum de discussões gerais ou de notícias, que não revelem consistência com o tema da semana, ou que não tenha sido feita pelo menos uma postagem pertinente no fórum dos conteúdos da semana;
- Manter vigilância principalmente sobre os estudantes que não estão sendo ágeis no registro dos seus blogs (ou estão ausentes no Moodle), enviando-lhes mensagens de motivação, entrando em contato direto se necessário;
- Coordenar a recepção das atividades da semana notificando os estudantes com pendências.

ATRIBUIÇÕES DO TUTOR PRESENCIAL

- Atender os requisitos e a atribuição clássica de suporte às atividades presenciais;
- Participar regularmente do “Fórum da tutoria” (não visível para os estudantes) onde são tratadas questões operacionais de acompanhamento e avaliação;
- Tomar ciência das atividades a serem desenvolvidas pelos estudantes com antecedência. Tirar pessoalmente dúvidas dos estudantes e/ou coordenar grupos de estudo no polo contando com a participação dos monitores, diagnosticando os casos em parceria com a tutoria online;
- Reservar parte de sua carga horária para um “plantão tira-dúvidas” no polo, em horário a combinar com os estudantes e os monitores (principalmente com respeito a execução de programas);
- Providenciar recursos para auxiliar os estudantes em suas dúvidas;

- Em seu contato com os estudantes, estimulá-los a fazer os exercícios de autoavaliação e a assumir uma atitude autônoma;
- Ajudar a tutoria online na investigação das ausências de estudantes na plataforma combatendo a evasão.

Cronograma / Índice

	<i>Introdução</i>	10
Primeira semana	<i>Módulo I - Tipos abstratos de dados (TAD)</i>	13
	<i>Unidade I.1 - TAD e o Paradigma Imperativo</i>	16
	<i>Laboratório - TAD</i>	24
Segunda semana	<i>Módulo I - Tipos abstratos de dados (TAD)</i>	13
	<i>Unidade I.2 - TAD e o Paradigma Orientado a Objetos</i>	32
	<i>Laboratório - Uso de classes predefinidas de Python</i>	36
Terceira semana	<i>Módulo II – Estruturas de dados lineares</i>	47
	<i>Unidade II.1 - Listas</i>	48
	<i>Laboratório - Listas</i>	57
Quarta semana	<i>Módulo II - Estruturas de dados lineares</i>	47
	<i>Unidade II.2 - Pilhas</i>	67
	<i>Laboratório - Pilhas</i>	73
Quinta semana	<i>Módulo II - Estruturas de dados lineares</i>	47
	<i>Unidade II.3 - Filas</i>	81
	<i>Laboratório - Filas</i>	85
Sexta semana	<i>Módulo III - Estruturas de dados não-lineares</i>	91
	<i>Unidade III.1 - Árvores</i>	92
	<i>Laboratório - Árvores</i>	101
Sétima semana	<i>Módulo III - Estruturas de dados não-lineares</i>	91
	<i>Unidade III.2 – Grafos</i>	106
	<i>Laboratório - Grafos</i>	116

Introdução

Este texto aborda elementos sobre as seguintes estruturas de dados avançadas e seus principais algoritmos: Listas, Filas, Pilhas, Árvores e Grafos, conteúdos da disciplina Algoritmo e Estrutura de Dados II. Toma os conteúdos da disciplina Algoritmo e Estrutura de Dados I como pré-requisitos. Será apresentado e utilizado o conceito de *tipo abstrato de dados* na programação estruturada e na orientada a objetos. Particularmente, o estudo da orientação a objetos servirá apenas para criar condições de uso de classes predefinidas da linguagem Python. Estão previstos testes dos algoritmos usando esta linguagem a fim de promover uma sedimentação dos conceitos.

O material está distribuído em três módulos, cada um com parte teórica seguida de *laboratórios* e exercícios de aprendizagem. Nos laboratórios, os programas seguem o mesmo tipo de identificação da disciplina anterior (Isto é, identificadores iniciados com a letra L seguida de dígitos identificando a subunidade e mais dois dígitos identificando o experimento realizado).

O primeiro módulo do curso apresenta o conceito de TAD (Tipos Abstratos de Dados). Esse conceito é então aplicado no desenvolvimento dos demais módulos, inclusive quanto aos aspectos de implementação na linguagem Python. O segundo módulo trata das estruturas de dados lineares (listas, pilhas e filas) e o terceiro, as estruturas não lineares (árvores e grafos).

A plataforma Python e os laboratórios

Os laboratórios planejados na disciplina visam conferir os elementos sobre algoritmos e estruturas dados exibidos na parte teórica. Cada laboratório deve ser acompanhado observando-se seus objetivos, recursos e experimentação. Os objetivos descrevem o que estudante deve atingir ao concluir os experimentos. Os recursos e experimentação fazem um paralelo entre a parte teórica e recursos disponíveis na linguagem utilizada, no caso, a linguagem Python.

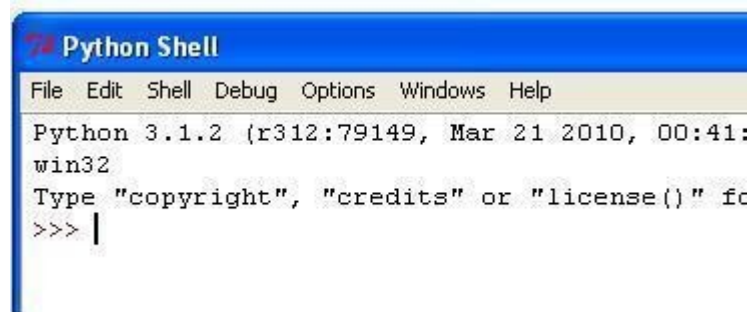
Identificação dos programas. A fim de facilitar a localização no disco, os programas são identificados com nomes iniciados com a letra L seguida de dígitos identificando a subunidade à qual estão associados e mais dois dígitos identificando o experimento realizado. Por exemplo, L213_07.py é o nome do programa correspondente ao Experimento 07 do laboratório da Subunidade 2.1.3.

Instalação do ambiente de programação Python. No Windows, é bastante acessar a página <http://www.python.org/download/>, baixar e instalar o arquivo com extensão "*.msi", escolhendo uma versão 3.x (tudo que é necessário é baixado automaticamente). Essa instalação vem acompanhada do aplicativo IDLE (Python GUI) de edição e execução dos programas. O sistema operacional Linux já vem normalmente com o Python instalado, mas não inclui o programa IDLE, que deve ser instalado conforme instruções específicas daquele sistema.

Após a instalação, devemos fazer um pequeno teste. Consideremos a instalação Windows. Executar o IDLE clicando em:

Todos os programas -> Python 3.1 -> IDLE (Python GUI),

A execução faz surgir uma tela como a seguinte:



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.2 (r312:79149, Mar 21 2010, 00:41:
win32
Type "copyright", "credits" or "license()" fo
>>> |
```

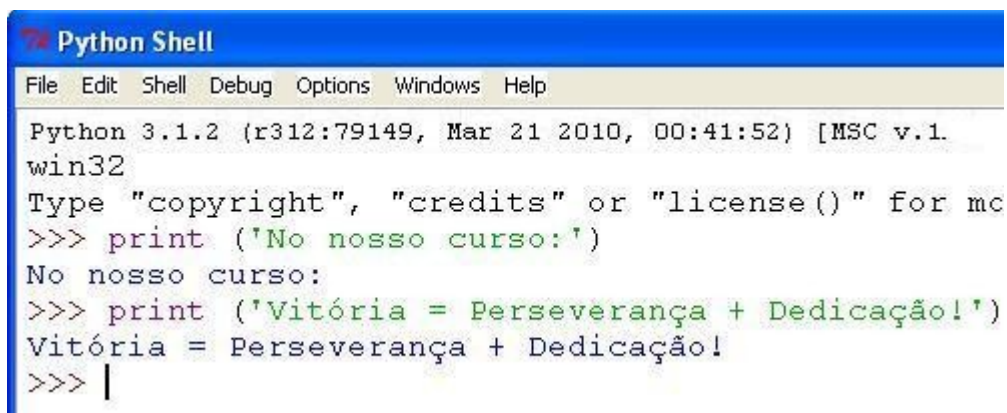
Python Shell

Essa tela, chamada de *Python Shell*, fica disponível para execução de comandos e exibição de resultados. A linguagem de programação Python é *interpretada* (Ver **Unidade 1.2**). Os comandos podem ser interpretados um a um interativamente ou reunidos em arquivos de programas chamados *módulos*.

Execução interativa de comandos. Inserimos o comando a ser executado diante do *prompt* ">>>". Como exemplo, vamos executar os seguintes (um de cada vez, apertando a tecla <Enter>):

```
print ('No nosso curso:')
print ('Vitória = Perseverança + Dedicação!')
```

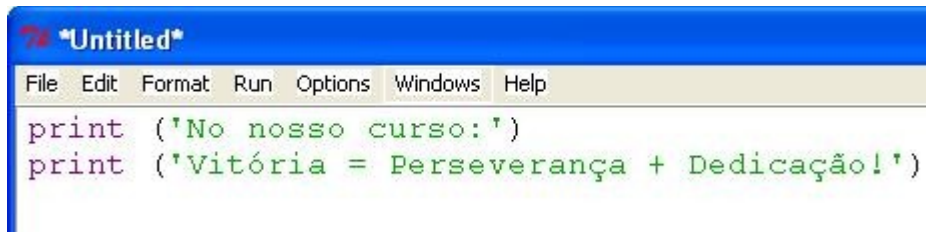
Cada comando executado produz seu resultado logo em seguida:



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.1.2 (r312:79149, Mar 21 2010, 00:41:52) [MSC v.1
win32
Type "copyright", "credits" or "license()" for mc
>>> print ('No nosso curso:')
No nosso curso:
>>> print ('Vitória = Perseverança + Dedicação!')
Vitória = Perseverança + Dedicação!
>>> |
```

prompt de comandos

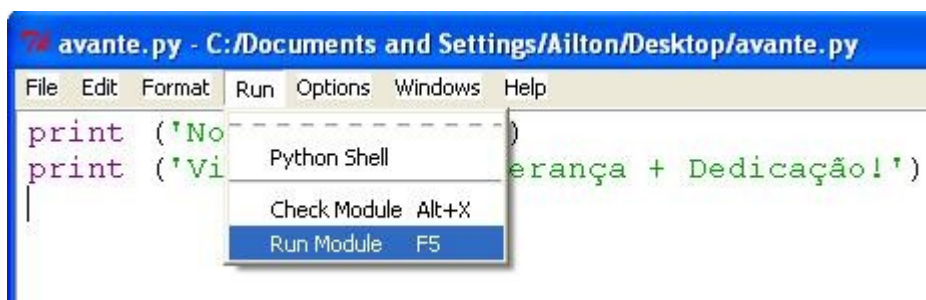
Edição e execução de programas. No menu principal, clicamos em *File* (arquivo) -> *New Window* (nova Janela) e imediatamente uma tela de edição é aberta (ver figura abaixo). Nesse espaço, experimentemos com os comandos acima (já executados interativamente):



```
*Untitled*
File Edit Format Run Options Windows Help
print ('No nosso curso:')
print ('Vitória = Perseverança + Dedicação!')
```

Janela de edição do IDLE

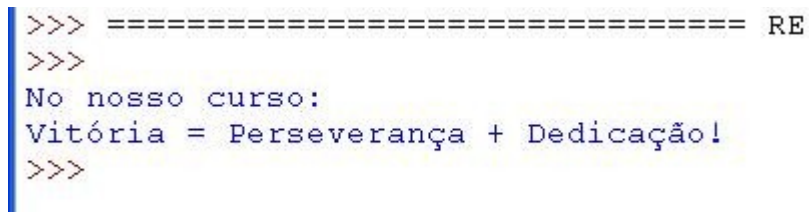
Após a digitação, *salvamos* o programa acessando o menu principal em *File* -> *Save* e escolhendo um nome, digamos, "avante.py" (escolhemos também o diretório de nossa preferência). Para executar, clicamos em *Run* -> *Run Module* (ou simplesmente apertamos a tecla F5).



```
avante.py - C:/Documents and Settings/Ailton/Desktop/avante.py
File Edit Format Run Options Windows Help
print ('No
print ('Vi
Python Shell
Check Module Alt+X
Run Module F5
erança + Dedicação!')
```

Sub-Menu de execução de programas

O resultado é o seguinte:



```
>>> ===== RE
>>>
No nosso curso:
Vitória = Perseverança + Dedicação!
>>>
```

Execução de `avante.py`

Módulo I

Tipos Abstratos de Dados (TAD)

Abordaremos neste primeiro módulo os *Tipos Abstratos de Dados*, que são, basicamente, construções lógicas dos dados que permitem uma estruturação mais intimamente ligada ao problema a ser resolvido. Consideremos o conhecimento prévio sobre o que são os tipos de dados e sua estruturação. Por exemplo, podemos ter um dado que é apenas uma letra (um caractere) e outro que pode ser o nome de uma pessoa (uma sequência de caracteres – uma *string*). Uma cadeia de caracteres é, portanto, um exemplo típico de estruturação de dados. A sequência de caracteres deixa de ser apenas uma coleção de caracteres, adquirindo um significado contextual, passando a ser fundamental na compreensão do problema.

Existe uma relação íntima entre o processo de estruturação dos dados e a elaboração de algoritmos mais eficientes. Vejamos o exemplo seguinte (**Exemplo 1.1**) que trata de um problema da vida diária e demonstra bem essa relação.

Exemplo 1.1

Inevitavelmente, a atividade de organizar coisas faz parte de nossas vidas, mesmo que, às vezes, não seja uma tarefa prazerosa¹.



“Gavetas”

(Fonte: <http://atchim-blog.blogspot.com/2007/06/gavetas.html>, acessado em 05/01/2012)

Suponhamos que uma pessoa tenha sido cobrada indevidamente de uma conta de energia elétrica. Então, para defender-se, ela precisa encontrar o respectivo documento da distribuidora de energia com a autenticação bancária de pagamento e, ainda, é sabido que os documentos estão em uma “daquelas” gavetas!

A preocupação inicial será: Por onde "atacar" o problema? Em palavras técnicas, qual “algoritmo” de busca essa pessoa deveria executar? Dentre as providências possíveis a serem

¹No texto “Gavetas”, disp. em <http://atchim-blog.blogspot.com.br/2007/06/gavetas.htm>, acessado em 2008, de forma bem humorada, o autor escreve: “Sempre chega o dia em que a gente paga o preço de uma gaveta desarrumada”.

tomadas, uma delas seria tentar encontrar exaustivamente a tal conta entre centenas de documentos de diversos tipos e datas lá existentes. Parecendo essa ideia ser inviável, logo é descartada. O que surge como mais sensato é, primeiramente, classificar qualquer documento encontrado em "conta de energia" e "não-conta de energia" e, em seguida, devolver para aquele amontoado os documentos que não sejam conta de energia. Estando em mãos as contas já separadas, alinhadas segundo uma ordem qualquer (uma conta sobre a outra), a busca será facilitada em relação à situação anterior (as contas estavam espalhadas, sem ordem alguma).

Podemos perceber que o algoritmo para busca com contas já organizadas será diferente de outro para busca com as contas desorganizadas (até misturadas com outros documentos). Uma vez separadas as de interesse, o passo seguinte é encontrar a conta específica (aquela que se espera ter sido paga), e isso pode ser feito ainda de duas maneiras. Uma delas é "varrer" o maço de contas de energia a partir da primeira até a última, até encontrar a conta desejada. A outra maneira é ordenar sequencialmente por data e efetuar a busca da forma seguinte: obedecer a sequência de datas, encontrar o ano e, em seguida, o mês correspondente ao da conta desejada.

Podemos perceber como uma vantagem considerável, que as providências acima farão detectar com mais facilidade uma possível inexistência da conta buscada, em comparação com a busca exaustiva em meio às tralhas!

O exemplo acima expõe com propriedade que o modo de "preparação" dos dados é uma fase importante da solução de um problema. Consegue-se decidir sobre que algoritmo se deve utilizar e ainda pode ser feita uma análise em termos de eficiência. Por exemplo, será mais eficiente buscar a desejada conta dentre aquelas previamente separadas e ordenadas.

Vejamos mais um aspecto do problema do **Exemplo 1.1**. Uma conta de energia, que se materializa num documento em papel para o consumidor, analogamente, existe de forma eletrônica no sistema da empresa de energia. Como isso é feito? O que seria o "maço de contas" quando estas são representadas eletronicamente? Como representar e organizar? Questionamentos semelhantes surgem sempre que precisamos executar algo no computador, mesmo sendo casos bem conhecidos no mundo real. Para chegarmos às respostas destas questões, devemos analisar a representação dos dados sob duas visões: *do ponto de vista da máquina e do ponto de vista do problema*.

- Do ponto de vista da máquina:

Pelo lado da máquina, os tipos básicos de dados definem como os dados estão organizados na memória do computador (no sentido de como o computador interpreta os bits, especificado pela linguagem de programação). Por exemplo, um inteiro (`int`) em Python é gravado em 4 bytes, um real (`float`) em 8 bytes etc. Nota-se que essas denominações não revelam os significados assumidos no problema, ou seja, um `int` pode servir tanto para informar um código da conta de luz de um consumidor, quanto para registrar a idade de uma

pessoa ou anotar a contagem de um certo conjunto de células em uma lâmina de laboratório, e o mesmo acontece com os demais tipos básicos.

- Do ponto de vista do problema:

"Ver" os dados pelo lado do problema consiste buscar tipos de dados que façam sentido para o problema que se quer resolver e que operações podem ser realizadas com esses tipos. Por exemplo, admitamos que um tipo seja chamado pelo programador de "idade", para representar a idade de uma pessoa. Embora o tipo básico associado seja simplesmente o tipo inteiro, nominalmente, as operações serão realizadas com "idade", como: "determinação" de uma idade a partir de uma data, "nascimento" (determinação de uma idade na data do nascimento – inicialização com idade igual a zero), "aniversário" (incremento da idade em um ano) etc.

Estamos então diante do conceito de *Tipo Abstrato de Dados (TAD)*. É um conceito que visa desligar a definição de tipo de dados das preocupações com as "intimidades" dos dados com o hardware.

O conceito de Tipo Abstrato de Dados consiste na definição de um bloco bem definido de dados e um conjunto de operações especialmente preparadas para a manipulação deste bloco de dados particulares. Os dados são chamados de *atributos* do TAD e o conjunto das operações forma a chamada *Interface* do TAD. Esse conceito surgiu ainda com a programação estruturada, mas deu a base para o paradigma orientado a objetos. Este módulo aborda os tipos abstratos de dados sob os dois paradigmas.

Objetivos

- Abordar o conceito de TAD como estratégia adequada ao estudo das estruturas de dados avançadas;
- Aplicar o conceito de TAD em programação estruturada, experimentando com o uso da linguagem Python;
- Identificar conceito de TAD em Programação Orientada a Objetos e experimentando com classes predefinidas da linguagem Python.

Unidades

- Unidade I.1 - TAD e o Paradigma Imperativo
- Unidade I.2 - TAD e o Paradigma Orientado a Objetos

Unidade I.1

TAD e o Paradigma Imperativo

O conceito de Tipo Abstrato de Dados (TAD), conforme já foi apresentado, permite a definição de novos tipos de dados, formados por um conjunto de atributos e uma interface. No paradigma imperativo, os atributos são reunidos normalmente numa variável composta de maneira heterogênea (usando tipos de dados variados), e as operações que constituem a interface são implementadas por funções.

1.1.1 Atributos e interface

Em um TAD, o objetivo da construção da interface é esconder da aplicação os detalhes do bloco de atributos. Esse bloco é tratado como uma *cápsula*, e as funções da interface são encarregadas de acessar essa cápsula e fornecer os resultados das operações.

Por exemplo, se for definido o TAD "idade" de uma pessoa (ver a introdução deste **Módulo I**), os atributos se resumem na idade (um número inteiro) apenas. A interface (o que manipulará "idade" de uma pessoa) é composta de "determinação", "nascimento", "aniversário", etc.

A partir da definição de um TAD, as variáveis desse tipo podem ser criadas como qualquer outro tipo de variável. Chamamos de *instanciação* a criação da variável do novo tipo. Após ser criada, a variável, ou *instância* do TAD, poderá ser manipulada pelas funções da interface.

Exemplo 1.2

Vamos supor que a distribuidora de energia que cobra a conta do problema do **Exemplo 1.1** seja a *Eletrobrás distr.Alagoas*. No site dessa empresa (<http://www.ceal.com.br/>), encontramos um detalhamento da conta de luz (clique em "*Informações Gerais/Sua Conta*"). A figura a seguir mostra um trecho de conta que contém "campos" (espaços para dados específicos de um consumidor) onde são registrados: nome e endereço do consumidor, datas de emissão do documento, de leituras (anterior, atual e próxima), classe de consumo (residencial, comercial, industrial ou poder público), mês do faturamento, etc.

Nota Fiscal / Conta de Energia Elétrica Nº
Regime especial de impressão autorizado pela Sec. de Fazenda

ceal

Data de emissão da fatura

Para contato com a empresa informe este número

Código Único

Emissão Data Leitura Anterior Data Leitura Atual Data Próxima Leitura Dias de Consumo Apresentação Mês Faturado

Cod. Fat. Classe Ligação Poste Forma Faturamento Motivo FD Número FD

Medidor Leit. Atual Leit. Anterior Constante NPL Cons. Medido Cons. Faturado

Histórico kWh Composição da Tarifa Itens Entregues

Base de Cálculo Aliquota Valor do ICMS Vencimento Valor a Pagar

Reservado ao Fisco

Trecho de uma conta de luz

(Fonte: Adaptada de "sua conta - frente", http://www.ceal.com.br/suaconta_frente.aspx, acessado em 14/06/2012)

Os campos existem como padrão para qualquer consumidor, mas, ao serem preenchidos, produzem como resultado a conta de um consumidor específico.

Vejamos então um paralelo do documento que chega às mãos do consumidor com aquele registrado no ambiente de computação. Uma conta de luz no sistema de computação da empresa deverá ser representada por um tipo abstrato de dado. Vamos chamá-lo de `conta_de_luz`. Portanto, o TAD `conta_de_luz` deverá ter seu conjunto de atributos e ficará completo quando for definida sua interface.

Seu conjunto de atributos é um bloco bem definido com variáveis que, a título de exemplo, denominaremos: `nome`, `endereco`, `data_leit_anterior`, `data_leit_atual`, `data_leit_proxima`, `classe_consumo`, `mes_fatura`,...

A interface do TAD será formada pelo conjunto das operações com uma conta de luz qualquer, a partir de funções que se tornarão as responsáveis por qualquer tipo de acesso aos atributos. Por exemplo, `criar_uma_conta` (preencher campos a partir dos dados de um consumidor), `determinar_valor_a_pagar` (calcular o valor a pagar a partir dos dados do consumidor e valores de tarifas), `exibir_dados_da_conta` (mostrar valores dos campos), etc.

Uma vez construído o tipo abstrato, qualquer conta de luz do sistema será do tipo `conta_de_luz`. Isso equivale a afirmar que qualquer conta de qualquer cliente terá as propriedades definidas para o TAD em questão. Por exemplo, chamemos de `conta_do_Joao` a conta do cliente de nome "João", que mora no endereço tal, do mês tal, etc. `conta_do_Joao` é, portanto, uma instância de `conta_de_luz`.

Registro

Consideremos a tarefa de implementação de TADs no paradigma imperativo. No caso das operações da interface, é bastante usar o bem conhecido conceito de função. Para implementar o bloco de atributos, em caso de dados heterogêneos, utilizaremos *variáveis compostas heterogêneas* (tipos diversos compondo uma mesma estrutura, acessados por um mesmo nome) chamadas de *registros*.

Na disciplina de Algoritmo e Estrutura de Dados I, foram vistas apenas as estruturas de dados correspondentes às *variáveis compostas homogêneas*. A estrutura de dados do tipo registro deverá aumentar as possibilidades de representação dos dados nas aplicações. Para definir um registro, utilizaremos a seguinte sintaxe:

```
Tipo_registro nome_do_tipo:
    Componente1 (tipo básico)
    Componente2 (tipo básico)
    ...
Fim_registro
```

Os componentes de um registro são também chamados de *campos do registro* (de modo análogo aos campos da conta de luz do **Exemplo 1.2**).

Exemplo 1.3

Consideremos o tipo abstrato `conta_de_luz` do **Exemplo 1.2**. Aqui, vamos confundir o nome do TAD com a denominação de seu bloco de atributos (seu registro). O bloco de atributos `conta_de_luz` pode então ser representado nos algoritmos com a seguinte sintaxe:

```
Tipo_registro conta_de_luz:
    nome (caracteres)
    endereco (caracteres)
    data_leit_anterior (caracteres)
    data_leit_atual (caracteres)
    data_leit_próxima (caracteres)
    classe_consumo (caracteres)
    mes_fatura ...
Fim_registro
```

Uma vez definido esse tipo registro, temos um "molde" para criar instâncias (contas de consumidores específicos):

Uma conta qualquer	nome ←	...
	endereco ←	...
	data leit anterior ←	...
	classe consumo ←	...
	mes fatura ←	...
	... ←	...

Querendo-se, por exemplo, criar a `conta_do_Joao`, como sendo do tipo `conta_de_luz` (ou seja, um caso particular de `conta_de_luz`) é bastante usar o gabarito:

<code>conta_do_Joao</code>	<code>nome</code> ←	"João"
	<code>endereco</code> ←	"R. Cafundó, 666... "
	<code>data leit anterior</code> ←	"15/11/2008"
	<code>classe consumo</code> ←	"Residencial"
	<code>mes fatura</code> ←	"Nov"
	<code>...</code> ←	...

Nos algoritmos, para criar uma instância de um tipo abstrato, adotaremos a sintaxe seguinte:

***instância* ← nome do tipo, com a notação default de função.**

Dado o exemplo acima, para criar a `conta_do_Joao` como uma instância de `conta_de_luz`, fazemos:

`conta_do_Joao ← conta_de_luz()`

A partir de então, a `conta_do_Joao` passa gozar de todas as funcionalidades previstas em `conta_de_luz`. Para acessar os campos do registro, é só usar o operador "." (ponto), da seguinte maneira:

Para preencher o campo no nome: `conta_do_Joao.nome ← "João";`

Para atribuir o endereço: `conta_do_Joao.endereco ← "R.Cafundó, 666... ";`

Para o mês de referência desta da conta: `conta_do_Joao.mes_fatura ← "Nov",`

e assim por diante.

Vetor de registros

Na construção dos tipos abstratos, podemos também combinar registros com estruturas de dados clássicas como vetores e matrizes. Em outras palavras, é possível construir, por exemplo, um vetor em que cada componente é do tipo registro.

Retomemos então o caso da conta de luz registrada no sistema da empresa de energia do **Exemplo 1.2**. No **Exemplo 1.3**, essa conta foi representada por um registro. Podemos agora pensar em como o conjunto de todas as contas estaria organizado.

Vimos no caso das contas em papel do **Exemplo 1.1** como o sequenciamento daqueles documentos permitiu disciplinar seu manuseio. De modo equivalente, as contas em formato eletrônico também podem ser "arrumadas" no computador.

Dentre os vários modos para montagem de uma sequência de contas no computador, tomemos uma estrutura já conhecida por nós que é o vetor, ou seja, na memória do computador, podemos montar um vetor em que cada componente é uma conta de um consumidor.

Exemplo 1.4

Consideremos o registro `conta_de_luz` do **Exemplo 1.3**. Se for declarado, por exemplo, o vetor `vetor_de_contas[qtde]` com uma quantidade `qtde` de contas de luz, teremos os seguintes elementos do tipo `conta_de_luz`: `vetor_de_contas[0]`, `vetor_de_contas[1]`, `vetor_de_contas[2]`, etc. Isto é:

vetor_de_contas[0]	nome ←	"Maria"
	endereco ←	"Rua X"
	data leit anterior ←	"..."
	...etc. ←	...
vetor_de_contas[1]	nome ←	"Joaquim"
	endereco ←	"Rua Y"
	data leit anterior ←	"..."
	...etc. ←	...
...etc.		

Assim feito, `vetor_de_contas[0].nome` armazenará o nome do primeiro consumidor, `vetor_de_contas[0].endereco` conterá seu endereço e assim por diante. Para o segundo consumidor, teremos: `vetor_de_contas[1].nome`, `vetor_de_contas[1].endereco` e assim por diante. O mesmo vai acontecer com os demais consumidores.

Exercício de autoavaliação

Com base nos conhecimentos construídos nesta subunidade, resolva a seguinte questão e discuta sua resposta no fórum dos conteúdos da semana: O quadro abaixo representa um cliente numa agenda telefônica de um pequeno empresário:

Cliente
Nome: _____ Fone: _____

Digamos que cada linha da agenda seja reservada para apenas um cliente e a qualquer momento um novo cliente pode ser anotado. Considere agora que se deseja montar uma agenda eletrônica, utilizando um princípio análogo ao da agenda em papel.

a) Escreva a representação algorítmica um cliente definindo o tipo registro `Cliente`, com os campos `Nome` e `Fone`.

b) Segundo a modelagem do problema, as declarações seguintes tornam-se equivalentes à reserva das linhas X e Y da agenda em papel (X e Y serão agora clientes na memória do computador):

```
X ← Cliente ()
```

```
Y ← Cliente ()
```

Escreva comandos de atribuição no formato algorítmico para preencher os clientes X e Y na memória do computador, sabendo-se que os dois clientes são os seguintes:

Cliente X
Nome: <u>Joaquim</u> Fone: <u>8888-7654</u>

Cliente Y
Nome: <u>Joaquina</u> Fone: <u>8999-4567</u>

c) Vamos considerar agora que a agenda cabe inteiramente na memória do computador em uma estrutura do tipo vetor. Chamemos este vetor de `vetorDeClientes` e corresponde fisicamente à coleção das linhas da agenda em papel. Assim, `vetorDeClientes[0]` será o cliente anotado na primeira linha, `vetorDeClientes[1]`, o cliente da segunda linha, e assim por diante. Escreva os comandos de atribuição equivalentes às anotações do nome e do telefone do cliente da décima quinta linha da agenda, com os dados: "João da Silva", "9999-4455".

d) Podemos dizer que o vetor de clientes do item anterior representa a agenda eletrônica. Mas, falta alguma coisa para a analogia com a agenda em papel ficar completa. Faltam as operações. Tente listar pelo menos duas operações que podem ser feitas no vetor e estão associadas a alguma operação da agenda em papel (por exemplo, uma operação seria "inserir" um cliente na agenda).

1.1.2 Experimentação

O objetivo do exemplo seguinte é, utilizando um caso simples, avaliar como é possível concretizar resultados usando os conceitos mostrados na subunidade anterior.

Exemplo 1.5

Uma determinada loja mantém um cadastro das mercadorias que comercializa a fim de dar suporte aos movimentos como: vendas, controle de estoque etc. Aqui, de modo simplificado, vamos assumir que cada mercadoria tem somente: código, denominação, preço de compra e preço de venda. Consideremos o seguinte problema:

Alimentar o cadastro de mercadorias na memória do computador e escrever o código, a denominação e o lucro (em %) de cada mercadoria que apresentou lucro superior ao lucro médio das mercadorias como um todo.

Neste exemplo, claramente, identificamos pelo menos duas entidades significativas para a solução dos problemas relativos a esse setor da loja. São elas: cadastro e mercadoria. Decidimos então pela construção dos tipos `cadastro` e `mercadoria`.

O tipo abstrato `mercadoria`

O conjunto de atributos consta de: código (*cod* - um número inteiro), denominação (*nome* - uma cadeia de caracteres), preço de compra (*pc* - um número real) e preço de venda (*pv* - um número real). Em se tratando de dados heterogêneos, o bloco de atributos será representado pelo registro:

```
Tipo_registro mercadoria:
    cod (um número inteiro)
    nome (uma cadeia de caracteres)
    pc (um número real)
    pv (um número real)
Fim_registro
```

A interface constará de funções destinadas à manipulação do registro *mercadoria*, isto é, no problema em questão, será preciso ler os dados de cada nova mercadoria, determinar o lucro de uma mercadoria, exibir os atributos de uma mercadoria. Essas operações podem ser representadas, respectivamente, pelas funções seguintes:

A função *NovaMercadoria()* cria e retorna um registro do tipo *mercadoria* preenchidos antes seus campos via teclado.

```
Defina NovaMercadoria():
    mc ← mercadoria()
    Escrever "Digite o código:"
    ler mc.cod
    Escrever "Digite a denominação:"
    ler mc.nome
    Escrever "Digite o preço de compra:"
    ler mc.pc
    Escrever "Digite o preço de venda:"
    ler mc.pv
    retornar mc
Fim_função
```

A função *LucroMercadoria()* recupera os campos *pc* e *pv* de uma mercadoria (passada como parâmetro) e usa esses valores para calcular e retornar seu lucro (porcentagem da diferença entre preço de compra e preço de venda em relação ao preço de compra):

```
Defina LucroMercadoria(mc):
    lucro ← (mc.pv - mc.pc) * 100 / mc.pc
    retornar lucro
Fim_função
```

A função *EscreveMercadoria()* escreve o código, a denominação e o lucro de uma mercadoria passada como parâmetro.

```
Defina EscreveMercadoria(mc):
    Escrever "Código:", mc.cod
    Escrever "Denominação:", mc.nome
    Escrever "Lucro:", LucroMercadoria(mc), "% "
Fim_função
```

O tipo abstrato cadastro

Os atributos do `cadastro` são os próprios dados das mercadorias, ou seja, podemos representar um `cadastro` por um vetor cujos elementos são do tipo `mercadoria`.

Quanto à interface, vão ser consideradas as seguintes operações: criar cadastro e determinar lucro médio. As respectivas funções são as seguintes:

A função `CriaCadastro()` cria primeiramente o vetor de mercadorias `cad[qt]` com a quantidade `qt` passada como parâmetro. Em seguida, preenche `cad[qt]` com os dados das mercadorias digitados pelo usuário.

```
Defina CriaCadastro(qt):
  Criar cad[qt]
  para i ← 0...qt-1, faça:
    cad[i] ← NovaMercadoria()
  retornar cad
Fim_função
```

A função `CalculaLucroMedio()` recebe como parâmetro o vetor com as mercadorias e retorna a média dos lucros (acumula em `soma` a soma dos lucros e divide pela quantidade de mercadorias). *Observação:* A fim de tornar mais prática a passagem do vetor como parâmetro, consideremos que se trata de uma estrutura de alto nível e permite o uso de uma função `Tamanho()` que recupera a quantidade de elementos do vetor.

```
Defina CalculaLucroMedio(cad):
  soma ← 0.0
  qt ← Tamanho(cad)
  para i ← 0...qt-1, faça:
    soma ← soma + LucroMercadoria(cad[i])
  retornar soma/qt
Fim_função
```

Finalmente, a solução do problema acima pode ser dada pelo algoritmo abaixo:

Algoritmo

```
{Declarar o tipo abstrato mercadoria}
{Declarar o tipo abstrato cadastro}
1 escrever "Digite a quantidade de mercadorias:"
2 ler qtde
3 Cad ← CriaCadastro(qtde)
4 LucroMedio ← CalculaLucroMedio(Cad)
5 para i ← 0...qtde-1, faça:
  se LucroMercadoria(Cad[i]) > LucroMedio então:
    EscreveMercadoria(Cad[i])
```

Fim-Algoritmo

Laboratório - TAD

Objetivos

Implementar o conceito de TAD segundo o paradigma estruturado, exercitando com registro e vetor de registros;

Utilizar os recursos da linguagem Python para testes dos algoritmos dessa subunidade.

Recursos e implementação

A tarefa agora é combinar os conhecimentos adquiridos até este momento sobre funções, tipos de dados básicos e estruturados (homogêneos e heterogêneos) com o objetivo de implementar os tipos abstratos de dados. Usaremos para tanto a linguagem Python sob o paradigma estruturado.

Recapitulando um pouco, adotamos a implementação de vetores a partir do recurso de *lista* definido em Python. Na verdade, as listas de Python têm propriedades avançadas associadas ao paradigma orientado a objetos, porém não as exploraremos neste momento. Falta-nos apenas verificar como será representado um registro em Python.

De antemão, devemos saber que Python não tem explicitamente uma estrutura para representar um registro exclusivamente. Então, faremos aqui uma adaptação usando a seguinte sintaxe:

```
class nome_do_tipo:
    Componente1 = valor_inicial
    Componente2 = valor_inicial
    ...
```

Observação: Essa notação é uma representação simplificada da estrutura adequada ao paradigma orientado a objetos (do qual falaremos na próxima unidade) onde, aqui, usaremos apenas para a definição dos atributos.

Em um TAD, *nome_do_tipo* será o nome do novo tipo de dado (abstrato), que, uma vez definido, poderemos criar suas instâncias através da seguinte sintaxe:

```
instância = nome_do_tipo()
```

Tomemos o problema dado no **Exemplo 1.5**. Repetindo o texto: Alimentar o cadastro de mercadorias de uma loja na memória do computador e escrever o código, a denominação e o lucro (em %) de cada mercadoria que apresentou lucro superior ao lucro médio das mercadorias como um todo.

Vamos explorar a solução desse problema em duas fases, realizando os dois experimentos seguintes.

Experimento 01

Começamos pela implementação e teste do tipo abstrato *mercadoria*. Criar um diretório no disco com o nome `L112_01`. Nesse diretório, serão colocados os arquivos

envolvidos no corrente experimento. Iniciar o IDLE e criar o arquivo `mercadoria.py` com as linhas de código a seguir:

```
#Tipo Abstrato 'mercadoria'
#ATRIBUTOS:-----
class mercadoria: #Implementação do 'tipo_registro mercadoria'
    cod = 0
    nome = ' '
    pc = 1.0
    pv = 1.0
#INTERFACE:-----
def NovaMercadoria(): #cria uma mercadoria em particular
    mc = mercadoria()
    mc.cod = int(input('Digite o código:'))
    mc.nome = input('Digite a denominação:')
    mc.pc = float(input('Digite o preço de compra(R$):'))
    mc.pv = float(input('Digite o preço de venda(R$):'))
    return mc

def LucroMercadoria(mc): #calcula e retorna o lucro de uma merc.
    lucro = (mc.pv - mc.pc) * 100.0 / mc.pc
    return lucro

def EscreveMercadoria(mc): #mostra uma mercadoria em particular
    print ('Código:', mc.cod)
    print ('Denominação:',mc.nome)
    print ('Lucro: %.1f%%' % LucroMercadoria(mc))
#-----
```

Observação: Na função `EscreveMercadoria()`, o comando

```
print ('Lucro: %.1f%%' % LucroMercadoria(mc))
```

corresponde ao formato especial do comando `print` (consultar bibliografia do curso) com o objetivo de conseguir a restrição a uma casa decimal (através do código `'%.1f'`) e escrever o símbolo de porcentagem junto ao valor do lucro (através do código `'%%'`).

O arquivo `mercadoria.py` possui as propriedades de um *módulo Python*, que exploraremos a seguir. Uma vez construído o módulo, ele pode ser testado antes mesmo da conclusão do programa, sendo isso uma vantagem considerável.

Vamos realizar os testes de `mercadoria.py`, aproveitando a interatividade do interpretador Python. No *prompt* de comandos, na mesma seção de edição do módulo `mercadoria.py`, fazemos a inclusão do módulo com o comando seguinte:

```
>>> from mercadoria import *
```

Observação: Este comando deve encontrar o referido módulo no disco. Para garantir isso, uma das maneiras é usar a mesma seção em que foi editado o módulo. Outra é, a qualquer

momento, no *Windows Explorer*, usar o botão direito do mouse sobre `mercadoria.py` e escolher a opção 'Edit with IDLE'.

Criemos uma mercadoria, chamada `merc`, testando com os seguintes dados, respectivamente: 15, "caneta hidrográfica", 6.20 e 7.25:

```
>>> merc = NovaMercadoria()  
Digite o código:15  
Digite a denominação: caneta hidrográfica  
Digite o preço de compra (R$):6.20  
Digite o preço de venda (R$):7.25
```

Para testar a correção do preenchimento dos campos da mercadoria `merc`, vamos imprimi-los:

```
>>> print (merc.cod)  
15  
>>> print (merc.nome)  
caneta hidrográfica  
>>> print (merc.pc)  
6.2  
>>> print (merc.pv)  
7.25
```

Para testar o cálculo do lucro através da função `LucroMercadoria()`:

```
>>> print (LucroMercadoria(merc))  
16.935483871
```

Ou, para escrever com uma casa decimal e com o símbolo de porcentagem (%):

```
>>> print ('%.1f%%' % LucroMercadoria(merc))  
16.9%
```

Para testar a função `EscreveMercadoria()`:

```
>>> EscreveMercadoria(merc)  
Código: 15  
Denominação: caneta hidrográfica  
Lucro: 16.9%
```

Também podemos testar o módulo `mercadoria.py`, fazendo um programa simples que envolva as funções do TAD, como o seguinte.

Usando o IDLE, criar o arquivo `TestaMercadoria.py` e gravar no diretório `L112_01`.

```
from mercadoria import *  
print ('Teste do TAD "mercadoria"')  
print ('Dados de uma mercadoria:')  
m = NovaMercadoria() #Cria uma mercadoria  
print ('-----')  
EscreveMercadoria(m) #e mostra os dados da mercadoria criada
```

Digitar os dados já testados acima (os dados da "caneta hidrográfica"):

```
>>>
```

```

Teste do TAD "mercadoria"
Dados de uma mercadoria:
Digite o código: 15
Digite a denominação: caneta hidrográfica
Digite o preço de compra(R$): 6.20
Digite o preço de venda(R$): 7.25
-----
Código: 15
Denominação: caneta hidrográfica
Lucro: 16.9%
>>>

```

Continuemos com a implementação associada à resolução do problema do **Exemplo 1.5**. Uma vez definido o que é uma *mercadoria*, a solução algorítmica institui o que é o *cadastro* de mercadorias. O *cadastro* será representado por um vetor de mercadorias na memória do computador, ou seja, cada elemento do vetor é uma *instância* de *mercadoria*.

Em Python, pelo que aprendemos, podemos criar um vetor já com a quantidade desejada de elementos, aplicando concatenação de listas. Por exemplo, sabendo-se previamente a quantidade, *qt*, de elementos do vetor de mercadorias, *cad*, este pode ser inicializado com os valores *default* de cada mercadoria da seguinte maneira:

```
cad = [mercadoria()]*qt
```

em que *mercadoria()* cria uma instância do tipo abstrato *mercadoria*. A seguir, o TAD *cadastro* será criado e testado.

Experimento 02

Para implementação do tipo abstrato *cadastro*, vamos criar um diretório com o nome `L112_02` onde serão colocados os arquivos envolvidos neste experimento. Em seguida, o arquivo `mercadoria.py`, criado no **Experimento 01**, deverá ser o primeiro a ser copiado para este diretório.

Iniciar o IDLE, criar o arquivo `cadastro.py`, e gravar no mesmo diretório, editando as seguintes linhas de código:

```

# Tipo Abstrato 'cadastro'
from mercadoria import *
#ATRIBUTOS:-----
# Obs.: Os dados das mercadorias estarão num vetor a ser criado
# na operação 'CriaCadastro()'

#INTERFACE:-----
def CriaCadastro(qt):
    cad = [mercadoria()]*qt #cria vetor local com mercs default
    for i in range(qt):     #subst mercs default lendo do tecl.
        cad[i] = NovaMercadoria()
    return cad              #devolve o vetor preenchido

```

```

def CalculaLucroMedio(cad):
    soma = 0.0
    qt = len(cad)          #obtém a quantidade de elementos de cad
    for i in range(qt):    #calcula a soma de todos os lucros
        soma += LucroMercadoria(cad[i])
    return soma/qt        #retorna a média aritmética dos lucros
#-----

```

Obs.: A fim de dar maior visibilidade ao processo de construção, a função `CalculaLucroMedio()` foi implementada seguindo ao máximo o algoritmo mostrado no **Exemplo 1.5**. As simplificações permitidas pela linguagem Python serão aplicadas oportunamente.

Para realizar os testes interativamente (no *prompt* de comandos) na mesma seção (após a edição do módulo `cadastro.py`), fazemos inicialmente a inclusão deste módulo com o comando seguinte:

```
>>> from cadastro import*
```

Experimentemos um cadastro com apenas duas mercadorias. Por exemplo, respectivamente:

15, "caneta hidrográfica", 6.20,7.25 e

17, "borracha", 0.43, 0.51

```

>>> Cad = CriaCadastro(2)
Digite o código:15
Digite a denominação:caneta hidrográfica
Digite o preço de compra(R$):6.2
Digite o preço de venda(R$):7.25
Digite o código:17
Digite a denominação:borracha
Digite o preço de compra(R$):0.43
Digite o preço de venda(R$):0.51

```

Conferimos então se os registros das mercadorias estão corretos para mostrar os dados da primeira mercadoria (índice 0 do vetor):

```

>>> EscreveMercadoria(Cad[0])
Código: 15
Denominação: caneta hidrográfica
Lucro: 16.9%

```

Para mostrar os dados da segunda mercadoria (índice 1 do vetor):

```

>>> EscreveMercadoria(Cad[1])
Código: 17
Denominação: borracha
Lucro: 18.6%

```

A última função a ser testada é a que calcula o lucro médio das mercadorias:

```
>>> CalculaLucroMedio(Cad)
```

```
17.770067516879223
>>>
```

Para escrever a solução final do problema proposto (mostrar os dados de cada mercadoria que apresentou lucro superior ao lucro médio das mercadorias como um todo) vamos optar pela edição do arquivo `cadastroLoja.py`. Criar e gravar este arquivo no diretório `L112_02`, escrevendo o seguinte código:

```
from cadastro import *
qtde = int(input('Digite a quantidade de mercadorias: '))
Cad = CriaCadastro(qtde)
LucroMedio = CalculaLucroMedio(Cad)
print ('Mercadorias com lucro superior à média')
for merc in Cad:
    if LucroMercadoria(merc) > LucroMedio:
        EscreveMercadoria(merc)
```

Usando os mesmos dados das mercadorias acima, obtemos:

```
>>>
Digite a quantidade de mercadorias: 2
Digite o código:15
Digite a denominação:caneta hidrográfica
Digite o preço de compra(R$):6.2
Digite o preço de venda(R$):7.25
Digite o código:17
Digite a denominação: borracha
Digite o preço de compra (R$):0.43
Digite o preço de venda (R$):0.51
Mercadorias com lucro superior à media
Código: 17
Denominação: borracha
Lucro: 18.6%
>>>
```

Observação:

No programa `cadastroLoja.py` e em outras situações em que vetores foram manipulados, o acesso aos seus elementos tem sido feito percorrendo os índices do vetor. Esta é a maneira aceita pela maioria das linguagens de programação e, por isso mesmo, tem sido preferida até agora, porém, em Python, temos a liberdade de acessar diretamente os próprios componentes do vetor. Por exemplo, no comando seguinte (do `cadastroLoja.py`),

```
for merc in Cad:
    if LucroMercadoria(merc) > LucroMedio:
        EscreveMercadoria(merc)
```

onde a variável auxiliar `merc` representa uma mercadoria genérica de `Cad` (um vetor, que, como sabemos, é uma lista em Python)

Exercício de autoavaliação

Resolva as questões abaixo com base nos conhecimentos construídos nesta subunidade. Discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Escreva um módulo Python para implementar o TAD `Cliente` do exercício de autoavaliação da **Subunidade 1.1.1**. Os atributos, como foram definidos, são `Nome` e `Fone`. A interface terá apenas as funções `NovoCliente()` (é sem parâmetros e retorna um `Cliente` com seus dados preenchidos via teclado) e `EscreveCliente()` (escreve na tela o nome e o telefone de um `Cliente` passado como parâmetro). Faça um teste do módulo com operações simples, por exemplo, criando os clientes X e Y e, em seguida, escrevendo seus dados na tela.

2 - Usando o tipo abstrato `Cliente` do item anterior, elabore a função `CriaAgenda(qt)` que cria um vetor de clientes (com nomes e telefones lidos do teclado), dada a quantidade, `qt`, de clientes. Faça um teste dessa função da maneira como fizemos acima com a função `CriaCadastro(qt)` no **Experimento 02**.

3 - Faça um programa que lê do teclado a quantidade de clientes, preenche o vetor de clientes (usando a função do item anterior), e, busca nesse vetor e escreve os dados dos clientes cujos números de telefone se iniciam com os caracteres '99'.

4 - Elabore um único programa que reúne as propriedades das ações dos itens anteriores. O programa deve ler a agenda inicialmente e, em seguida, disponibilizar os dados para consulta, onde o usuário digita o que se pede seguidamente, encerrando a consulta ao digitar o caractere '0' (zero). Se o usuário digitar apenas os dois primeiros dígitos de um número de telefone, o programa mostra uma lista com os dados dos clientes cujos números de telefone tem tal propriedade. Se digitar um número de telefone completo (com oito dígitos, sem espaços), o programa mostrará os dados do cliente respectivo ou apenas a mensagem "Cliente não localizado", se o tal número não existir.

5 - Certa loja fez uma listagem das vendas realizadas por seus vendedores, deixando a mesma para consulta pelo gerente. De cada vendedor são anotados seu nome e o montante de vendas realizadas pelo mesmo. O vendedor que mais vendeu receberá um prêmio de melhor vendedor do ano. Os que venderam abaixo da média aritmética das vendas dentre todos os vendedores deverão participar de novo treinamento.

l) Implemente em Python um TAD chamado `Vendedor` com as propriedades abaixo.

Atributos:

`cod` (código do vendedor - um número inteiro);

`nome` (nome do vendedor - string de 20 dígitos);

`vendas` (montante de vendas realizadas pelo vendedor em R\$ - um número real),

A interface deve conter as funções:

`novoVendedor()` - Sem parâmetros, retorna um `Vendedor` lido pelo teclado;

`mostraVendedor(v)` - Escreve código, nome e total de vendas em R\$ do vend. `v`;

II) Considere que a listagem dos vendedores, **L**, é um vetor onde cada elemento é do tipo `Vendedor`. Elabore um programa que lê a quantidade de vendedores da loja, preenche **L** via teclado, escreve os dados do vendedor que receberá o prêmio de melhor vendedor e os dados de cada vendedor que participará de novo treinamento.

Unidade I.2

TAD e o Paradigma Orientado a Objetos

O conceito de *Orientação a Objetos* é resultado de uma evolução dos tipos abstratos de dados já existentes na programação estruturada. Sobre orientação a objetos, em linhas gerais, podemos esboçar o seguinte. Para se resolver um problema sob este paradigma, o mundo real deve ser visto como um conjunto de elementos, chamados de *objetos*, que devem interagir entre si. Ou seja, resolver um problema consiste em construir objetos (um objeto tem dados e ações na sua definição) para em seguida promover a "cooperação" entre os mesmos. Diante de um problema pergunta-se: Que objetos estão em jogo e como eles se combinam para resolvê-lo?

Esses objetos são abstratos. Isto é, são coisas que existem apenas logicamente na solução do problema. Objetos precisam de algo que o definam para poder existirem. Primeiramente são definidas suas propriedades reunindo as características no que é chamado de *classe*. Assim acontece na vida real. Por exemplo, antes de buscar materiais, ferramentas, etc. para se construir uma cadeira, precisamos saber *o que é uma cadeira!* Sabendo-se quais são os requisitos que definem uma cadeira, pode-se, a partir daí, construir as cadeiras fisicamente. Desse modo, aproveitando o exemplo:

"Uma cadeira" é a *classe*

e "A cadeira" é o *objeto* (algo específico, classificado como "cadeira").

Outros exemplos:

Classe: Carro → Objetos: Palio, Gol, Celta, ...

Classe: Aluno → Objetos: Ana, Maria, Pedro, ...

A relação com os tipos abstratos estudados na **Unidade I.1** é imediata. Em orientação a objetos o tipo de dado abstrato é a classe e o objeto é uma instância da mesma (uma instância do tipo abstrato). As funções da interface passam a ser chamadas de *métodos* e representam os *comportamentos* dos objetos. Os atributos da classe representam o *estado* do objeto.

As classes, no entanto, incorporam características que as tornam avançadas em relação aos tipos abstratos convencionais. Para ilustrar, podemos mencionar uma destas características que é a *herança* entre classes (do que trata o **Exemplo 1.6**). A herança consiste no fato de que determinadas classes (chamadas *subclasses*) herdam as características de outras (chamadas *superclasses*). As subclasses são casos particulares das superclasses (classes existentes anteriormente). Há, portanto, uma hierarquia entre as classes.

Exemplo 1.6

Consideremos a existência das classes denominadas de: *Mamífero*, *Carnívoro*, *Herbívoro* e *Onívoro*, cuja hierarquia é descrita no quadro seguinte:

Classe: Mamífero Características: <i>temMamas</i> ; <i>Aleitar()</i>		
Classe: Carnívoro Características: <i>Mamífero</i> ; <i>Caçar()</i>	Classe: Herbívoro Características: <i>Mamífero</i> ; <i>BuscarVegetais()</i>	Classe: Onívoro Características: <i>Mamífero</i> ; <i>Alimentar()</i>

(*Observação*: As denominações dessas classes referem-se a animais mamíferos, onde, como sabemos, os carnívoros alimentam-se exclusivamente de carne, os herbívoros alimentam-se exclusivamente de vegetais e os onívoros alimentam-se tanto de carne quanto de vegetais).

A classe *Mamífero* tem como características um atributo, *temMamas*, e um método, *Aleitar()*. Uma vez criada a classe *Mamífero*, esta será a superclasse em relação às classes *Carnívoro*, *Herbívoro* e *Onívoro*. Isto significa que estas últimas (as subclasses) herdam as características da primeira.

No processo de construção, as subclasses são obtidas por acréscimo ou por adaptação de características da superclasse respectiva. Ou seja, os objetos das classes *Carnívoro*, *Herbívoro* e *Onívoro*, já recebem previamente o atributo *temMamas* e o método *Aleitar()*, bastando apenas, no caso, serem acrescentados os comportamentos: *Caçar()*, se for carnívoro, *BuscarVegetais()*, se for herbívoro, e *Alimentar()*, se for onívoro (este comportamento não especifica se o alimento será carne ou vegetal).

1.2.1 Atributos e interface

Em orientação a objetos, a resolução de um problema se inicia pela identificação e elaboração das classes (com a definição de atributos e métodos). A partir de então, realiza-se a construção dos objetos como instâncias destas. Os objetos possuem comportamentos e são ativados mediante a *chamada* de seus métodos (suas funções da interface). A ação de provocar a ativação de um método é chamada de *mensagem*. Portanto, a solução do problema é resultado da interligação de mensagens entre os objetos adequados.

Fizemos acima um paralelo declarando que TAD no paradigma orientado a objetos é associado a classe, sabendo-se que, a rigor, classe é mais que apenas um tipo de dado. Falamos em atributos da classe como os atributos do TAD e interface da classe como a interface do TAD. Veremos a seguir exemplos que descrevem operacionalmente esta afirmação.

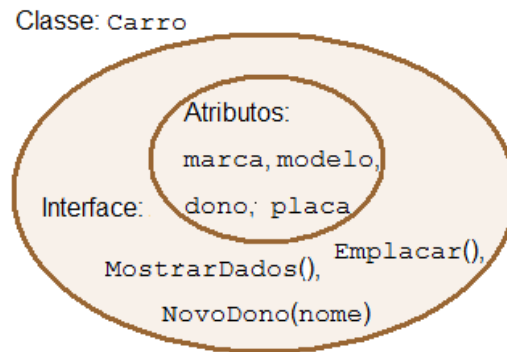
Classes e objetos

Uma particularidade fundamental da classe é que os atributos (dados) e a interface (funções) estão reunidos numa mesma cápsula (e não somente os atributos, como em um TAD

genérico). Isto significa que dados e funções são referenciados por um mesmo nome (o nome da classe). Na prática, isto quer dizer que dados e funções *são acessados como os campos de um registro*. Podemos conferir isto no **Exemplo 1.7** a seguir.

Exemplo 1.7

Tomemos a classe `Carro` representada no gráfico abaixo:



Onde `marca`, `modelo`, `dono` e `placa` são os atributos e os métodos são:

`MostrarDados()`: Escreve todos os dados de um objeto desta classe;

`Emplacar()`: Providencia o emplacamento e atribui um valor para placa;

`NovoDono(nome)`: Constrói e devolve um novo objeto `Carro` com os mesmos dados, mas com novo dono chamado `nome`.

Um objeto `Carro`, chamemos de, por exemplo, `meuCarro`, pode ser criado assim:

```
meuCarro ← Carro("Volkswagen", "fusca", "eu", " ")
```

(cria um carro ainda sem o valor da placa). Isto é, o objeto é criado com o seguinte estado inicial:

```
meuCarro.marca ← "Volkswagen",
```

```
meuCarro.modelo ← "fusca",
```

```
meuCarro.dono ← "eu",
```

```
meuCarro.placa ← " ",
```

O acesso aos métodos acontece de modo semelhante ao acesso aos atributos. Executando-se, por exemplo,

```
meuCarro.MostrarDados()
```

(diz-se que uma mensagem foi enviada ao objeto `meuCarro`) serão escritos os dados atuais de `meuCarro`, ou seja, `marca: "Volkswagen"`, `modelo: "fusca"`, `dono: "eu"`, `placa: " "`.

Executando-se

```
meuCarro.Emplacar(),
```

este comando vai alterar o campo `placa` com um valor definido pelo departamento de trânsito.

Vamos supor que tenham atribuído para a placa o valor `"MMM-5678"`. Executando-se novamente `meuCarro.MostrarDados()` (após a mensagem `emplacar()`) a saída será:

```
marca:"Volkswagen", modelo:"fusca", dono:"eu", placa:"MMM-5678".
```

Vejamus uma situação em que `meuCarro` seja transferido para outro dono. Então, a mensagem de novo dono deve ser enviada a `meuCarro` (deverá ser copiado para outro objeto `Carro`, mas com novo dono). Chamemos esse outro objeto `Carro` de `seuCarro` e o dono chamaremos de "você". Assim, o comando será:

```
seuCarro ← meuCarro.NovoDono("você")
```

Dessa maneira, se for executado o comando

```
seuCarro.MostrarDados(),
```

o resultado será:

```
marca:"Volkswagen", modelo:"fusca", dono:"você", placa:" MMM-5678"
```

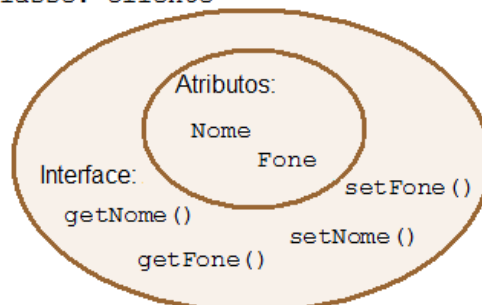
Numa aplicação, a partir daí, o objeto `meuCarro` não será mais referido (e pode ser destruído). O objeto que receberá referências daqui por diante é o `seuCarro`.

Observação: Neste curso não será explorada a construção de classes. O objetivo aqui é a compreensão do paradigma de programação orientada a objetos visando apenas o *uso de classes*. Continuaremos com a programação estruturada, mas os conhecimentos sobre orientação a objetos permitirão o uso de classes predefinidas da linguagem Python.

Exercício de autoavaliação

Resolva a questão seguinte e discuta sua resposta no fórum dos conteúdos da semana: Considere que cada linha da agenda telefônica de um pequeno empresário contém nome e telefone de um único cliente. Desejando-se montar uma agenda eletrônica sob o mesmo princípio da agenda em papel, o tipo abstrato `Cliente` pode ser criado. Vamos considerar que o cliente aqui será modelado pela classe `Cliente`, descrita no quadro abaixo:

Classe: `Cliente`



Os métodos `getNome()` e `getFone()` acessam os atributos e retornam seu respectivo valor. Os métodos `setNome(nome)` e `setFone(fone)` recebem um dado como parâmetro (respectivamente, `nome` e `fone`) e inserem como valor do respectivo atributo.

Observação: Na construção de classes, a fim de evitar o acesso direto aos atributos (respeitando o *encapsulamento*), criam-se métodos para realizar esta tarefa. Os prefixos em inglês nos nomes dos métodos acima significam, respectivamente, "obter" (get) valor e "ajustar, inserir" (set) novo valor.

Sabe-se que, a qualquer momento, um novo cliente pode ser anotado na agenda em papel e isto equivale à criação de um objeto `Cliente` na agenda eletrônica. Por exemplo, os espaços para dois clientes, X e Y, são criados criando-se os objetos `Cliente X` e `Y` usando os comandos (formato algorítmico):

```
X ← Cliente()
```

```
Y ← Cliente()
```

a) Usando os métodos adequados da citada classe (e exemplificando com nomes e fones de sua escolha) escreva os comandos algorítmicos para definir valores para os atributos de X e Y.

b) Escrever comandos algorítmicos de escrita para mostrar nome e telefone dos clientes X e Y, usando os métodos adequados para acessar os respectivos atributos.

1.2.2 Experimentação

As linguagens de programação orientadas a objetos normalmente estão construídas sobre uma base de classes predefinidas. Python é uma destas linguagens. O fato marcante, já comentado em outra oportunidade, é que esta linguagem não obriga que a programação seja orientada a objetos. No nosso caso, a estamos explorando sob o paradigma estruturado (não nos custa repetir que o objetivo desta subunidade é criar condições para compreensão e uso de classes predefinidas de Python).

Tudo em Python pertence a alguma classe. As variáveis, mesmo que sejam de tipos básicos, possuem atributos (por exemplo, seu valor numérico, ou, literal) e métodos (por exemplo, as operações que podem ser realizadas com seus atributos numéricos, ou, literais, respectivamente). Python possui classes predefinidas que modelam desde estruturas básicas até arquivos em disco, sites da Internet, etc. Estão disponíveis também comandos que verificam atributos e métodos das classes (veremos o uso dos comandos `dir()` e `help()` no próximo laboratório).

A seguir, vamos selecionar classes de nosso interesse, algumas delas já aplicadas por nós nos programas estruturados. O que estamos realizando é o aproveitamento da versatilidade da linguagem Python, fazendo uso estruturado de algo que, internamente, é orientado a objetos. Mais adiante compreenderemos melhor porque isto é possível.

Laboratório - Uso de classes predefinidas de Python

Objetivos

Aplicar o conceito de TAD segundo o Paradigma Orientado a Objetos;

Aplicar os conhecimentos sobre orientação objetos no sentido de explorar classes predefinidas na linguagem Python.

Recursos e implementação

Conforme foi citado acima, mesmo os tipos básicos de Python são objetos de classes predefinidas. Nos experimentos seguintes verificaremos determinadas classes, julgadas como mais diretamente ligadas ao escopo desta disciplina, sendo de grande utilidade ao longo deste curso.

A classe `str`. Como exemplo de classe que já utilizamos, mas não havíamos tratado como tal, tomemos o caso do tipo *string* de Python (revisar textos da disciplina Alg. e Estr. de Dados I sobre *expressões literais*), que na definição interna corresponde à classe `str`.

Experimento 01

Usando o interpretador interativamente digitemos o código seguinte. Primeiramente é criada a variável `meuLugar` cujo conteúdo é `'brasil'`, e é do tipo `str` (string) de Python.

```
>>> meuLugar = 'brasil'
>>> type (meuLugar)
<class 'str'>
>>>
```

O comando revela que existe uma classe chamada `str` e a operação acima cria uma instância da mesma, chamada `meuLugar`. Portanto, `meuLugar` tem atributos e métodos para sua manipulação. Por exemplo, existe o método chamado `capitalize()` que produz uma nova string onde o primeiro caractere em sendo uma letra será tornada maiúscula. O uso, de acordo com o nosso aprendizado, será `meuLugar.capitalize()`:

```
>>> print (meuLugar)          #valor anterior da variável meuLugar
brasil
>>> print (meuLugar.capitalize())  #valor retornado pelo método
Brasil
```

Querendo-se substituir o conteúdo da atual variável por aquele com primeira letra maiúscula, deve ser feito:

```
>>> print (meuLugar)          #valor antigo ainda não alterado
brasil
>>> meuLugar = meuLugar.capitalize()      #alteração do valor
>>> print (meuLugar)          #valor atual da variável meuLugar
Brasil
```

A classe `str` possui outros métodos além do `capitalize()` (Agora sabemos que `str` é uma classe, mas podemos continuar chamando de "tipo", pois linguagem Python permite!). Para consultar a disponibilidade de métodos de uma dada classe podemos usar o comando `dir()`, colocando como parâmetro o nome da classe desejada. Se fizermos isto para a classe `str`, encontramos:

```
>>> dir(str)
['_add_', '__class__', '__contains__', '__delattr__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__',
 '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'formatter_field_name_split', 'formatter_parser',
 'capitalize', 'center', 'count', 'encode', 'endswith',
 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 'translate', 'upper', 'zfill']
>>>
```

Se desejarmos saber mais sobre o método 'upper', por exemplo, devemos usar o comando `help()`, colocando como parâmetro 'str.upper' (ou seja, no formato `classe.método`):

```
>>> help(str.upper)
Help on method_descriptor:
upper(...)
    S.upper() -> string
    Return a copy of the string S converted to uppercase.
>>>
```

Se houver alguma dificuldade em traduzir do Inglês, pode ser bastante útil usar um dos serviços gratuitos de tradução disponíveis na Internet. Embora a tradução por esse meio não seja contextual, é suficiente para a compreensão.

No caso da variável `meuLugar`, imprimindo o resultado, encontramos:

```
>>> print (meuLugar.upper())
BRASIL
>>>
```

Convém observar na lista de características da classe `str` a existência de alguns métodos com nomes especiais, iniciados e terminados por `"__"` (dois sublinhados). Trata-se dos chamados *métodos mágicos*. Estes são métodos preparados para serem invocados em formato de operadores sobre o objeto e não necessariamente citando o nome do método. Aqui se explica o fato de podermos usar determinados métodos associados aos tipos de Python da mesma maneira que é feita em linguagens essencialmente estruturadas.

Para que esses conceitos fiquem bem claros, consideremos o método `__add__`, um desses métodos mágicos (o primeiro da lista impressa acima). Executando o comando `help()` com o parâmetro 'str.__add__', obtemos:

```
>>> help(str.__add__)
Help on wrapper_descriptor:
__add__(...)
    S.__add__(T) -> object
    Return the object representing the sum of S and T.
```

```
x.__add__(y) <==> x+y
>>>
```

Como está explícito, o método é ativado ao ser realizada a operação de concatenação (que é uma operação já nossa conhecida dentro da abordagem estruturada). Isto é, dada uma string x e outra y , a chamada do método através de

$$x.__add__(y)$$

é equivalente à concatenação de x com y ,

$$x + y.$$

Esta equivalência pode ser demonstrada da seguinte maneira. Usando a operação de concatenação assim como a conhecemos, podemos montar, por exemplo, a variável `frase` concatenando string `meuLugar` com a string `', terra de contrastes!'`.

```
>>> frase = meuLugar + ', terra de contrastes!'
>>> print (frase)
Brasil, terra de contrastes!
>>>
```

Esta operação na verdade é respaldada pelo método `__add__` como é conferido abaixo:

```
>>> frase = meuLugar.__add__(', terra de contrastes!')
>>> print (frase)
Brasil, terra de contrastes!
>>>
```

É importante notar a importância do conhecimento sobre os métodos que pode ser útil em muitas situações.

A classe `list`. Os vetores em Python que estudamos neste curso são, na verdade, objetos da classe `list`. As características dessa classe podem ser exploradas via comandos `dir()` e `help()` do mesmo modo como fizemos com a classe `str`. Quando estudamos vetores, uma das maneiras que utilizamos para criá-los foi inicializando com tamanho previamente conhecido. Por exemplo, a inicialização com zeros de um vetor x que irá conter t números reais pode ser feita da seguinte maneira:

$$x = [0.0] * t$$

Esta expressão é uma forma compacta da concatenação de t listas do tipo `[0.0]`. Ou seja, ela produz o mesmo resultado da expressão seguinte:

$$x = [0.0] + [0.0] + \dots + [0.0] \text{ (com } t \text{ parcelas } [0.0] \text{).}$$

Este “crescimento” do vetor ocorre em tempo de execução, pois as listas de Python são *dinâmicas*. Este termo é usado para indicar que o gerenciamento de espaço para a lista na memória é feito em tempo de execução do programa. No experimento a seguir, o vetor v nasce vazio e seus elementos são acrescentados um a um.

Experimento 02

Tomemos um vetor `v` inicialmente sem nenhum elemento. Se resolvermos preenchê-lo com cinco números reais, por exemplo, podemos acrescentá-los lendo cada um deles do teclado e aplicando o recurso de concatenação de listas. Criar o diretório `L122_02` para abrigar os arquivos desse experimento, salvando lá o arquivo `testeList_a.py` com o seguinte código:

```
v = []#É criada a referência, mas não há conteúdo definido
for i in range(5):
    x = float(input('Digite um número: '))
    v += [x]
print('Vetor v =',v)
```

Executando esse programa e digitando os números 5.5, 7.0, 125.3, 8.25, 26.0, obtemos:

```
>>>
Digite um número: 5.5
Digite um número: 7
Digite um número: 125.3
Digite um número: 8.25
Digite um número: 26
Vetor v = [5.5, 7.0, 125.3, 8.25, 26.0]
>>>
```

Esta mesma operação de preenchimento do vetor `v` pode ser realizada usando o método `append()` da classe `list` (que acrescenta elementos ao final da lista). Vejamos então a versão `testeList_b.py` com o seguinte código:

```
v = []#É criada a referência, mas não há conteúdo definido
for i in range(5):
    x = float(input('Digite um número: '))
    v.append(x)
print('Vetor v =',v)
```

Executando o programa `testeList_b.py`, e digitando os mesmos números acima, a saída será absolutamente idêntica à de `testeList_a.py`:

```
>>>
Digite um número: 5.5
Digite um número: 7
Digite um número: 125.3
Digite um número: 8.25
Digite um número: 26
Vetor v = [5.5, 7.0, 125.3, 8.25, 26.0]
>>>
```

Uma operação bastante útil na solução de diversos problemas é a ordenação. Com este objetivo, a classe `list` possui o método `sort()`. No caso do vetor `v`, a chamada será `v.sort()` que colocará seus elementos em ordem crescente. A fim de facilitar os testes,

fixemos o vetor `v` como `[5.5, 7.0, 125.3, 8.25, 26.0]`. Executando a operação de ordenação, obtemos:

```
>>> v = [5.5, 7.0, 125.3, 8.25, 26.0]
>>> v.sort()
>>> v
[5.5, 7.0, 8.25, 26.0, 125.3]
```

Dentre outros métodos da classe `list` (e, logo, aplicáveis a `v`), vejamos mais os seguintes: `extend()`, `reverse()`, `count()`, `remove()`.

O método `extend()` - Estende uma lista usando os elementos de uma segunda lista dada. Por exemplo, desejando acrescentar os elementos da lista `w = [7.0, 28.0]` ao final de `v` em sua última versão acima, podemos fazer:

```
>>> w = [7.0, 28.0]
>>> v.extend(w)
>>> v
[5.5, 7.0, 8.25, 26.0, 125.3, 7.0, 28.0]
```

Observação: Há uma importante diferença para o método `append()`, que acrescenta um elemento de cada vez. Ou seja, conseguiríamos igual efeito de `extend(w)` sobre `v` somente com duas chamadas do método `append()`, uma para cada elemento de `w`.

O método `reverse()` - Inverte a ordem de uma lista. Aproveitando a sequência de operações sobre o vetor `v`, para inverter a ordem dos elementos é bastante fazer:

```
>>> v.reverse()
>>> v
[28.0, 7.0, 125.3, 26.0, 8.25, 7.0, 5.5]
```

Quanto aos métodos `remove()` e `count()`, dado um certo valor e uma lista, `remove()` remove a primeira ocorrência do valor na lista e `count()` retorna a quantidade de vezes em que o valor dado ocorre na lista. Por exemplo, seguindo a mesma seção de comandos acima, podemos contar quantas vezes o valor `7.0` aparece em `v` assim:

```
>>> v.count(7.0)
2
>>>
```

Ou seja, o valor `7.0` aparece duas vezes. Aplicando o método `remove` para este mesmo valor, apenas será removida a primeira ocorrência da esquerda para a direita:

```
>>> v.remove(7.0)
>>> v
[28.0, 125.3, 26.0, 8.25, 7.0, 5.5]
>>>
```

Observação: Esse método retorna um erro se o valor a ser removido não for encontrado. Deve ser usado com proteção suficiente para o caso da inexistência do tal valor.

A classe tuple. Uma tupla é como uma lista constante (conferir suas características usando o comando `dir(tuple)`). Tuplas são recomendadas quando se trabalha com um conjunto fixo de valores. Já utilizamos tuplas, por exemplo, quando fizemos atribuições múltiplas como a seguinte:

```
>>> Nome, ok, a, b = 'Maria', True, 0, 0
```

Temos à esquerda uma tupla de variáveis e à direita uma tupla com os respectivos valores.

Experimento 03

Executando o comando acima e fazendo a impressão em seguida, obtemos:

```
>>> Nome, ok, a, b = 'Maria', True, 0, 0
>>> print (Nome, ok, a, b )
Maria True 0 0
>>>
```

Tuplas podem receber identificadores e normalmente são representadas entre parênteses. A título de demonstração podemos então fazer:

```
>>> valores = ('Maria', True, 0, 0)
>>> (Nome, ok, a, b) = valores
>>> print (valores)
('Maria', True, 0, 0)
>>> print (Nome, ok, a, b)
Maria True 0 0
>>>
```

Podemos usar tuplas também como retorno de funções. Por exemplo, se desejarmos trocar o conteúdo de duas variáveis, `a` e `b`, podemos usar a função `troca(x, y)`:

```
>>> def troca(x, y):
    return y, x

>>> a, b = 125, 67.9                                     #valores arbitrários
>>> print (a, b)
125 67.9
>>> a, b = troca(a, b)
>>> print (a, b)
67.9 125
>>>
```

A classe dict. A linguagem Python possui uma estrutura de alto nível chamada de *dicionário* (`dict`), que implementa um conjunto de associações entre pares de valores. O primeiro valor, chamado de *chave*, serve para localizar/identificar o segundo valor, denominado de *conteúdo*. Um objeto `D` da classe `dict` deverá ser representado da seguinte maneira:

```
D = {chave: conteúdo, chave: conteúdo, ...}
```

onde, os conteúdos podem ser de qualquer tipo, mas as chaves somente podem ser de tipo imutável. Para conhecer mais sobre a classe `dict` podemos aplicar os comandos `dir()` e `help()`, da maneira como já utilizamos anteriormente.

As aplicações de dicionários são as mais diversas, mas, sempre ligadas aos casos em que os dados precisam ser localizados através de valores índices.

Experimento 04

Veamos como, usando dicionário, podemos representar uma lista de telefones que estariam na agenda de um aparelho celular. Faremos como de fato acontece na maioria desses aparelhos, onde a chave de busca usada é, normalmente, o nome do contato (um nome abreviado). Os conteúdos serão representados por listas com os números de telefone de cada contato (serão usadas listas porque cada contato pode ter mais de um telefone).

A título de exemplo, tomemos o quadro abaixo com dados arbitrários, de nomes de contatos e seus números de telefones:

Contato	Telefones
'Antonio'	'93067177'
'Eliane'	'99050614', '87142525', '33263887'
'Jonas'	'93067178', '91057091'

Criamos o dicionário `contatos` fazendo:

```
>>> contatos = {
    'Antonio': ['93067177'],
    'Eliane': ['99050614', '87050614', '33263887'],
    'Jonas': ['93067178', '91057091']}
>>>
```

Observação: Um dicionário também pode ser criado a partir de uma lista com os pares (*chave*, *conteúdo*), aplicando o comando `dict()` sobre a mesma (consultar bibliografia do curso).

Conferindo a atribuição acima:

```
>>> print (contatos)
{'Jonas': ['93067178', '91057091'], 'Eliane': ['99050614',
'87050614', '33263887'], 'Antonio': ['93067177']}
>>>
```

Para acrescentar um novo contato é bastante fazer a atribuição usando a chave como índice. Por exemplo, para inserir o contato ('Carolina', ['99029355']):

```
>>> contatos['Carolina'] = ['99029355']
>>> print (contatos)
{'Jonas': ['93067178', '91057091'], 'Carolina': ['99029355'],
'Eliane': ['99050614', '87050614', '33263887'], 'Antonio':
['93067177']}
>>>
```

Consideremos que Carolina tem mais um número (digamos, '88029344') e precisamos acrescentá-lo à sua lista. Sabendo-se que `contatos['Carolina']` é uma lista (por enquanto com apenas um número de telefone), aplicaremos o método `append()`:

```
>>> contatos['Carolina'].append('88029344')
>>> print (contatos)
{'Jonas': ['93067178', '91057091'], 'Carolina': ['99029355',
'88029344'], 'Eliane': ['99050614', '87050614', '33263887'],
'Antonio': ['93067177']}
```

Exercício de autoavaliação

Resolva as questões abaixo e discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Escreva um programa para criar a lista `L` dos números inteiros de 1 a 10 usando comandos da linguagem (e não digitando esses números diretamente) e executar as operações abaixo mostrando logo em seguida o estado da lista (Pesquise um pouco mais sobre os métodos da classe `list` consultando a bibliografia do curso):

- Inserir o número 99 na posição de índice 4 sem remover o que lá está;
- Remover o elemento está na posição de índice 5;
- Remover o elemento de valor igual a 9;
- Inserir o valor -5 na primeira posição da lista;
- Escrever o índice da posição do valor 99 (use um método para extrair esse índice);
- Construir a sub-lista `S` com 4(quatro) elementos de `L` iniciando no segundo elemento;
- Remover os elementos de `L` que sejam de `S`;
- Concatenar `S` ao final da lista `L`.

2 - Quando estudamos conjuntos em Algoritmo e Estrut. de Dados I, descobrimos uma maneira prática de tomar apenas os elementos distintos de uma lista. Podemos montar uma lista `W` com os elementos distintos de uma lista `L` da seguinte maneira:

```
W = list(set(L))
```

Ou seja, a conversão em conjunto `set(L)` escolhe apenas os elementos distintos de `L`. Depois, é só converter novamente em lista, `list(set(L))`. Faça um exemplo mostrando que esse comando realmente funciona.

3 - Ao executar o comando do item anterior, notamos que a sequência original dos elementos de `L` pode não ser mantida. Elabore e experimente a função `EliminaRep(lst)` que elimina repetições mantendo as primeiras ocorrências dos valores da lista `lst` passada como parâmetro, retornando uma lista com apenas os elementos distintos. *Sugestão: Manter a lista original (de entrada) e, seguidamente, copiar para uma temporária (de saída) apenas as primeiras ocorrências (ou seja, copiar verificando antes se o elemento já está na lista de saída).*

4 - Elabore um programa para consultar o dicionário de contatos definido no **Experimento 04** do laboratório da **Subunidade 1.2.2** (pesquise mais sobre os métodos da classe *dict*). O programa recebe repetidamente nomes de pessoas e mostra seus números de telefones. Se o nome não pertencer à lista das chaves do dicionário (ver o método `keys()`) o programa apenas emitirá uma mensagem (“Contato inexistente”).

5 - A linguagem Python permite a definição iterativa de listas, algo semelhante ao que é feito em Matemática. Por exemplo, podemos escrever o conjunto **I** dos números ímpares das duas maneiras seguintes. Construímos **I** listando diretamente seus elementos,

$$I = \{1, 3, 5, 7, \dots\},$$

(sendo um conjunto infinito, usamos as reticências) ou, usando uma descrição (uma regra),

$$I = \{x \mid x = 2i+1, i \in \mathbf{N}\}$$

(Lê-se, conjunto dos números x , tal que $x = 2i+1$, sendo i um número natural).

Em computação, como não trabalhamos com infinito, é claro, vamos escolher um limite. Tomemos então os cinco primeiros números ímpares, por exemplo. Podemos construir em Python a lista **Imp** com tais elementos, da seguinte maneira:

```
>>> Imp = [2*i+1 for i in range(5)]
>>> Imp
[1, 3, 5, 7, 9]
>>>
```

(Isto é, **Imp** é uma lista com os elementos obtidos por $2*i+1$, sendo i na faixa de 0 a 4).

Na construção de uma lista, também podemos fazer seleções. Por exemplo, podemos conseguir a mesma lista **Imp** usando um mecanismo com seleção. Percorremos os 10 primeiros números naturais e tomamos apenas os números ímpares. Ou seja:

```
>>> Imp = [i for i in range(10) if i%2!=0]
>>> Imp
[1, 3, 5, 7, 9]
>>>
```

(Isto é, **Imp** é a lista formada pelos elementos i na faixa de 0 a 10 e se i **não** for divisível por 2).

Como faríamos então se desejássemos construir uma lista iterativamente com elementos lidos do teclado? Tomemos o seguinte exemplo: Construir a lista **Nomes** com quatro nomes de pessoas lidos do teclado. Fazemos:

```
>>> Nomes = [input('> ') for contador in range(4)]
> João
> Maria
> Joaquim
> José
>>> Nomes
['João', 'Maria', 'Joaquim', 'José']
>>>
```

(Isto é, a lista **Nomes** é a lista formada pelos elementos obtidos por *input()* chamada para cada valor de *contador* – ou seja, quatro vezes).

A seguir, A lista **L** é formada pelos elementos de **Nomes** iniciados com 'J':

```
>>> L = [n for n in Nomes if n[0].upper()=='J']
>>> L
['João', 'Joaquim', 'José']
>>>
```

(Isto é, a lista **L** é formada por elementos **n** de **Nomes** tais que a primeira letra de **n** (convertida para maiúscula) seja 'J').

a) Teste os exemplos acima;

b) Elabore um programa que lê dois inteiros, **m** e **n**, e, usando construção iterativa de listas, constrói e escreve uma lista com todos os números pares maiores ou iguais a **m** e menores que **n**.

6 - Uma palavra é um palíndromo se a sua leitura for idêntica tanto da esquerda para a direita quanto da direita para a esquerda. Por exemplo, “anilina”, “arara” e “assa”, são palíndromos. Elabore e faça um teste da função chamada *ehPalindromo(w)*, que recebe uma palavra *w* como parâmetro e retorna *True* se esta for um palíndromo e *False* no caso contrário.

7 - Para eliminar a primeira ocorrência de um valor numa lista (da classe *list*) temos o método *remove()*, como foi visto no texto. Deseja-se desta vez que seja removida a última ocorrência. Com esta finalidade, elabore a função *removeUltimo(val, lst)* que recebe a lista original *lst* e o valor *val* a ser removido, e retorna *lst* já alterada. Para testar a função, escreva um pequeno programa que lê do teclado uma lista **L** de números reais (usando o critério que desejar) e um número real **x**, e, em seguida, escreve **L** tendo eliminado a última ocorrência de **x**. Se **x** não existir em **L**, o programa emite uma mensagem de erro.

8 - Elabore um programa em Python que lê **n** números reais armazenando-os na lista **v** e, em seguida, usando construção iterativa de listas, calcula e escreve o *desvio médio (DM)* dos elementos de **v**. Para calcular o desvio médio, primeiramente, determina-se a soma **S** dos valores absolutos (ou módulos) das diferenças entre cada elemento e a média aritmética **m**. Se $v = [x_1, x_2, \dots, x_n]$,

$$S = \text{abs}(x_1 - m) + \text{abs}(x_2 - m) + \dots + \text{abs}(x_n - m),$$

que é a soma dos elementos da lista $[\text{abs}(x_1 - m), \text{abs}(x_2 - m), \dots, \text{abs}(x_n - m)]$, onde *abs()* é o valor absoluto (Em Python, veja a função *fabs()*, da biblioteca *math*).

O desvio médio é calculado dividindo-se esta soma pela quantidade de elementos **n**. Ou seja,

$$DM = S/n$$

Módulo II

Estruturas de dados lineares

Já conhecemos tipos de dados estruturados homogêneos, como vetores e matrizes, e heterogêneos, como os registros e até vetores de registros. Então, avançaremos com tipos abstratos de dados que tomam essas estruturas como base.

Podemos dividir as estruturas de dados clássicas em estruturas de dados *lineares* e *não-lineares*. As lineares são *listas*, *pilhas* e *filas*, onde os dados estão montados sequencialmente (algo como os bem conhecidos vetores) e as não-lineares são *árvores* e *grafos*, estruturas em que cada dado pode ter mais de um sucessor (algo como o organograma de uma empresa, uma árvore genealógica, etc.).

Este módulo está dedicado às estruturas lineares. Fazendo uma comparação breve entre as estruturas desse tipo, dizemos que as listas são tais que o acesso aos dados (para leitura, inserção, etc) pode ser feito a partir de qualquer um de seus pontos. Uma pilha é uma lista especial em que inserções ou retiradas são permitidas somente no *topo* (equivalente ao fim da lista) e uma fila admite dois pontos de acesso: um no fim exclusivamente para inserções e outro no início para retiradas.

Objetivos

- Identificar problemas cujas soluções podem ser modeladas usando listas, pilhas ou filas;
- Implementar lista, pilha e fila aplicando o conceito de TAD e usando estruturas predefinidas na linguagem Python.

Unidades

- Unidade II.1 - Listas
- Unidade II.2 - Pilhas
- Unidade II.3 – Filas

Unidade II.1

Listas

Com as *listas*, iniciamos o estudo das *estruturas de dados lineares*. Como já foi dito, nas estruturas ditas lineares, os dados são montados mantendo-se um único sucessor para cada um deles. As listas são bem conhecidas nossas. Dentre vários exemplos que podemos citar (uma lista de compras, uma relação de alunos de uma turma etc.), tomemos o **Exemplo 1.1** (inserido na introdução ao **Módulo I**). Vimos, naquele caso, que a solução do problema dependia da organização dos objetos de interesse. As contas de luz apresentavam-se antes espalhadas e misturadas com outros documentos numa gaveta.

No citado exemplo das contas de luz, para uma eficiente busca de uma das contas, o acesso a cada uma delas precisou ser "disciplinado". O simples fato de sequenciar, colocando uma conta sobre a outra, já se produz o que podemos chamar de *lista*. Depois da organização, cada conta passou a ter um único sucessor (com exceção da última, é claro), resultando em maior facilidade para sua manipulação em operações, como: retirar uma conta da lista, inserir, buscar uma conta na sequência em que estão colocadas etc. Percebemos também que a colocação, segundo uma ordem, é mais um ponto a favor da eficiência dessas operações.

2.1.1 Conceituação

Numa lista, é possível retirar e inserir novos componentes *em qualquer um dos seus pontos*. Portanto, o conjunto das contas de luz do **Exemplo 1.1**, após sua organização, é um caso típico. Consideremos que as contas são colocadas em ordem por mês de faturamento (por ano e, em seguida, por mês em cada ano). Observamos, no processo de ordenação, que uma conta pode ser retirada e inserida em qualquer ponto da lista, de modo que seja mantida a ordem da conta mais antiga para a mais recente.

Lista sequencial e lista encadeada

Para providenciar os algoritmos de manipulação de uma lista, é necessário decidir sobre as condições da futura implementação, pois interferem na maneira de execução das operações básicas (por exemplo, inserção ou remoção de elementos). Uma das condições é relativa à disposição física na memória do computador. Sob esse critério, uma lista pode ser classificada como *sequencial* ou *encadeada*.

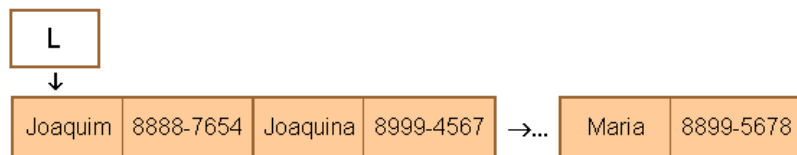
Lista sequencial. Em uma lista sequencial, os dados estão dispostos como os elementos de um vetor (ou seja, em posições vizinhas na memória), como no exemplo seguinte.

Exemplo 2.1

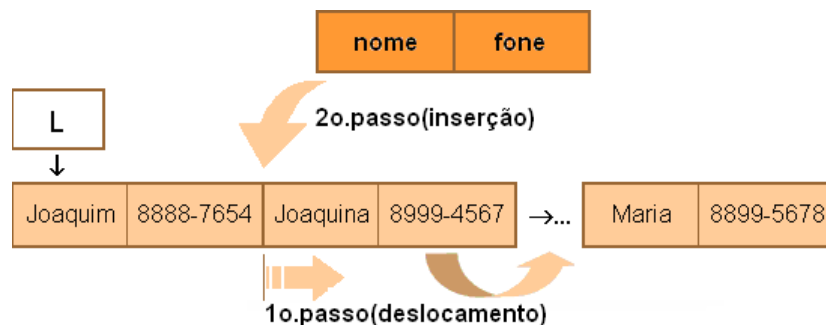
Considere-se uma lista de nomes e respectivos telefones de clientes em uma agenda telefônica de um pequeno empresário (este caso foi abordado no exercício de autoavaliação da **Subunidade 1.1.1 – Módulo I**). O registro de cada cliente terá o formato:

nome	fone
------	------

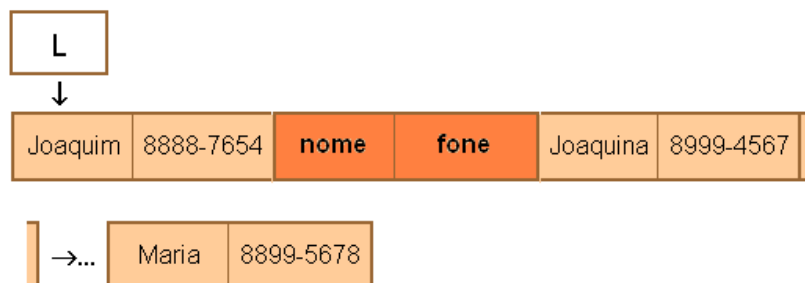
Montados segundo uma lista sequencial, eles estarão dispostos em posições contíguas da memória do computador, conforme está representada a lista **L** abaixo:



Sob a disposição sequencial, vizinhos lógicos na lista são também vizinhos físicos na memória do computador. Por essa razão, operações de inserção de novos elementos ao longo da lista, ou remoção de algum deles, requererem o deslocamento de vários elementos fisicamente vizinhos. Desejando-se, por exemplo, inserir um novo cliente na segunda posição da lista, todos os elementos a partir dessa posição devem ser deslocados para a direita como na figura abaixo.



E o resultado será:



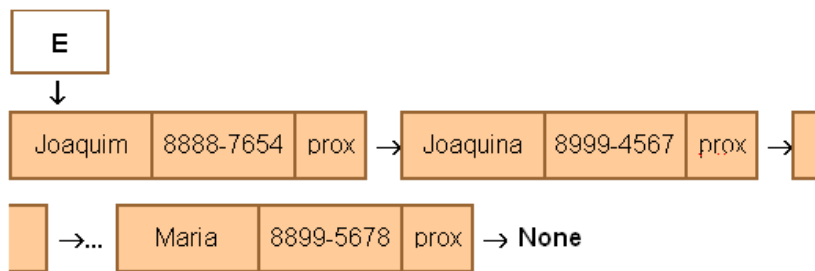
Lista encadeada. Em uma lista encadeada, os dados podem não estar em posições contíguas da memória. Os blocos de dados são dispostos de modo que cada bloco contém a indicação do seguinte, como no exemplo abaixo.

Exemplo 2.2

Tomemos a lista de clientes do exemplo anterior. Para se construir uma lista encadeada com estes dados, o registro de um cliente deverá conter mais um campo, que servirá de "ponteiro" para o próximo cliente, ou seja,

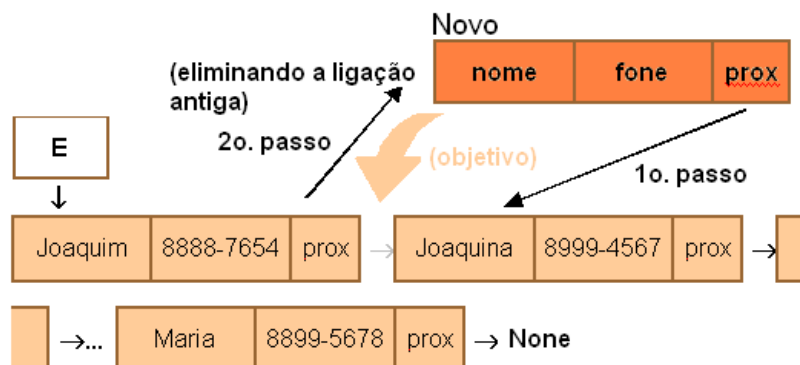


O campo `prox` contém uma referência ao próximo cliente da lista. Construímos a lista `E`, encadeando elementos deste tipo (cada um indicando qual é o cliente seguinte):



`E` guarda uma referência ao primeiro cliente e este é que tem (no seu campo `prox`) o endereço do segundo, o segundo cliente apontará para o terceiro e assim por diante. O último cliente tem o seu campo `prox` igual a **None** (usando a notação da linguagem Python para referências nulas) para indicar que não aponta para mais nenhum cliente.

Sob essa disposição, vizinhos lógicos (que obedecem à sequência das ligações) não necessariamente são vizinhos físicos, pois a restrição de contiguidade na memória do computador não existe. Os blocos só precisam ser localizados pelo seu antecessor e apontar para seu sucessor. Por essa razão, executar inserção ou remoção de elementos ao longo da lista não exige o deslocamento de blocos de dados. A manipulação das ligações é o bastante. Como exemplo, a figura abaixo mostra uma sequência de operações para inserir um novo cliente (chamado `Novo`) na segunda posição da lista.



Lista estática e lista dinâmica

Considerando o modo de alocação de memória, as listas são classificadas como *estáticas* e *dinâmicas*.

Lista estática. Quando uma lista recebe uma implementação estática, a quantidade total de memória abrangida pelos dados é conhecida previamente e não muda durante toda a execução do programa. Os vetores (não incluindo a implementação em Python que estamos adotando, ou quando, propositalmente, não se deseja variar o tamanho do vetor durante a execução do programa) podem ser considerados listas estáticas.

Lista dinâmica. Numa implementação dinâmica, a quantidade de memória utilizada pelos dados do programa durante a execução é variável. Por exemplo, ao utilizarmos listas da linguagem Python como base, ganhamos o direito de incrementar elementos além daqueles definidos no início do programa. Essa modificação de tamanho acontece durante a execução do programa. Isso caracteriza uma lista dinâmica. Por exemplo, no **Experimento 02**, da **Subunidade 1.2.2**, a lista (chamada naquele exemplo de “vetor”) iniciada com $v = []$ ganha novos elementos digitados pelo usuário quando o programa está em execução.

Observação: Este curso, particularmente, será concentrado nas *listas sequenciais dinâmicas*. O objetivo é facilitar a aplicação dos conceitos aproveitando recursos disponíveis na linguagem de programação adotada.

Exercício de autoavaliação

Realize os exercícios abaixo e discuta no fórum dos conteúdos da semana. Tire suas dúvidas e, oportunamente, auxilie também.

1 - Consta no **Exemplo 2.1** uma descrição gráfica da operação de inserção de um elemento numa lista sequencial, L . O tipo registro `Cliente` representa um elemento da lista:

```
Tipo_registro Cliente:
    Nome (caracteres)
    Fone (caracteres)
Fim_registro
```

A lista L pode ser representada por t elementos `Cliente` indexados (um vetor de registros `Cliente` – ver **Subunidade 1.1.1**) da seguinte maneira:

```
L[0], L[1], L[2], ..., L[t-1]
```

Use uma linguagem simples para descrever os passos para se fazer a inserção de um novo elemento no início da lista, orientando-se pelo que está descrito na figura do **Exemplo 2.1**.

2 - O **Exemplo 2.2** mostra uma representação gráfica de uma lista encadeada, E , e de uma operação de inserção nesta. Represente um elemento da lista por

```
Tipo_registro Cliente:
    Nome (caracteres)
    Fone (caracteres)
    prox (do tipo Cliente)
Fim_registro
```

A lista `E` é representada pelo seu primeiro elemento. Não há índices, pois o campo `prox` de `E` indicará o segundo elemento, o `prox` do segundo informará qual é o terceiro, e assim por diante. Logo, só será possível localizar elementos e navegar na lista se o acesso for feito a partir da sua “cabeça” que é o elemento indicado por `E`.

Tomando a figura do **Exemplo 2.2** como base e chamando o novo elemento de `Novo`, escreva os passos para inserir esse elemento no início da lista.

2.1.20 TAD lista

Conforme sabemos, para se definir um tipo abstrato de dado, devem-se determinar os atributos (quais são os dados propriamente) e o conjunto das operações sobre esses atributos (a interface com suas funções).

Os atributos de uma lista são, basicamente, seus elementos e seu tamanho.

A interface corresponde ao conjunto das operações sobre uma lista. Classicamente, as operações são as seguintes:

Criação - Criar uma lista na memória do computador inicialmente vazia (pode preencher em seguida com dados lidos do teclado ou de um arquivo em disco).

Pesquisa - Dado um elemento (do mesmo tipo da lista), determinar se este pertence à lista, ou qual elemento está armazenado em uma dada posição na lista.

Inserção - Criar um espaço na lista (numa lista sequencial) e inserir um dado elemento nesse espaço.

Remoção - Remover um elemento de uma dada posição da lista. Desejando-se remover algum elemento pelo seu valor de atributo, executa-se primeiramente uma pesquisa para determinar em que posição está o valor.

Alteração - Uma vez localizado o elemento na lista, modificar valores de seus atributos.

Outras operações possíveis são: ordenação, determinação da quantidade de elementos, concatenação de listas, partição ou determinação de sublistas etc. Podemos observar que a linguagem Python implementa listas com um semelhante conjunto de operações.

Exemplo 2.3

Consideremos que se deseja montar uma “caderneta escolar eletrônica” para registrar as notas dos alunos de uma turma numa certa disciplina. Cada aluno tem nome, matrícula, notas (AB1 e AB2) e quantidade de faltas, e estes dados irão conferir-lhe um resultado (aprovado, reprovado ou em reavaliação). Sabe-se que a carga horária da disciplina é 120 horas, e o aluno que faltar mais que 25% desta estará automaticamente reprovado (e o resultado será “RF” - reprovado por falta). Observando a média aritmética das notas, o resultado será determinado da seguinte maneira:

Média das notas, M	Resultado
$M \geq 7,0$	"AP" - Aprovado
$5,0 \leq M < 7,0$	"RA" - Em reavaliação
$M < 5,0$	"RM" - Reprovado por média

A lista de alunos será implementada, usando uma lista sequencial dinâmica. Um programa deverá ser projetado para gerenciar a caderneta, inserindo ou removendo elementos da lista de alunos, e consultando dados existentes. Definiremos o tipo abstrato `Caderneta` e este faz uso de outro tipo abstrato chamado `Aluno`.

O TAD `Aluno`

Os atributos serão definidos pelo tipo registro `Aluno`.

Tipo_registro `Aluno`:

nome (caracteres)
matric (caracteres)
AB1 (numérico)
AB2 (numérico)
n_faltas (numérico)

Fim_registro

A interface será composta das três funções a seguir.

Função `NovoAluno()` - Cria e retorna um registro do tipo `Aluno` com seus campos preenchidos via teclado:

```
Defina NovoAluno():
    al ← Aluno()
    Escrever "Nome: "
    ler al.nome
    Escrever "Matrícula: "
    ler al.matric
    Escrever "AB1: "
    ler al.AB1
    Escrever "AB2: "
    ler al.AB2
    Escrever "Quant.faltas: "
    ler al.n_faltas
    retornar al
Fim_função
```

Função `Resultado(al)` - Recebe um `Aluno` como parâmetro, determina e retorna seu resultado ("AP", "RA", "RF" ou "RM"). Uma variável global, chamada `CH`, será declarada para conter a carga horária da disciplina:

```
Defina Resultado(al):
    se al.n_faltas/CH > 0.25 então: retornar "RF"
    media ← (al.AB1 + al.AB2)/2.0
    se media >= 7.0 então: retornar "AP"
    se media >= 5.0 então: retornar "RA"
    retornar "RM"
Fim_função
```

Função EscreveAluno(al) - Mostra os dados de um dado Aluno:

```
Defina EscreveAluno(al):  
  Escrever "Nome: ", al.nome  
  Escrever "Matrícula: ", al.matric  
  Escrever "AB1: ", al.AB1, "AB2: ", al.AB2  
  Escrever "Num. faltas: ", al.n_faltas  
  Escrever "Resultado: ", Resultado(al)  
Fim_função
```

O TAD Caderneta

Os atributos de Caderneta serão os de uma lista de dados do tipo Aluno:

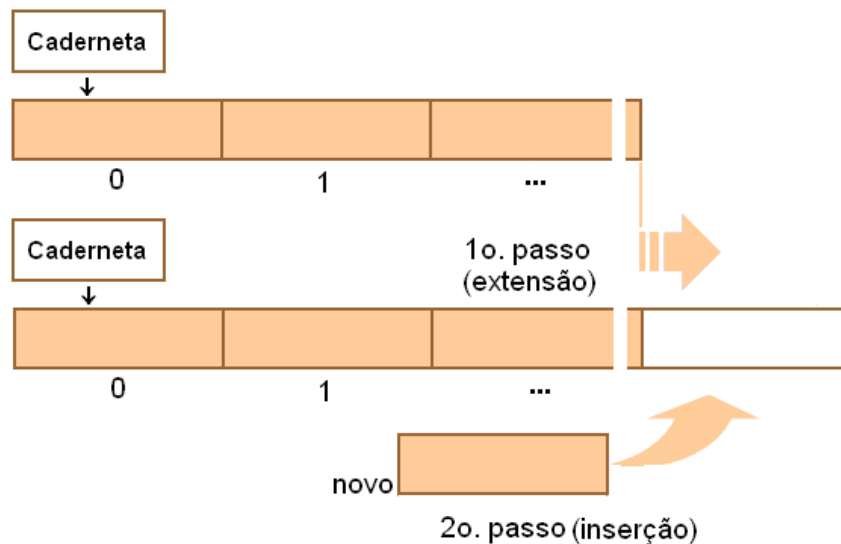
Caderneta = [] (uma lista inicialmente vazia)

Quanto à interface, vão ser consideradas as operações para inserção remoção e consulta à lista.

Função insereNoFim(al, ct) - Insere um dado Aluno al no fim da lista localmente denominada de ct:

```
Defina insereNoFim(al, ct):  
  {alocar memória após o último elemento de ct}  
  {atribuir o elemento al a este local}  
Fim_função
```

Os passos do algoritmo insereNoFim() estão descritos na figura abaixo



Observação: No momento da implementação, os dois comandos da função acima poderão ser substituídos até por uma única chamada de uma função pré-definida.

Função posicaoAluno(matricula, ct) – Retorna a posição na lista (localmente denominada de ct) a partir de uma informação associada a um aluno. Nesse exemplo, a informação utilizada é sobre a matrícula do aluno, porém, variações podem ser feitas,

permitindo uma localização baseada em qualquer dos campos de um registro `Aluno` (Utilizamos aqui uma lógica direcionada para o aproveitamento das características da linguagem Python):

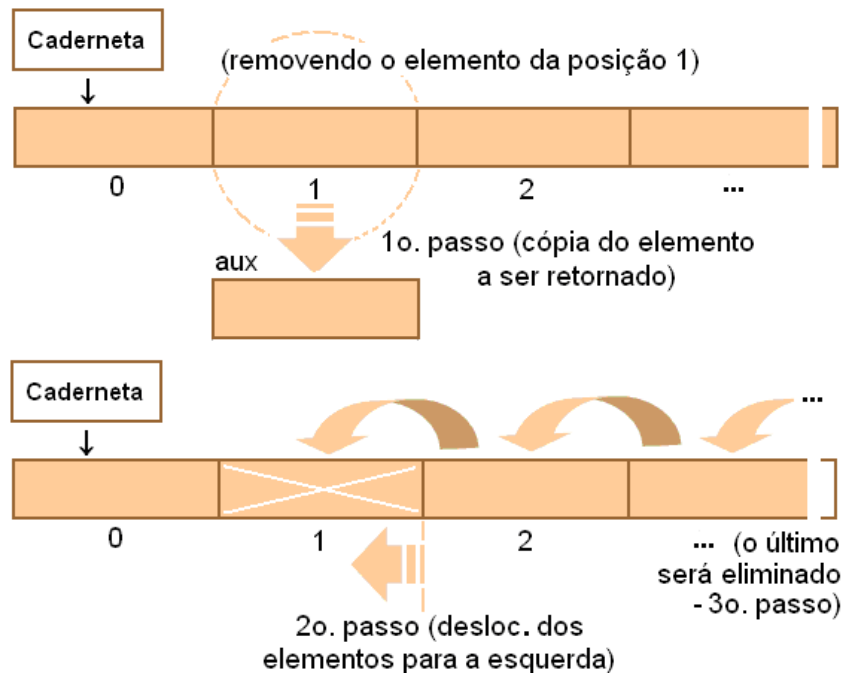
```
Defina posiçãoAluno(matricula, ct):  
    {obter uma lista com todas as matrículas dos alunos na mesma sequência de ct}  
    se (matricula não estiver na lista de matrículas) então: retornar None  
    retornar índice de matricula na lista de matrículas  
Fim_função
```

O primeiro passo do algoritmo é construir uma lista com todas as matrículas, mantendo a ordem da lista original. Se a matrícula dada estiver na lista das matrículas, é bastante descobrir qual é o seu índice, e este valor será retornado pela função.

Função `removeAluno(pos, ct)` - Remove da lista `ct` e retorna o elemento que foi removido da posição válida `pos`:

```
Defina removeAluno(pos, ct):  
    {salvar em aux o elemento da posição pos de ct}  
    {deslocar os elementos de ct de um item para a esquerda a partir de pos+1}  
    {eliminar o último elemento de ct}  
    retornar aux  
Fim_função
```

A figura seguinte descreve a lógica dos comandos da função `removeAluno()`:



O último passo do algoritmo (apenas este não está visualizado na figura) corresponde ao retorno do elemento gravado na variável auxiliar `aux`.

Função `acessaAluno(pos, ct)` - Retorna (sem remover) um elemento da lista `ct` que está na posição válida `pos` (Observemos que esta função tem como base uma operação simples e bem conhecida, que é o acesso a elementos de vetores):

```
Defina acessaAluno(pos, ct):  
    retornar ct[pos]  
Fim_função
```

A solução do problema proposto nesse exemplo pode então ser dada pelo algoritmo abaixo:

Algoritmo

```
{Declarar o tipo abstrato Aluno}  
{Declarar o tipo abstrato Caderneta}  
1 Cdt = Caderneta  
2 faça:  
2.1 Escrever " (1)Inserir aluno"  
2.2 Escrever " (2)Remover aluno"  
2.3 Escrever " (3)Consultar lista"  
2.4 Escrever " (4)Encerrar"  
2.5 Escrever "Digite sua opção > "  
2.6 Ler opc  
2.7 Se (opc=="1") então:  
    Escrever "Digite os dados de um aluno: "  
    insereNoFim(NovoAluno(), Cdt)  
Senão se (opc=="2") então:  
    Ler mat {lê a matrícula do aluno a ser removido}  
    indice ← posicaoAluno(mat, Cdt)  
    Se (indice!=None) então:  
        Escrever "Aluno removido: "  
        removido ← removeAluno(indice, Cdt)  
        EscreveAluno(removido)  
    senão: Escrever "--> Aluno inexistente! "  
Senão se (opc=="3") então:  
    Ler mat {lê a matrícula do aluno a ser exibido}  
    indice ← posicaoAluno(mat, Cdt)  
    Se (indice!=None) então:  
        Escrever "Dados do aluno: "  
        alunoAcessado ← acessaAluno(indice, Cdt)  
        EscreveAluno(alunoAcessado)  
    Senão: Escrever "--> Aluno inexistente! "  
Senão se (opc=="4") então: (vai para o fim do programa)  
Senão: Escrever "Opção inválida! "  
enquanto (opc!="4")
```

Fim-Algoritmo

O primeiro comando do algoritmo acima cria um nome compacto `Cdta` para a lista `Caderneta` apenas para tornar explícita a lista que está sendo manipulada. O comando 2.7 (dentro da estrutura `faça-enquanto`) é uma estrutura de seleção que direciona as opções do usuário (sem muito rigor na verificação de erros). A conferência dos passos deste algoritmo será feita no laboratório desta subunidade.

Laboratório - Listas

Objetivos

Conferir os algoritmos apresentados nesta unidade;

Identificar e aplicar os recursos disponíveis na linguagem Python para implementar o TAD Lista.

Recursos e implementação

Consideremos o problema do **Exemplo 2.3**: Desejando-se montar e gerenciar uma lista com dados de alunos, são construídos dois tipos abstratos: `Aluno` e `Caderneta`. A solução desse problema será explorada nos dois experimentos seguintes.

Experimento 01

Neste experimento, será construído e testado um módulo Python, que contém a definição do tipo abstrato `Aluno`. Criar um diretório no disco, atribuindo o nome `L212_01`, onde serão colocados todos arquivos envolvidos nesse experimento. Iniciar o IDLE e criar o arquivo `aluno.py` com as linhas de código a seguir:

```
# Tipo Abstrato 'Aluno'
CH = 120.0 #Carga horária da disciplina
#ATRIBUTOS:-----
class Aluno: #Implementação do 'tipo_registro Aluno'
    nome = ' '
    matric = ' '
    AB1 = 0.0
    AB2 = 0.0
    n_faltas = 0

#INTERFACE:-----
def NovoAluno(): #cria um aluno em particular
    al = Aluno()
    al.nome = input('Nome: ')
    al.matric = input('Matrícula: ')
    al.AB1 = float(input('AB1: '))
    al.AB2 = float(input('AB2: '))
    al.n_faltas = int(input('Quant.faltas: '))
    return al

def Resultado(al): #determina e retorna o resultado do aluno
    global CH
```

```

    if al.n_faltas/CH > 0.25: return 'RF'
    media = (al.AB1 + al.AB2)/2
    if media >= 7.0: return 'AP'
    if media >= 5.0: return 'RA'
    return 'RM'

def EscreveAluno(al):          #mostra um Aluno em particular
    print ('Nome:', al.nome)
    print ('Matrícula:',al.matric)
    print ('AB1: %.1f, AB2: %.1f' % (al.AB1, al.AB2))
    print ('Num. faltas:', al.n_faltas)
    print ('Resultado:', Resultado(al))
#-----

```

Observemos que os códigos das funções estão naturalmente explicados pelos algoritmos apresentados no **Exemplo 2.3**.

Antes de integrar o módulo `aluno.py` à versão final do programa, vamos testá-lo. Fazendo isso no *prompt* de comandos, usamos o comando seguinte para incluir todas as definições contidas no referido módulo:

```
>>> from aluno import *
```

Para testar a função `NovoAluno()`, criamos um `Aluno a1`, atribuindo os seguintes dados de um aluno fictício: "Manoel", "2006G0765", 6, 7.5, 20:

```
>>> a1 = NovoAluno()
Nome: Manoel
Matrícula: 2006G0765
AB1: 6
AB2: 7.5
Quant.faltas: 20
>>>
```

Podemos ainda conferir o preenchimento dos campos, imprimindo cada um deles (`al.nome`, `al.matric`, `al.AB1`, `al.AB2` e `al.n_faltas`).

Testando a função `Resultado()`:

```
>>> Resultado(a1)
'RA'
>>>
```

De fato, a média do aluno $((6.0+7.5)/2=6.75)$ está entre 5.0 e 7.0 indica que ele está em reavaliação, conforme a tabela.

Testando a função `EscreveAluno()` (que também irá conferir as primeiras funções):

```
>>> EscreveAluno(a1)
Nome: Manoel
Matrícula: 2006G0765
AB1: 6.0, AB2: 7.5
Num. faltas: 20
```

```
Resultado: RA
>>>
```

Experimento 02

Inicialmente, será criado e testado o tipo abstrato `Caderneta`. Em seguida, deveremos combinar o uso de `Aluno` e `Caderneta` para implementar a solução do problema proposto no **Exemplo 2.3**. Criar um diretório com o nome `L212_02` onde serão colocados os arquivos envolvidos neste experimento. Neste diretório, deverão estar o arquivo `aluno.py`, criado no **Experimento 01**, e o arquivo `caderneta.py`, cujo código está descrito abaixo:

```
# Tipo Abstrato 'Caderneta'
#ATRIBUTOS:-----
Caderneta = []
#INTERFACE:-----
def insereNoFim(al, ct = Caderneta):
    ct.append(al)

def posicaoAluno(matricula, ct = Caderneta):
    matriculas = [al.matric for al in ct]
    if matricula not in matriculas: return None
    return matriculas.index(matricula)

def removeAluno(pos, ct = Caderneta):
    return ct.pop(pos)

def acessaAluno(pos, ct = Caderneta):
    return ct[pos]
#-----
```

Os códigos das funções estão de acordo com os algoritmos do TAD `Caderneta` apresentadas no **Exemplo 2.3**.

Podemos observar que o uso da linguagem Python oferece vantagens quanto à quantidade e simplicidade de comandos. Outra observação importante é sobre o parâmetro de cada função, que referencia a lista de alunos `Caderneta`. `ct` é uma variável local, mas permite o acesso direto à lista de alunos. Sendo isso automático, esse parâmetro poderá ser omitido na chamada da função. Por outro lado, essa implementação obriga que a manipulação seja de apenas uma caderneta por vez. Este e outros detalhes de implementação são comentados abaixo:

Constam os três comandos abaixo na função `insereNoFim()`:

```
{alocar memória após o último elemento de ct} e
{atribuir o elemento a1 a este local}
{eliminar o último elemento de ct}
```

Estes comandos estão sendo implementados por uma única chamada do método `append()` da classe `list` de Python (pois `ct` é uma lista):

```
ct.append(al)
```

A função `posicaoAluno()` tem o objetivo de generalizar a localização de um aluno em função de um de seus campos do registro. Uma vez escolhida a chave para a localização (no caso, a busca é pela matrícula do aluno), o primeiro passo do algoritmo é construir uma lista com todos exemplares da chave considerada, mantendo a ordem da lista original. Chamando esta lista de `matriculas`, o comando seguinte executa esta construção:

```
matriculas = [al.matric for al in ct]
```

De outro modo (usando construção iterativa de listas), `matriculas` é a lista formada pelos campos `matric` de `al`, onde `al` é um aluno da lista original `ct`. Se o parâmetro `matricula` pertencer à lista `matriculas`, o seguinte comando irá extrair o índice associado (consultar bibliografia do curso):

```
matriculas.index(matricula)
```

A função retornará `None` se não encontrar `matricula` na lista `matriculas`.

A implementação da função `removeAluno()` se reduz a um único comando da linguagem Python (consultar a bibliografia do curso sobre o método `pop()` da classe `list`):

```
ct.pop(pos)
```

(observando que `pos` é uma posição válida – conferida fora da função). O significado está revelado nos seguintes comandos do algoritmo:

```
{salvar em aux o elemento da posição pos de ct}
{deslocar os elementos de ct de um item para a esquerda a partir de pos+1}
{eliminar o último elemento de ct}
```

Finalmente, a função `acessaAluno()` também possui um único comando, dessa vez, por sua simplicidade, idêntico ao algoritmo (para pegar um dado de uma lista é bastante informar seu índice):

```
ct[pos]
```

Realizemos interativamente os testes dessas funções. No *prompt* de comandos, fazemos a inclusão do módulo `caderneta.py` e, para referenciar explicitamente a lista de alunos chamada `Caderneta`, criamos a variável (de nome mais curto) `Cdta`:

```
>>> from caderneta import*
>>> Cdta = Caderneta
```

(Não esquecer que o IDLE deve encontrar os arquivos `aluno.py` e `caderneta.py`)

Experimentemos uma lista com apenas dois alunos. Sejam eles: "Manoel", "2006G0765", 6, 7.5, 20; e "Josefa", "2007G0741", 5.4, 9.5, 18:

```

>>> insereNoFim(NovoAluno(), Cdta)
Nome: Manoel
Matrícula: 2006G0765
AB1: 6
AB2: 7.5
Quant.faltas: 20
>>> insereNoFim(NovoAluno(), Cdta)
Nome: Josefa
Matrícula: 2007G0741
AB1: 5.4
AB2: 9.5
Quant.faltas: 18
>>> print(len(Cdta)) # conferindo o tamanho atual da lista
2
>>>

```

Observação: Devemos notar que, pela definição do parâmetro `ct`, a chamada da função acima pode ser feita, omitindo-se o segundo parâmetro: `insereNoFim(NovoAluno())` (pois o valor *default* de `ct` é `Caderneta`).

Um aluno deverá ser acessado a partir de sua matrícula. Para tanto, determina-se inicialmente seu índice na lista. Considerando que a matrícula pode não ser encontrada, o índice deve ser testado antes de usar a função de acesso. Por exemplo, localizar e exibir o aluno de matrícula '2007G0741':

```

>>> indice = posicaoAluno('2007G0741', Cdta)
>>> print(indice) #imprime indice ou se verifica: indice!=None
1
>>> alunoBuscado = acessaAluno(indice, Cdta)
>>> EscreveAluno(alunoBuscado)
Nome: Josefa
Matrícula: 2007G0741
AB1: 5.4, AB2: 9.5
Num. faltas: 18
Resultado: AP
>>>

```

Para remover um aluno da lista, será usada a função `removeAluno()` que precisa da correta posição do aluno (encontrada a partir da matrícula do aluno). Vamos remover o aluno de matrícula '2006G0765' (lembrando que, aqui, trata-se de um teste, mas, na verdade, este valor pode vir de diversas fontes – teclado, arquivo, etc.):

```

>>> indice = posicaoAluno('2006G0765', Cdta)
>>> print(indice!=None)
True
>>> removido = removeAluno(indice, Cdta) #remove salva o aluno
>>> EscreveAluno(removido)
Nome: Manoel
Matrícula: 2006G0765

```

```

AB1: 6.0, AB2: 7.5
Num. faltas: 20
Resultado: RA
>>>

```

Agora que o aluno de matrícula '2006G0765' foi removido, se fizermos um teste com a sua localização, o resultado será negativo, e o tamanho da lista estará reduzido de uma unidade:

```

>>> indice = posicaoAluno('2006G0765', Cdta)
>>> print (indice!=None)
False
>>> print (len(Cdta)) # conferindo o tamanho atual da lista
1
>>>

```

A implementação do algoritmo principal mostrado no **Exemplo 2.3** é mostrada a seguir. Reiniciamos o IDLE e editamos o arquivo `GerenciaCaderneta.py`, gravando-o no diretório `L212_02`, com o código seguinte:

```

from caderneta import *
Cdta = Caderneta #Cdta é um nome compacto para 'Caderneta'.
                #Explicita que lista está sendo manipulada
while(True):
    print ('-----')
    print (' (1)Inserir aluno')
    print (' (2)Remover aluno')
    print (' (3)Consultar lista')
    print (' (4)Encerrar')
    opc = input('Digite sua opção > ')
    if opc=='1':
        print ('Digite os dados de um aluno:')
        insereNoFim(NovoAluno(), Cdta)
        #também pode ser: insereNoFim(NovoAluno())
    elif opc=='2':
        indice = posicaoAluno(input('Remove aluno, matrícula: '))
        if indice!=None:
            print ('Aluno removido:')
            removido = removeAluno(indice, Cdta)
            EscreveAluno(removido)
        else: print ('--> Aluno inexistente!')
    elif opc=='3':
        indice = posicaoAluno(input('Acessa aluno, matrícula: '))
        if indice!=None:
            print ('Dados do aluno:')
            alunoAcessado = acessaAluno(indice, Cdta)
            EscreveAluno(alunoAcessado)
        else: print ('--> Aluno inexistente!')
    elif opc=='4': break
    else: print ('Opção inválida!')

```

Observação: O programa executa operações sobre apenas uma turma em cada execução (isto foi feito somente para torná-lo mais simples). Porém, uma variação pode ser implementada, onde a atual lista seja copiada em outra, ou em um arquivo, por exemplo, e limpando a memória para nova execução.

Executando o programa acima e inserindo os dois alunos seguintes (usando opção 1 do menu): "Manoel", "2006G0765", 6, 7.5, 20; e "Josefa", "2007G0741", 5.4, 9.5, 18, obtemos:

```
>>>
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção: 1
Digite os dados de um aluno:
Nome: Manoel
Matrícula: 2006G0765
AB1: 6
AB2: 7.5
Quant.faltas: 20
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção: 1
Digite os dados de um aluno:
Nome: Josefa
Matrícula: 2007G0741
AB1: 5.4
AB2: 9.5
Quant.faltas: 18
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção >
```

Consultando a lista (opção 3), tentando encontrar os alunos de matrículas '2007G0741' e '2007G0700', obtemos:

```
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção > 3
```

```
Acessa aluno, matrícula: 2007G0741
Dados do aluno:
Nome: Josefa
Matrícula: 2007G0741
AB1: 5.4, AB2: 9.5
Num. faltas: 18
Resultado: AP
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção > 3
Acessa aluno, matrícula: 2007G0700
--> Aluno inexistente!
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção >
```

Conforme o resultado acima, de fato, o aluno de matrícula 2007G0700 não está presente na lista. Vejamos agora um caso de remoção. Desejando-se remover um aluno de matrícula 2006G0765, escolhemos a opção 2:

```
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção > 2
Remove aluno, matrícula: 2006G0765
Aluno removido:
Nome: Manoel
Matrícula: 2006G0765
AB1: 6.0, AB2: 7.5
Num. faltas: 20
Resultado: RA
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção >
```

Realmente, se consultarmos a lista buscando este aluno após a operação acima, não mais o encontraremos. O resultado será o seguinte:


```
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção > 3
Acessa aluno, matrícula: 2006G0765
--> Aluno inexistente!
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção >
```

Para encerrar o programa, é bastante digitar a opção 4:

```
-----
(1) Inserir aluno
(2) Remover aluno
(3) Consultar lista
(4) Encerrar
Digite sua opção > 4
>>>
```

Exercício de autoavaliação

Realize os exercícios abaixo e discuta no fórum dos conteúdos da semana.

1 - Consideremos o problema do **Exemplo 1.5 (Subunidade 1.1.2)**. A questão menciona uma loja que mantém um cadastro das mercadorias que comercializa a fim de dar suporte aos movimentos como vendas, controle de estoque etc. Uma mercadoria foi modelada por um tipo abstrato com os atributos: código, denominação, preço de compra e preço de venda de uma mercadoria.

Escreva um programa em Python para gerenciar uma lista de mercadorias na memória do computador. Use o TAD `mercadoria` do citado exemplo e explore os recursos de listas da linguagem Python. O programa deverá permitir as seguintes operações (utilize um menu que deve constar também a opção de encerrar o programa):

-Inserir uma mercadoria na lista (inserir no fim da lista);

-Remover uma mercadoria da lista, tendo fornecido o seu código (executar esta operação somente se a lista não estiver vazia e se o código existir);

-Dado o código de uma mercadoria (e se o código existir), escrever seu código, sua denominação e seu lucro (em %);

*Sugestão: Proceder com as mercadorias de modo análogo aos alunos dos **Experimentos 01 e 02** desta **Subunidade 2.1.2**.*

2 - Consideremos o TAD `Veiculo` assim definido:

Atributos:

`placa` (string composta de 7 dígitos sendo três letras seguidas de quatro números),
`marca` (string que identifica o fabricante),
`modelo` (string com o nome do modelo dado pelo fabricante),
`ano` (ano de fabricação – uma string),
`cod` (código do proprietário – um número inteiro).

Interface:

`novoVeiculo()` - Sem parâmetros, retorna um `Veiculo` lendo os dados via teclado;
`mostraVeiculo(v)` - Escreve os dados do `Veiculo v` passado como parâmetro;
`finalPlaca(v)` - Recebe um `Veiculo v` como parâmetro e retorna o algarismo final da placa deste veículo.

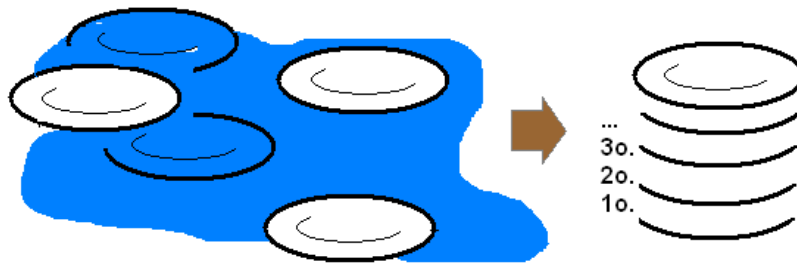
Escreva um programa para gerenciar um cadastro de veículos como uma lista na memória do computador. Use o TAD `Veiculo` e explore os recursos de listas da linguagem Python (guiando-se pelos experimentos dessa subunidade). O programa deverá permitir as seguintes operações (utilize um menu que deve constar também a opção de encerrar o programa):

- Inserir um veículo na lista (no fim da lista);
- Remover um veículo da lista, dada a placa deste (executar a operação somente se a lista não estiver vazia e se a placa de fato existir);
- Consultar o cadastro da seguinte maneira: Dada a placa de um veículo, se a mesma existir, escrever todos os dados do veículo; Se for dado apenas um inteiro de 0 a 9, o programa escreverá todos os dados dos veículos cujos finais de placa sejam o tal número.

Unidade II.2

Pilhas

Vimos que listas são dispositivos bastante práticos para a organização dos dados. Genericamente, os dados de uma lista podem ser inseridos ou retirados de qualquer um dos seus pontos. Fazendo-se certas restrições a esse acesso, as listas podem ganhar novas denominações. Para ilustrar um desses casos, tomemos, por exemplo, a tarefa doméstica de lavar pratos:



Os pratos, antes desorganizados, são empilhados em seguida.

Sabendo-se que vão ser lavados um a um, melhor será organizá-los, colocando um sobre o outro, empilhando-os. A pilha de pratos formada é uma lista especial, chamada de *pilha*: inserção e retirada de elementos acontecem em apenas um ponto da lista. Sendo assim, o primeiro prato que foi colocado na pilha será o último a ser retirado.

2.2.1 Conceituação

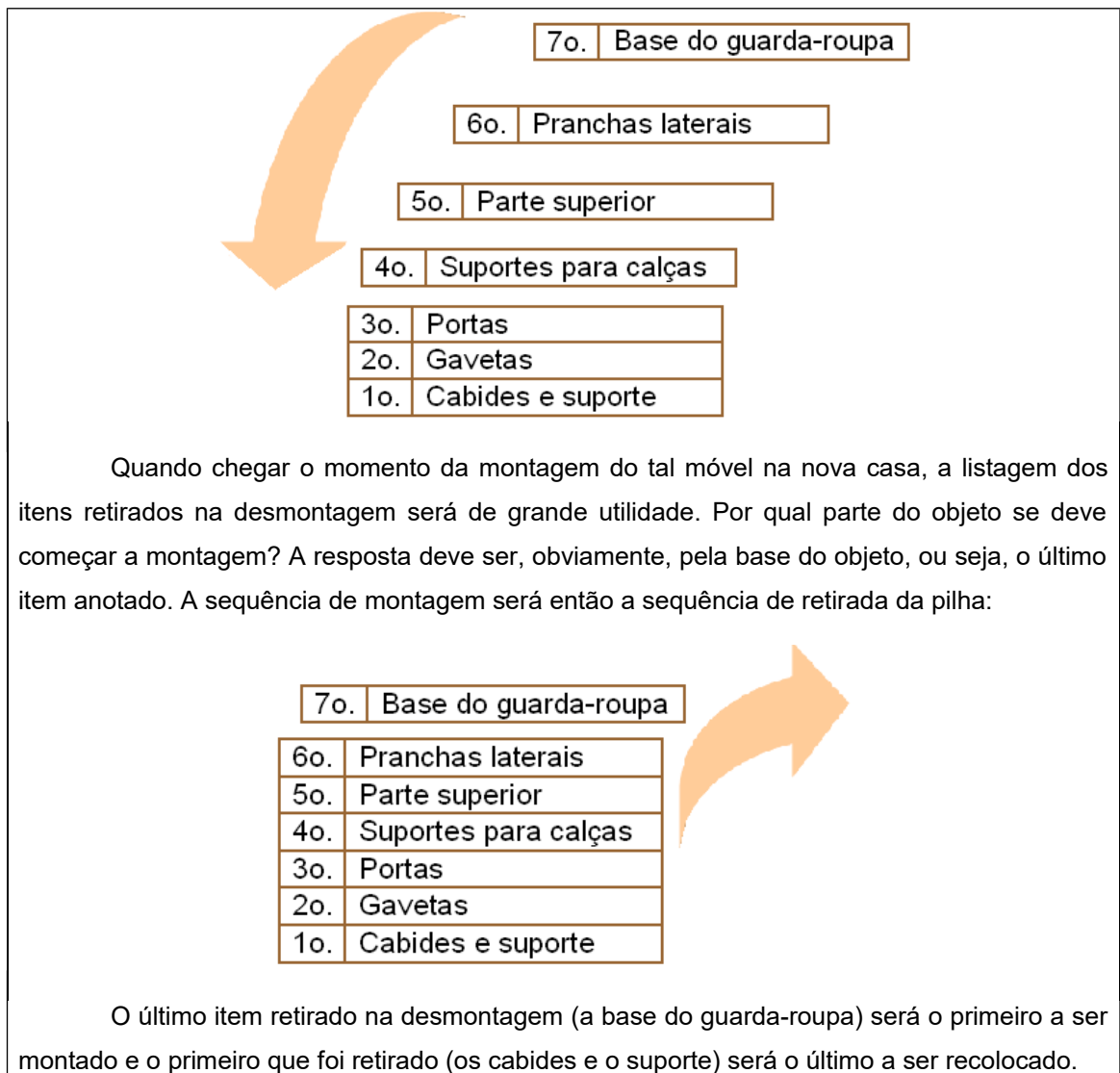
Para compreender bem o conceito de pilha, vejamos o caso seguinte.

Exemplo 2.4

Um guarda-roupa está sendo desmontado por um dono de casa (sem experiência no assunto) durante os afazeres de mudança para uma nova residência. Justamente por sua falta de experiência, essa pessoa vai anotando em sequência as partes que retira, conforme quadro abaixo:

1º.	2º.	3º.	4º.	5º.	6º.	7º.
Cabides e suporte	Gavetas	Portas	Suportes para calças	Parte superior	Pranchas laterais	Base

Os dados serão anotados, seguindo um empilhamento:

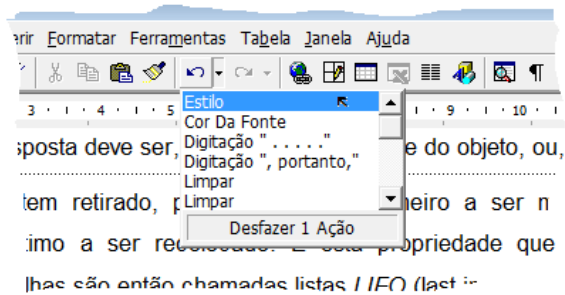


A propriedade de acesso observada no exemplo acima é o que dá a característica de pilha à lista construída durante o desmonte do guarda-roupa. Por isso, as pilhas são chamadas de *listas LIFO* (last in, first out): “último a entrar, primeiro a sair”.

O ponto único de acesso de uma pilha é denominado de *topo* (a partir dele os dados entram e saem da pilha). Por exemplo, na pilha de pratos, cada prato ensaboado colocado na pilha, é colocado no topo. Os que estão em cima (topo), serão enxaguados primeiramente. Durante a desmontagem do guarda-roupa, cada item anotado pertence ao topo da pilha (que vai sendo construída) e, na montagem, cada item recolocado no móvel está sendo retirado do topo da pilha (que vai, portanto, desfazendo-se até tornar-se vazia).

Exercício de autoavaliação

A lista de correções de um editor de textos é, na verdade, uma pilha. As alterações no texto são empilhadas e, ao desfazer uma delas, o digitador recupera o que havia modificado, conforme a ação indicada no topo:



Outros exemplos de aplicação direta de pilha em computação podem ser dados. Pesquise sobre aplicações de pilhas, dê exemplo de pelo menos uma delas, apresente e discuta no fórum dos conteúdos da semana.

2.2.20 TAD Pilha

Os atributos de uma pilha são também seus elementos e seu tamanho, visto que o conceito de pilha é extraído do conceito de lista. Uma referência mais direta ao topo da pilha pode ser incluída em determinadas implementações, mesmo sabendo que o início e o tamanho da pilha sirvam para determinar qual é o topo.

A interface corresponde ao conjunto das operações que são um subconjunto das operações sobre uma lista. As operações mais comuns são as seguintes:

- Criação - Criar uma pilha (vazia);
- Inserção (chamada de *push*) - Inserir um elemento no topo da pilha;
- Remoção (chamada de *pop*) - retornar um elemento do topo da pilha, removendo-o;
- Obtenção do topo - Informar qual elemento está no topo da pilha (sem removê-lo);
- Verificação de pilha vazia - Testar se a pilha está vazia.

Exemplo 2.5

Uma aplicação clássica de pilhas é seu uso na avaliação de expressões matemáticas (determinação de resultado). Para o computador, expressões como a seguinte,

"a * b + c / 2 - d", onde **a**, **b**, **c**, **2** e **d** são os operandos,

têm aspecto ambíguo. Isso exige um conhecimento prévio das prioridades das operações, ou seja, no cálculo do valor da expressão acima, por exemplo, é preciso conhecer as prioridades de "*", "+", "/" e "-". Conforme sabemos,

- (1) $a * b$ e $c / 2$ devem ser calculados primeiramente e
- (2) estes resultados parciais são então somados e, em seguida,
- (3) é subtraído d .

Essas prioridades ficam mais claras se forem usados parênteses (um para cada dupla de operandos e seu operador), da seguinte forma:

"(((a * b) + (c / 2)) - d)".

Existem outras maneiras de se guardar a prioridade das operações sem o uso de parênteses. Trata-se da *notação polonesa* (também chamada *notação prefixa*, ou, *prefix*) e da *notação polonesa reversa* (chamada *notação posfixa*, ou, *posfix*). Obs.: A notação com o operador escrito entre os operandos é chamada de *infixa* (ou, *infix*).

A mesma expressão acima (infixa) em notação prefixa é escrita da seguinte maneira:

“- + * a b / c 2 d”

Neste formato, da esquerda para a direita, a cada operador encontrado, aguarda-se o próximo par de operandos para que seja efetuada a operação indicada (cada operação efetuada produz um novo operando).

No formato posfix, a expressão se torna:

“a b * c 2 / + d -”

Neste formato, da esquerda para a direita, a cada par de operandos encontrado executa-se a operação indicada logo a seguir (cada operação efetuada produz um novo operando).

Podemos observar que (em comparação com a notação prefix) a notação posfix revela mais claramente a sequência de prioridade das operações verificada na notação com parênteses. Por isso mesmo, e por permitir agilidade (pois não há necessidade de digitação de parênteses), esta notação é preferida em muitas calculadoras científicas. A seguir, veremos um problema que é resolver uma expressão em notação pós-fixa, imitando o uso de uma calculadora, isto é, os itens da expressão serão digitados um a um, visualizando os resultados parciais (para demonstração, serão consideradas apenas as operações de adição, subtração, multiplicação e divisão). Para tanto, será usado o TAD Pilha.

O TAD Pilha

Os atributos de *Pilha* serão os de uma lista de dados numéricos:

Pilha = [] (uma pilha inicialmente vazia)

Quanto à interface, vão ser consideradas as operações para inserção (*Push()*), remoção (*Pop()*), leitura do topo da pilha (*Topo()*), verificação do tamanho (*tamanhoPilha()*) e eliminação dos dados da pilha (*limpaPilha()*). Dependendo da aplicação, podem ser incorporadas novas funções como, por exemplo, a verificação de pilha vazia.

Função *Push()* - Insere o item *n* no topo da pilha *p* (semelhante à função *insereNoFim()* do TAD *Caderneta* do **Exemplo 2.3**):

```
Defina Push(n, p) :  
    {alocar memória para o novo topo de p}  
    {atribuir o elemento n a este local}  
Fim_função
```

Função Pop(p) – Remove e retorna o elemento do topo da pilha *p* (semelhante à função `removeAluno()` do TAD Caderneta do **Exemplo 2.3**, sendo que a posição não precisa ser informada, pois já se sabe que é o topo):

```
Defina Pop(p) :
    {salvar em aux o elemento do topo de p}
    {eliminar a posição associada ao topo de p}
    retornar aux
Fim_função
```

Função Topo(p) - Retorna (sem remover) o elemento do topo da pilha *p*:

```
Defina Topo(p) :
    retornar p[posição do topo]
Fim_função
```

Função tamanhoPilha(p) - Retorna a quantidade de elementos da pilha *p*:

```
Defina tamanhoPilha(p) :
    retornar tamanho(p)
Fim_função
```

Função limpaPilha(p) - Esvazia a pilha *p* (mas a referência continua existindo):

```
Defina limpaPilha(p) :
    {eliminar os elementos de p mantendo a variável ativa}
Fim_função
```

Para construir a solução do problema proposto, serão elaboradas mais duas funções auxiliares:

A função `ehNumerico(item)` - verifica e retorna 'verdadeiro' se a *string* `item` é constituída de caracteres convertíveis em numérico e 'falso' se for um caractere qualquer;

A função `opera(a,b,op)` - Calcula e retorna o valor da expressão onde *a* e *b* são os operandos e *op* é o operador. Se o operador for de divisão e o operando *b* for zero, a função irá retornar *None* (usando a notação da linguagem Python).

Algoritmo

```
{Declarar o tipo abstrato Pilha}
```

```
Defina ehNumerico(item) :
```

```
    Se (item for convertível em número) então:
        retornar verdadeiro
    retornar falso
```

```
Fim_função
```

```
Defina opera(a,b,op) :
```

```
    Se op = '/' e b = 0 então: retornar None
    Se op = '+' então: retornar a + b
```

```

Se op = '-' então: retornar a - b
Se op = '*' então: retornar a * b
Se op = '/' então: retornar a / b

```

Fim_função

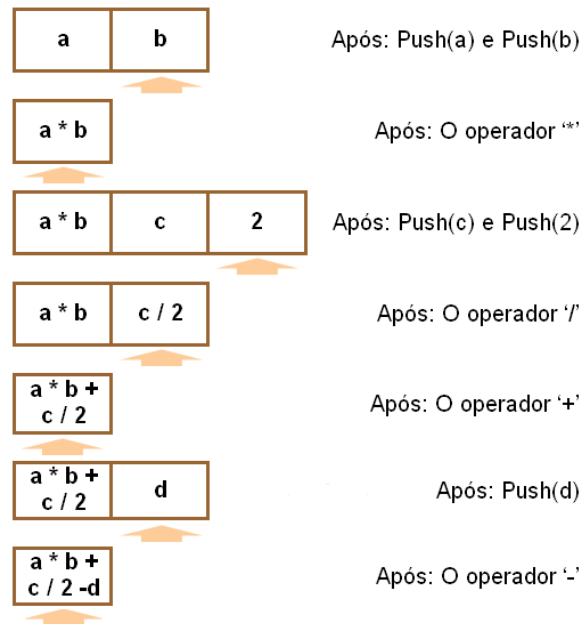
```

1 p = Pilha {p é a Pilha definida no TAD}
2 Escrever "Calculadora Polonesa Reversa,"f"- encerra a expressão "
3 limpaPilha(p) {elimina da pilha qualquer vestígio de cálculos anteriores}
4 fim ← Falso{repetição será interrompida quando a variável fim for 'Verdadeiro'}
5 Enquanto (fim = Falso) faça:
5.1 Ler item {lê um item da expressão posfixa}
5.2 Se (item for 'f' ou 'F') então: {se o usuário deseja interromper}
    fim ← Verdadeiro
    Senão Se ehNumerico(item) então:
        Push(item, p) {insere item na pilha convertido para número}
    Senão Se (item não for '+', '-', '*', nem '/') então:
        Escrever "Erro(caractere)!"
        fim ← Verdadeiro
    Senão Se (tamanhoPilha(p) < 2) então:
        Escrever "Erro(balaceamento)!"
        fim ← Verdadeiro
    Senão:
        oper2 ← Pop(p) {desempilha o segundo operador da próxima operação...}
        oper1 ← Pop(p) {e, seguida, o primeiro operador}
        result ← opera(oper1, oper2, item) {executa a operação}
        Se (result = None) então:
            Escrever "Erro(divisão por zero)!"
            fim ← Verdadeiro
        Senão:
            Push(result, p) {empilha o resultado parcial result}
            Escrever Topo() {escreve o topo da pilha, que é o valor de result}
6 Se (item for 'f' ou 'F') então:
    Se (tamanhoPilha(p) > 1) então:
        Escrever "Erro(balaceamento)!"
    Senão Se (tamanhoPilha(p) == 1) então:
        Escrever 'Resultado final:', Topo()

```

Fim-Algoritmo

Para compreender melhor a utilidade da pilha usada no exemplo acima, considere a digitação da expressão pós-fixa "a b * c 2 / + d -" (equivalente à expressão "(a * b) + (c / 2) - d" mencionada no início do citado exemplo). As linhas da figura abaixo descrevem os estados da pilha durante a execução do algoritmo:



Ao ser encontrado um operador entre os itens da expressão, dois operandos são retirados do topo da pilha, é efetuada operação indicada pelo operador e o resultado é inserido na pilha.

Laboratório - Pilhas

Objetivos

Conferir os algoritmos apresentados nesta unidade;

Identificar e aplicar os recursos disponíveis na linguagem Python para implementar o TAD Pilha.

Recursos e implementação

Abordamos a avaliação de expressões como sendo uma aplicação clássica de pilhas. Para o caso de notação infixa, a linguagem Python já tem pronta a função `eval()`. O resultado de uma expressão `expr` em notação infixa (fornecida sob o formato de cadeia de caracteres), é obtido executando-se a chamada `eval(expr)`, como no experimento a seguir.

Experimento 01

Interativamente, podemos determinar o resultado da expressão `expr = 'a * b + c / 2 - d'`, com `a`, `b`, `c` e `d`, correspondendo aos valores arbitrários 3, 4, 7.0, 5, respectivamente, da maneira seguinte:

```
>>> a, b, c, d = 3, 4, 7.0, 5
>>> expr = 'a * b + c / 2 - d'
>>> print ('Resultado:', eval(expr))
Resultado: 10.5
>>>
```

A expressão também pode ser inserida diretamente via teclado. Considerando a facilidade de o usuário digitar uma expressão inválida, podemos usar o recurso de *tratamento de exceções*, até agora não explorado nesse texto.

Esta terminologia, “tratamento de exceções”, é usada em computação para designar as operações de “defesa” do sistema contra os possíveis erros ocorridos durante a execução de um programa, normalmente decorrentes de atitudes do usuário. Aqui, usaremos, de forma breve, a seguinte sintaxe da linguagem Python:

```
try:
    bloco de comandos
except:
    ação de tratamento
else:
    ação livre de erro
```

Explicando de modo bastante simplificado, essa estrutura, funciona da maneira seguinte: o *bloco de comandos* é executado primeiramente. Se não ocorrer nenhum problema (nenhuma exceção), a *ação de tratamento* será ignorada e uma *ação livre de erro* será executada (no sentido inverso, após a tentativa de execução do *bloco de comandos*, se algum problema ocorrer, a *ação de tratamento* será executada e a *ação livre de erro* será ignorada). Obs.: A existência da *ação livre de erro* (else) é facultativa. A partir desse conhecimento, desejando-se um uso avançado, devem-se pesquisar os tratamentos que a linguagem Python já tem previstos (consultar o *Tutorial Python* de G. Van Rossum, da bibliografia complementar).

Criemos o arquivo `testEval.py` abaixo, armazenando no diretório `L222_01`:

```
try: # tenta ler e calcular a expressão
    expr = input('Digite uma expressão arit.> ')
    result = eval(expr)
except: # se ocorrer algum problema apenas imprime uma mensagem
    print ('Erro!')
else: # se não ocorrer problema imprime o resultado a expressão
    print ('Resultado:', result)
```

Nesse pequeno programa, primeiramente, a expressão aritmética `expr` será lida do teclado (segundo a sintaxe da linguagem Python) e haverá uma tentativa de cálculo (`eval(expr)`) armazenando o resultado na variável `result`. Se, por algum problema na expressão, o uso do comando causar um erro, a mensagem “Erro!” será impressa. Caso contrário, o conteúdo da variável `result` será impresso. Para testar, determinemos o resultado da expressão `3*4+7.0/2-5`:

```
>>>
Digite uma expressão arit.> 3*4+7.0/2-5
Resultado: 10.5
>>>
```

Simulando um erro, se for digitado $3*4+7.o/2-5$ (se, por engano, a letra **o** foi digitada em lugar de **0-zero**), encontramos:

```
>>>
Digite uma expressão arit.> 3*4+7.o/2-5
Erro!
>>>
```

Experimento 02

Neste experimento, implementaremos a calculadora pós-fixa do **Exemplo 2.5**. Criamos o diretório `L222_02` e nele colocamos os arquivos `pilha.py` e `calcPosfix.py` a seguir. O arquivo `pilha.py` abaixo implementa os algoritmos do TAD Pilha descritos no citado exemplo.

```
# Tipo Abstrato 'Pilha'
#ATRIBUTOS:-----
Pilha = []
#INTERFACE:-----
def Push(n, p = Pilha):
    p.append(n)

def Pop(p = Pilha):
    return p.pop()

def Topo(p = Pilha):
    return p[-1]

def tamanhoPilha(p = Pilha):
    return len(p)

def limpaPilha(p = Pilha):
    p[:] = []
#-----
```

O leitor desse texto deve:

- Conferir os comentários exibidos no programa consultando a bibliografia do curso;
- Nas funções, notar que elas fazem referência à `Pilha` (implementada por uma lista de Python) através de uma variável local chamada `p`. A igualdade `p = Pilha` permite que esta referência seja feita mesmo que o parâmetro seja omitido. Essa característica foi abordada durante o estudo de funções, na disciplina Algoritmo e Estrutura de Dados I.

O arquivo `calcPosfix.py` contém o código da solução do problema:

```
# Calculadora posfixa
from pilha import *
#-----
```

```

def ehNumerico(x):
    try: teste = float(x)
    except: return False
    return True
def opera(a, b, op):
    if op=='/' and b==0.0: return None
    return eval(str(a)+op+str(b))
#-----
print ('Calculadora Polonesa Reversa\n("f" - encerra a
expressão)')
limpaPilha()
while True:
    item = input('> ')
    if item in ['f','F']:
        break
    elif ehNumerico(item):
        Push(float(item))
    elif item not in '+-*/':
        print ('Erro(caractere)!')
        break
    elif tamanhoPilha() < 2:
        print ('Erro(balaceamento)!')
        break
    else:
        oper2 = Pop()
        oper1 = Pop()
        result = opera(oper1,oper2,item)
        if result == None:
            print ('Erro(divisão por zero)!')
            break
        else:
            Push(result)
            print ('parcial>',oper1,item,oper2,'=',Topo())
if item in ['f','F']:
    if tamanhoPilha() > 1: print ('Erro(balaceamento)!')
    elif tamanhoPilha() == 1: print ('Resultado final:',Topo())
#-----

```

Observações:

- Embora existam outras maneiras de implementação da função `ehNumerico()`, aqui, está sendo usada a tentativa direta de conversão em *float* e verificando se ocorrerá exceção.

- A função `opera()` usa a função `eval()` pré-definida, construindo a expressão literal `str(a)+op+str(b)` a partir dos parâmetros `a`, `op` e `b`, respectivamente.

Testaremos então o programa da seguinte maneira. Consideremos que exista uma conta corrente com um saldo de R\$ 2.520,00 e são feitos em seguida um saque de R\$ 230,00 e um depósito de R\$ 300,00. Assim, obtemos:

```

>>>
Calculadora Polonesa Reversa
("f" - encerra a expressão)
> 2520
> 230
> -
parcial> 2520.0 - 230.0 = 2290.0
> 300
> +
parcial> 2290.0 + 300.0 = 2590.0
> f
Resultado final: 2590.0
>>>

```

A seguir, podemos simular um engano na digitação. Por exemplo, a alteração da ordem pode causar um erro de balanceamento:

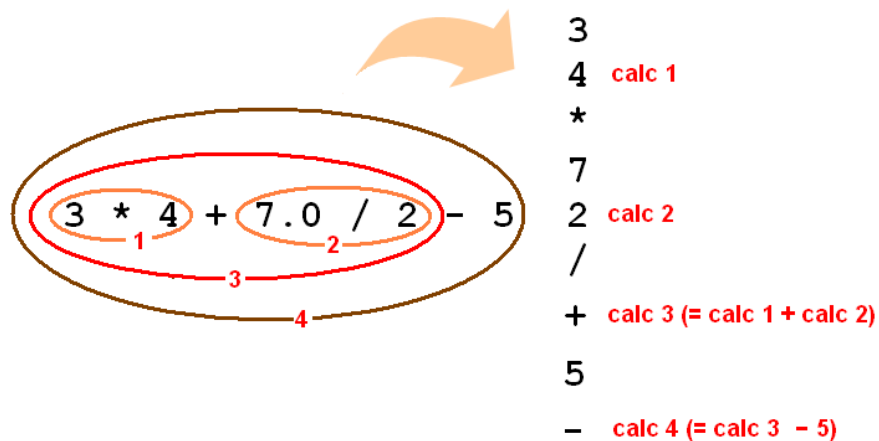
```

>>>
Calculadora Polonesa Reversa
("f" - encerra a expressão)
> 2520
> -
Erro (balanceamento) !
>>>

```

Esse tipo de erro irá acontecer porque, digitando-se o operador, pressupõe-se a existência de dois operandos, ou seja, nesse caso, a quantidade de operandos torna-se incompatível com a de operadores.

Vejamos o cálculo da expressão $3*4+7.0/2-5$. Em notação pós-fixa, esta expressão será digitada na ordem indicada na figura abaixo:



Após a execução, encontramos:

```

>>>
Calculadora Polonesa Reversa
("f" - encerra a expressão)
> 3
> 4

```

```

> *
parcial> 3.0 * 4.0 = 12.0
> 7
> 2
> /
parcial> 7.0 / 2.0 = 3.5
> +
parcial> 12.0 + 3.5 = 15.5
> 5
> -
parcial> 15.5 - 5.0 = 10.5
> f
Resultado final: 10.5
>>>

```

Exercício de autoavaliação

Resolva as questões abaixo e discuta no fórum dos conteúdos da semana.

1 - Esse exercício deve ser resolvido aproveitando o **TAD Pilha** implementado nesta subunidade. Elabore um programa em Python para:

a) Ler uma sequência de caracteres de tamanho qualquer e armazenar em uma pilha *P* (Através de chamadas sucessivas da função *Push()*);

b) Criar uma pilha *P2* (inicialmente vazia);

c) Copiar os elementos de *P* para *P2* (Elaborar e aplicar uma função a ser chamada *empilhaP2(P, P2)* que desempilha *P* e empilha em *P2* fazendo chamadas sucessivas das funções *Pop()* e *Push()*, respectivamente - Notar que, ao final desse procedimento a pilha *P* ficará vazia);

d) Mostrar o resultado da operação descrita no item anterior, exibindo os conteúdos das duas pilhas;

Importante! Antes de iniciar a resolução, verifique as seguintes orientações sobre o uso do TAD Pilha aqui:

- No TAD Pilha há a definição da pilha *default*, inicializada como `Pilha = []`. Esse fato está representado nas funções da interface pela indicação `p = Pilha`. Para usar a pilha *default* nesse exercício é bastante fazer: `P = Pilha`. Decorre disso que, se for omitido o parâmetro *p* das funções da interface, isso irá indicar automaticamente que a função respectiva age sobre a pilha *P*.

- Por uma característica do problema do **Exemplo 2.5** (avaliação de expressões), implementado no *Experimento 02*, nós não nos preocupamos com pilha vazia. Agora, considerando que a pilha *P* ficará vazia em algum momento, um teste deve ser realizado. Assim, esta questão requer que o TAD Pilha seja incrementado. Então, acrescente a função

pilhaVazia(p) abaixo, que retorna verdadeiro (True) se a pilha estiver vazia e falso (False), no caso contrario, da seguinte maneira:

```
# Tipo Abstrato 'Pilha'
#ATRIBUTOS:-----
Pilha = []
#INTERFACE:-----
...
(demais funções já implementadas)
...
def pilhaVazia(p = Pilha):
    return (len(p)==0) #retornará True se (len(p)==0)
#-----
```

2 - Faça um pequeno programa que lê do teclado uma expressão aritmética em notação infixa e escreve seu resultado (usar a função predefinida `eval()` de Python). Usar tratamento de exceções para repetir a leitura se a expressão digitada for inválida.

3 - Resolva o problema do *Item 6* do exercício de autoavaliação da **Subunidade 1.2.2**, desta vez, usando a estrutura de dados Pilha. *Observação: Não está sendo exigido construir novas estruturas (ou novos tipos). A questão pode ser resolvida aproveitando o TAD Pilha desenvolvido no texto. Como sugestão, construa uma pilha (implementada por uma lista de Python) com as letras da palavra dada. Em seguida, desempilhe sequencialmente para uma pilha temporária (como sabemos, esta última ficará com a sequência das letras invertida). Construa a nova palavra usando a pilha temporária. Depois, compare a palavra original com aquela obtida da pilha temporária e, logicamente, se estas forem idênticas, a palavra fornecida será um palíndromo.*

4 - Considere o caso mostrado no **Exemplo 2.4**. Elabore um programa que, seguindo ordem de desmontagem, cria a pilha com os itens do móvel em questão lendo suas denominações do teclado. Para simular a montagem, o programa vai desempilhando (pode ser a cada digitação da tecla ENTER), mostrando o item que está sendo montado e ainda informando quantos itens faltam para concluir a montagem.

5 - Considere uma lista de caracteres quaisquer, onde alternam-se letras, números (dígitos) e outros caracteres. Escrever uma função que recebe uma lista desse tipo e retorna outra em que os dígitos são mantidos na posição original e os não-dígitos são recolocados na sequência inversa. Exemplos:

-Dada a lista **[m, 8, G, X, 1, e, 5, t, 8]**, a função retornará **[t, 8, e, X, 1, G, 5, m, 8]**.

-Dada a lista **[a, \$, 2, b, %, 7, c]**, a função retornará **[c, %, 2, b, \$, 7, a]**.

Observação: Usar uma pilha na resolução.

Sugestão: Enquanto percorre a lista original, armazene numa pilha os caracteres que não são números. Percorrendo novamente a lista original, substitua os caracteres diferentes de números por aqueles que vão sendo desempilhados.

6 - Elabore um programa que lê uma expressão aritmética e verifica somente o balanceamento de parênteses. Isto é, verifica se aberturas e fechamentos se combinam, sem criticar as operações. O programa deverá emitir uma mensagem informando se os parênteses estão balanceados ou não, conforme o algoritmo dado abaixo. Para implementação, utilize os recursos apresentados nesta unidade.

Algoritmo

```
1 - Ler a expressão expr
2 - Fechou = True    #Fechou é um 'flag'
3 - Para cada caractere ch de expr:
    Se ch for '(', então:
        empilhar(ch)
    Senão:
        Se ch for ')', então:
            Se pilha estiver vazia precocemente, então:
                Fechou = False
            Senão:
                ch0 = desempilha()
                Se ch e o extraído ch0 não se casarem, então:
                    Fechou = False
4 - Se a pilha estiver vazia e Fechou = True, então:
    'Balanceada!'
    Senão:
        'Não balanceada!'
```

Fim-Algoritmo

Unidade II.3

Filas

Vemos nas cidades filas nos bancos, filas para compras de ingressos em cinemas, estádios etc. Por que nos parece justo que isso aconteça?



O emprego de filas

(fonte: <http://novosplanos.blogspot.com/2008/05/no-fique-na-fila.html>, acessado em 05/01/2011)

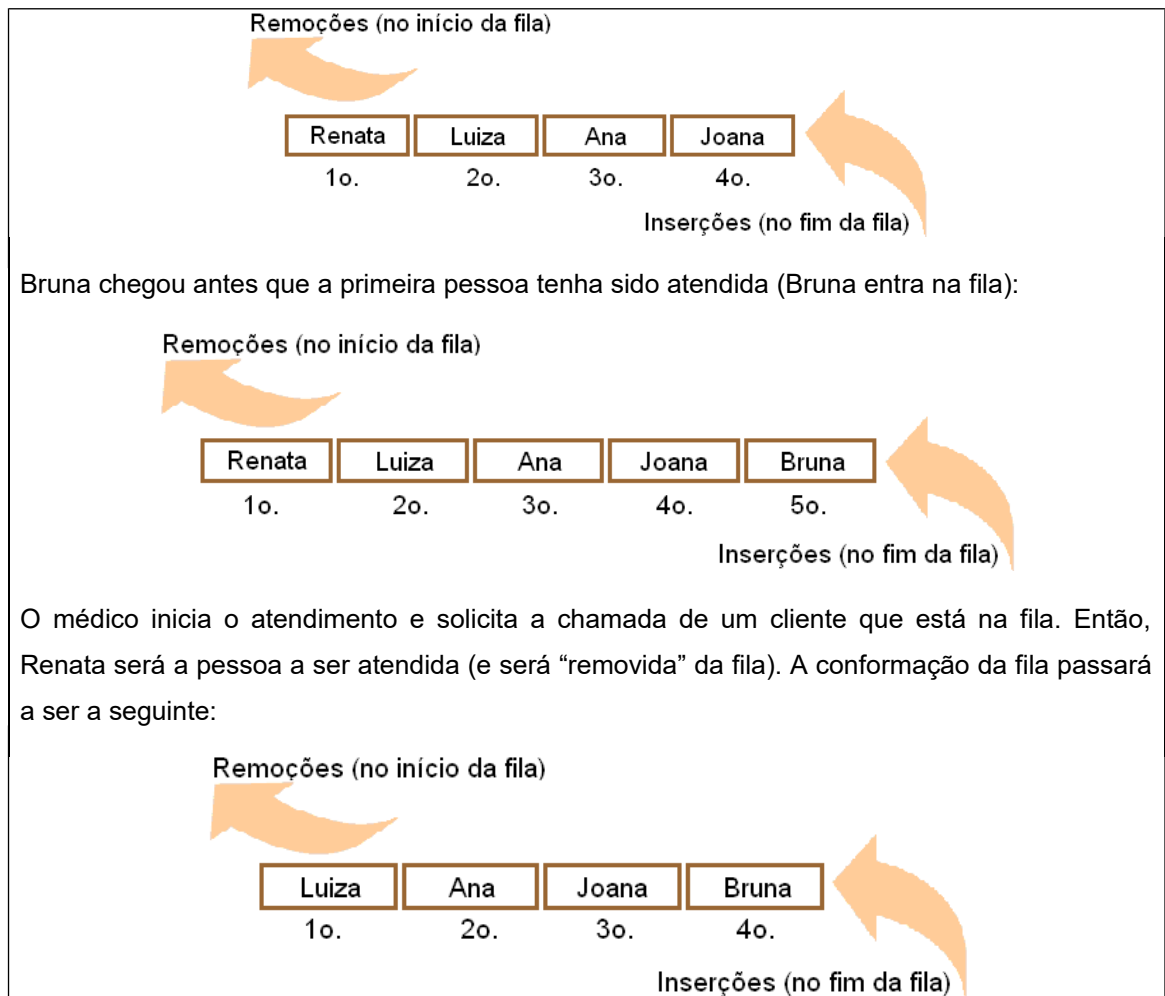
Em Maceió, por exemplo, não temos o hábito de formar filas nos pontos de ônibus e muitas vezes ocorrem impasses! Quem terá direito ao primeiro assento vago no coletivo (sem contar com os especiais)? A justiça da fila, todos nós certamente concordamos, está no fato de que o primeiro a chegar será o primeiro a sair da fila (a fila é definida por ordem de chegada), ou seja, no caso do ônibus, a primeira pessoa que chegou ao ponto deverá ser a primeira a embarcar (sair da fila), e a última que chegou será a última a sair da fila. Mas, a justiça será efetiva se alguns requisitos forem atendidos. Para começar, é terminantemente proibido "furar" a fila, isto é, podem-se acrescentar pessoas somente no fim da fila e será passageiro somente quem estiver no início da fila. Essa lógica é de grande utilidade em computação e governa uma estrutura que recebe o mesmo nome, *fila*, e tem larga aplicação, como em sistemas operacionais (fila de processos aguardando disponibilidade), por exemplo.

2.3.1 Conceituação

Notamos que o conceito de fila é também extraído do conceito de lista, isto é, uma fila é uma lista que possui exclusivamente dois pontos de acesso: As inserções ocorrem somente no fim da lista e as remoções somente no início. Vejamos o caso seguinte.

Exemplo 2.6

Num consultório médico, os clientes são atendidos por ordem de chegada. Assim que chegam, seus nomes são anotados pela secretária. A figura abaixo mostra uma fila com quatro clientes: Renata foi a primeira a chegar, e Joana é atualmente a última.



O exemplo acima expõe claramente a propriedade de acesso de uma fila: as inserções (chegada de clientes) acontecem no fim da fila e as remoções (atendimento e saída de um cliente da fila) acontecem no início. As filas são chamadas de *listas FIFO* (first in, first out): primeiro a entrar, primeiro a sair.

Exercício de autoavaliação

Pastas de processos acumulam-se sobre a mesa de certo executivo. Dado que os processos atendidos têm suas respectivas pastas retiradas da parte de cima, informe sob que condição (como devem ser a inserção e a remoção de pastas) a "pilha" de pastas deverá ser vista, de fato, como uma fila. Discuta sua resposta no fórum dos conteúdos da semana.

2.3.20 TAD Fila

Dado que o conceito de fila é extraído do conceito de lista, seus atributos são também seus elementos e seu tamanho. Em determinadas implementações, pode-se ainda incluir como atributos as referências ao primeiro e ao último elemento. A interface corresponde ao conjunto das operações que são um subconjunto das operações sobre uma lista. As operações mais comuns são as seguintes:

- Criação - Criar uma fila vazia;
- Verificação de fila vazia - Testar se a fila está vazia;
- Obtenção do elemento que está na primeira posição da fila;
- Inserção de um elemento - Inserir um elemento no fim da fila;
- Remoção de um elemento - Mostrar o elemento do início da fila, removendo-o;
- Verificação do tamanho da fila;

Exemplo 2.7

O **Exemplo 2.6** descreve brevemente o gerenciamento da fila de atendimento de um consultório médico, onde os clientes são atendidos por ordem de chegada, sendo anotados seus nomes. Desenvolveremos a seguir a solução desse problema por computador. Será utilizado o tipo abstrato `Fila`. O gerenciamento será feito da seguinte maneira: a partir de um menu de opções, pode ser anotado um cliente que acabou de chegar, retirado da fila o cliente atendido, ser informado o primeiro cliente da fila, como também o tamanho da fila. Ao ser encerrado o atendimento, serão mostrados os nomes dos clientes que não conseguiram ser atendidos nesta seção.

O TAD `Fila`

Os atributos de `Fila` serão os de uma lista de nomes:

`Fila = []` (uma fila inicialmente vazia)

Quanto à interface, vão ser consideradas as seguintes operações: inserção, remoção, leitura do primeiro cliente da fila, verificação do tamanho da fila, verificação de fila vazia e exibição de todo conteúdo da fila.

Função `insere(n, f)` - Insere o nome de um cliente no fim da fila (denominada localmente de `f`):

```
Defina insere(n, f):
    {alocar memória após o último elemento de f}
    {atribuir o elemento n a este local}
Fim_função
```

Função `remove(f)` - Retorna e remove o nome do primeiro cliente da fila:

```
Defina remove(f):
    {salvar em aux o primeiro elemento de f}
    {eliminar a posição associada ao primeiro elemento de f}
    retornar aux
Fim_função
```

Função `primeiroDaFila(f)` - Retorna o nome do primeiro cliente da fila, sem removê-lo:

```
Defina primeiroDaFila(f):
    retornar f[0]
Fim_função
```

Função tamanhoFila(f) – Retorna o tamanho da fila:

```
Defina tamanhoFila(f):
    retornar tamanho(f)
Fim_função
```

Função filaVazia(f) – Retorna 'verdadeiro' se a fila estiver vazia e 'falso' no caso contrário:

```
Defina filaVazia(f):
    Se (tamanhoFila(f)=0) então: retornar 'verdadeiro'
    Senão: retornar 'falso'
Fim_função
```

Função mostraFila(f) – Escreve todo conteúdo da fila:

```
Defina mostraFila(f):
    Para i ← 1...tamanhoFila(f), faça:
        Escrever f[i]
Fim_função
```

O algoritmo seguinte (usa o TAD Fila) representa a solução do problema do **Exemplo 2.6**:

Algoritmo

{Declarar o tipo abstrato Pilha}

1 f = Fila {f é a Fila definida no TAD Fila}

2 Faça:

 Escrever ' (1) Inserir cliente na fila'

 Escrever ' (2) Mostrar o primeiro da fila'

 Escrever ' (3) Atender cliente'

 Escrever ' (4) Tamanho da fila'

 Escrever ' (5) Encerrar atendimento'

 ler opc {Lê a opção do usuário}

 Se (opc == '1') então:

 Ler nome {Lê nome do cliente}

insere(nome) {Chama a função de inserção}

 Senão se (opc == '2') então:

 Se **filaVazia**() então: Escrever '> Não há clientes na fila'

 Senão: Escrever '> Primeiro da fila:', **primeiroDaFila**()

 Senão se (opc == '3') então:

 Se **filaVazia**() então: Escrever '> Não há clientes na fila'

 Senão: Escrever '> Cliente atendido:', **remove**()

```

        Senão se (opc == '4') então:
            Escrever '> Tamanho atual da fila:', tamanhoFila()
        Senão se (opc=="5") então: (vai para o fim do programa)
        Senão: Escrever "Opção inválida! "
    enquanto(opc ≠ "4")
3 Se filaVazia() então:
    Escrever 'Todos clientes foram atendidos'
    Senão:
        Escrever 'Clientes não atendidos:'
        mostraFila()
Fim-Algoritmo

```

Laboratório - Filas

Objetivos

Conferir os algoritmos apresentados nesta unidade;

Identificar e aplicar os recursos disponíveis na linguagem Python para implementar e testar o TAD Fila.

Recursos e implementação

Implementaremos neste laboratório o problema apresentado e resolvido nos exemplos 2.6 e 2.7. Usaremos como filas as listas pré-definidas da linguagem Python. Criemos o diretório L232 para armazenar os arquivos `fila.py` e `atendFila.py` seguintes.

O arquivo `fila.py` abaixo contém o TAD Fila:

```

# Tipo Abstrato 'Fila'
#ATRIBUTOS:-----
Fila = []
#INTERFACE:-----
def insere(n, f =Fila):
    f.append(n)
def remove(f = Fila): #retorna o primeiro eliminando-o
    return f.pop(0)
def primeiroDaFila(f = Fila):#retorna o primeiro sem eliminar
    return f[0]
def tamanhoFila(f = Fila):
    return len(f)
def filaVazia(f = Fila):
    return (len(f)==0)
def mostraFila(f = Fila):
    for nome in f: print (nome)
#-----

```

Devemos notar que cada função faz referência aos atributos representados pela lista chamada de *Fila* (localmente indicada pelo parâmetro `f = Fila`).

Armazenar no diretório L232 o arquivo `atendFila.py` abaixo. Esse arquivo é o código da solução do problema resolvido no **Exemplo 2.7**. *Observação:* Nesta implementação, foi acrescentado o comando `input('Digite ENTER\n')` após as mensagens, exclusivamente com o objetivo de parar a tela antes da exibição do menu.

```
from fila import *
print ('Atendimento de clientes')
while True:
    print ('-----')
    print (' (1)Inserir cliente na fila')
    print (' (2)Mostrar o primeiro da fila')
    print (' (3)Atender cliente')
    print (' (4)Tamanho da fila')
    print (' (5)Encerrar atendimento')
    opc = input('Digite sua opção > ')
    if opc == '1':
        nome = input('Digite o nome do cliente: ')
        insere(nome)
    elif opc == '2':
        if filaVazia(): print ('> Não há clientes na fila')
        else: print ('> Primeiro da fila:', primeiroDaFila())
        input('...Digite ENTER ')
    elif opc == '3':
        if filaVazia(): print ('> Não há clientes na fila')
        else: print ('> Cliente atendido:', remove())
        input('...Digite ENTER ')
    elif opc == '4':
        print ('> Tamanho atual da fila:', tamanhoFila())
        input('...Digite ENTER ')
    elif opc == '5': break
    else:
        print ('> Opção inválida!')
        input('...Digite ENTER ')

if filaVazia(): print ('Todos clientes foram atendidos')
else:
    print ('Clientes não atendidos:')
    mostraFila()
```

Vamos testar esta solução executando o programa `atendFila.py` com os dados indicados no **Exemplo 2.6**, ou seja, rodamos o programa e, usando a opção “1”, inserimos os nomes de Renata, Luiza, Ana e Joana (nesta ordem de chegada ao consultório).

```

Atendimento de clientes
-----
(1) Inserir cliente na fila
(2) Mostrar o primeiro da fila
(3) Atender cliente
(4) Tamanho da fila
(5) Encerrar atendimento
Digite sua opção > 1
Digite o nome do cliente: Renata
... Digite ENTER
... (e assim é feito para inserção dos demais clientes - até a cliente Joana)
...

```

Após a digitação dos primeiros clientes, se usarmos a opção “4” do menu, encontraremos:

```

-----
(1) Inserir cliente na fila
(2) Mostrar o primeiro da fila
(3) Atender cliente
(4) Tamanho da fila
(5) Encerrar atendimento
Digite sua opção > 4
> Tamanho atual da fila: 4
... Digite ENTER

```

Inserindo mais uma cliente de nome Bruna (usando a opção “1” do menu) e verificando o tamanho da fila (opção “4”), encontraremos que o tamanho atual da fila passa a ser 5:

```

-----
(1) Inserir cliente na fila
(2) Mostrar o primeiro da fila
(3) Atender cliente
(4) Tamanho da fila
(5) Encerrar atendimento
Digite sua opção > 1
Digite o nome do cliente: Bruna
... Digite ENTER
-----
(1) Inserir cliente na fila
(2) Mostrar o primeiro da fila
(3) Atender cliente
(4) Tamanho da fila
(5) Encerrar atendimento
Digite sua opção > 4
> Tamanho atual da fila: 5
... Digite ENTER

```

Estando o médico disponível para o atendimento, o nome de um cliente deve ser anunciado (é feita a leitura do primeiro da fila). É usada a opção “2”:

```
-----  
(1) Inserir cliente na fila  
(2) Mostrar o primeiro da fila  
(3) Atender cliente  
(4) Tamanho da fila  
(5) Encerrar atendimento  
Digite sua opção > 2  
> Primeiro da fila: Renata  
...Digite ENTER
```

Sendo esta pessoa atendida de fato, a opção “3” é usada (Podemos conferir em seguida que o tamanho da fila será reduzido de uma unidade):

```
-----  
(1) Inserir cliente na fila  
(2) Mostrar o primeiro da fila  
(3) Atender cliente  
(4) Tamanho da fila  
(5) Encerrar atendimento  
Digite sua opção > 3  
> Cliente atendido: Renata  
...Digite ENTER
```

Desejando-se apenas verificar qual será o próximo cliente, basta usar a opção 2 (sabemos que será Luiza):

```
>>>  
-----  
(1) Inserir cliente na fila  
(2) Mostrar o primeiro da fila  
(3) Atender cliente  
(4) Tamanho da fila  
(5) Encerrar atendimento  
Digite sua opção > 2  
> Primeiro da fila: Luiza  
...Digite ENTER
```

Ou seja, Luiza será a primeira no próximo atendimento.

Finalizando agora (opção “5”), teremos que algumas pessoas não conseguiram atendimento:

```
-----  
(1) Inserir cliente na fila  
(2) Mostrar o primeiro da fila  
(3) Atender cliente  
(4) Tamanho da fila  
(5) Encerrar atendimento  
Digite sua opção > 5  
Clientes não atendidos:  
Luiza  
Ana
```



```
Joana  
Bruna  
>>>
```

Luiza, Ana, Joana e Bruna não foram atendidas. Em uma versão mais completa, os dados devem ser armazenados em um arquivo em disco para que esses clientes (nesta ordem) sejam tratados como os primeiros do próximo atendimento.

Exercício de autoavaliação

Resolva as questões abaixo e discuta no fórum dos conteúdos da semana.

1 - Escreva um programa para gerenciar um conjunto de processos que chegam ao setor competente de uma determinada empresa pública. Cada processo tem um número de protocolo, o nome da pessoa interessada e um assunto. Aqui, para simplificar a solução do problema, os processos são localizados perfeitamente pelo protocolo (aqui, vamos representar por uma string). Processos que chegam vão para a fila. Os que vão para atendimento são retirados da fila. Também, antes de ser retirado da fila, é possível anunciar o processo a ser atendido. O programa também deixa disponível a opção de exibição da fila dos processos ainda não atendidos.

Sugestão: Monte a solução tomando o laboratório desta unidade como modelo. Podemos proceder com a fila de processos de modo análogo à fila de clientes. Sendo os processos localizados pelo número de protocolo, logo, é bastante considerar uma fila de números de protocolo.

2 - Construa uma nova solução para o problema do item anterior, obedecendo aos mesmos requisitos, sendo que, dessa vez, também manipule os demais dados de um processo. Para tanto, crie o tipo abstrato `Processo`. O TAD `Processo` já tem seus atributos (strings para representar o número de protocolo, o nome da pessoa interessada e o assunto), mas falta elaborar a interface com funções que permitam fazer com uma instância de `Processo` o que foi feito quando a representação era apenas por uma string (o número de protocolo), ou seja, elabore uma função para criar um `Processo`, lendo os atributos via teclado e outra para escrever os atributos de um `Processo` no monitor, isto é, esta versão deve manipular uma fila de instâncias do TAD `Processo`.

3 - Um certo hospital mantém um cadastro de pacientes que necessitam de transplante de coração. Cada paciente recebe um código (uma string numérica de seis dígitos) que o identifica no cadastro. Assim que o coração de algum doador chega ao hospital, o cadastro é consultado para que seja conhecido o próximo paciente a ser operado. Elabore um programa em Python para montar a fila dos pacientes candidatos a receptores (seguindo a ordem de ocorrência da necessidade de transplante e registrando via teclado o código de cada paciente). Em seguida, gerenciar esse cadastro como uma fila. Usando um menu, o programa deve permitir a inserção de um paciente na fila, o atendimento de um paciente, a exibição do código do próximo paciente a ser operado, o tamanho da fila de espera, a ordem na fila de um dado

paciente sendo fornecido seu código via teclado e encerrar o programa. Obs.: Elaborar e usar a função `ordemNaFila(cod, f = Fila)` como mais uma função do TAD Fila, que retorna a ordem do paciente (1 para 1º, 2 para 2º, e assim por diante) de código `cod` na fila `f`. Se o paciente não existir, a função retorna `None`.

Módulo III

Estruturas de dados não-lineares

Vimos que nas estruturas de dados lineares os dados são encaixados de tal forma que cada um tem apenas um sucessor. Todavia, em determinadas aplicações é requerido que qualquer dado possa ter mais de um sucessor. Com tal característica, temos as estruturas *não-lineares* *árvores* e *grafos* que serão abordadas nesse módulo de maneira introdutória. Em primeira análise, numa estrutura em árvore, os dados estão ligados como o organograma de uma empresa, onde existe uma distribuição hierárquica. Num grafo, os dados estão interligados como as estradas entre as cidades.

Objetivos

- Abordar os conceitos mais importantes sobre Árvores e Grafos;
- Identificar problemas resolvíveis através de árvores e Grafos;
- Implementar Árvores e Grafos aplicando o conceito de TAD e estruturas predefinidas na linguagem Python.

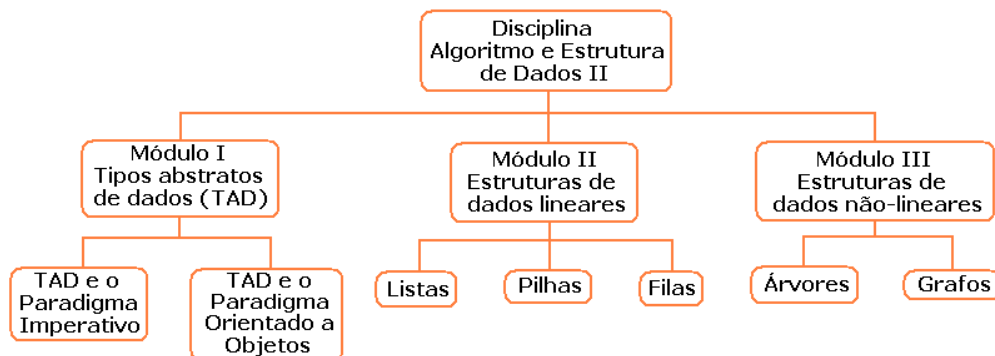
Unidades

- Unidade III.1 - Árvores
- Unidade III.2 - Grafos

Unidade III.1

Árvores

Vimos no **Módulo II** que as estruturas lineares são caracterizadas por reunirem seqüências de dados: O dado tem somente um antecessor e um sucessor. Mas, diversas aplicações requerem estruturas de dados onde eles podem ter mais de um sucessor. Consideremos o caso da estrutura em árvore. De uma maneira intuitiva, tomemos como exemplo de uma árvore a estrutura dessa disciplina para tratar das estruturas de dados avançadas:



**Organização da disciplina Alg. e Estr. de Dados II
- Formato em árvore**

Isto é, a figura exibe uma hierarquia onde temos três módulos e cada módulo, suas unidades. Por exemplo, os assuntos sobre listas, pilhas e filas são tratados respectivamente nas Unidades II.1, II.2 e II.3 subordinadas ao Módulo II. Essa árvore ainda pode ser expandida porque cada unidade tem subunidades e estas últimas, seções específicas. Devemos notar na estrutura em árvore que cada bloco se liga logicamente a apenas um bloco anterior, mas pode estar ligado posteriormente a múltiplos blocos.

Outros exemplos de organizações em árvore podem ser mencionados: O organograma de uma empresa, a distribuição dos arquivos no computador em diretórios e sub-diretórios, uma árvore genealógica, um livro, etc.

3.1.1 Conceituação

O exemplo mostrado na introdução dessa unidade bem representa o conceito de *árvore*. Uma árvore é uma estrutura que serve para armazenar dados de forma hierárquica. Na figura, observamos que os blocos representam denominações da estrutura lógica dos assuntos abordados na disciplina. Usando a terminologia adequada, aqueles blocos são chamados de *nós* (ou *vértices*) da árvore e estão ligados aos outros através de *ramos* (ou, *arestas*).

O nó de topo de uma árvore é chamado de *raiz*. Cada nó, exceto a raiz, está subordinado a um nó chamado de *pai* e subordina zero ou mais nós chamados de *filhos*. Os nós que não têm filhos são chamados de *folhas* e os nós que não são folhas são chamados de *nós internos*.

O número de filhos de um nó define o *grau do nó*. O grau do nó de maior grau define o *grau da árvore*. Numa dada árvore, um nó qualquer e seus filhos (seus *descendentes*) formam uma *subárvore* onde o nó referido é a raiz.

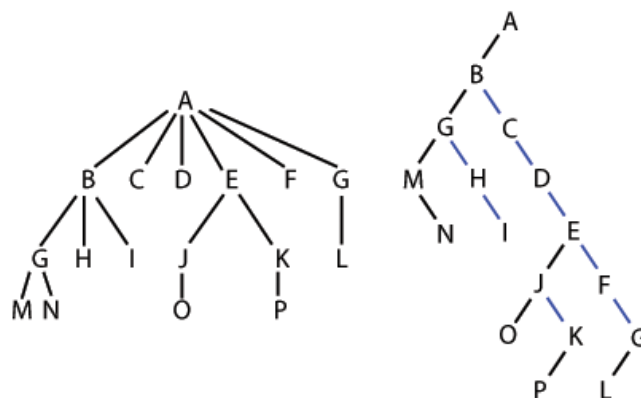
O movimento pelos ramos da árvore, de um nó para qualquer um dos seus descendentes, é feito através de um *caminho*. Dado um nó, os nós encontrados no caminho deste até a raiz são chamados de *ascendentes* (ou, *ancestrais*). A quantidade de ramos percorrida no caminho da raiz até o nó define a *profundidade* deste. O conjunto dos nós com a mesma profundidade é chamado de *nível*. A maior profundidade atingida dentre os nós serve para definir a *altura da árvore*.

Uma árvore é dita *ordenada* quando for estabelecida uma ordem entre os filhos de cada nó, podendo-se falar em primeiro filho, segundo, etc. Essa ordem (definida de acordo com a aplicação desejada) passa a fazer parte da definição da estrutura.

Particularmente, chamamos de *árvore binária* uma árvore ordenada em que cada *nó interno* tem no máximo dois filhos (um filho esquerdo e um filho direito). Ou seja, são árvores de grau dois.

Observação: Estudaremos neste curso apenas as árvores binárias, por ser suficiente para compreensão dos conceitos ligados a este assunto, além de serem estas as mais utilizadas nas aplicações computacionais.

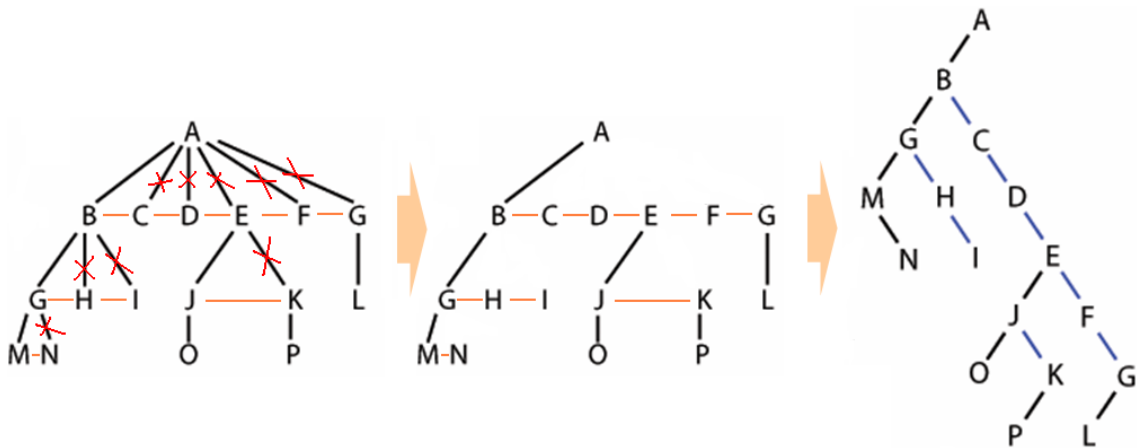
Sabe-se também que árvores de grau superior podem ser convertidas para binárias. Vejamos o exemplo mostrado na figura abaixo:



Conversão para árvore binária

(fonte: http://en.wikipedia.org/wiki/File:Nary_to_binary_tree_conversion.png)

Na figura, a árvore binária à direita é conseguida em dois passos (conforme figura adaptada abaixo). Iniciamos a conversão ligando os nós irmãos (filhos do mesmo pai). Depois, eliminamos os ramos que ligam o pai aos filhos deixando apenas a ligação com o primeiro filho.



Exemplo 3.1

Considere a árvore de números inteiros abaixo, onde o número 25 está na raiz:

A árvore foi construída de modo que cada nó contém um número inteiro sendo o filho à esquerda menor que o pai, e o filho à direita maior que o pai. Trata-se, então, de uma árvore ordenada (e não possui elementos repetidos). Por exemplo, o número 22 está na raiz da subárvore assinalada na figura. Os nós com os números 19 e 23 são seus descendentes. 19 é menor que 22 e 23 é maior que 22. Uma árvore com estas características é chamada de *árvore binária de busca*.

A árvore deste exemplo é binária porque cada nó tem no máximo dois filhos. É, portanto, o grau máximo de cada nó (grau dois). Portanto, é o grau da árvore.

Na figura, há um caminho assinalado da raiz até o valor 30. Este caminho mostra que os números 43 e 25 são os ascendentes de 30.

Os nós com os números 28 e 40 (têm profundidade três) são os de maior profundidade. Logo, a altura da árvore é três.

Propriedades

Em se tratando de árvore binária, pelo menos as duas propriedades seguintes são observadas:

- Se uma árvore binária tem n nós (ou, vértices), então terá $n-1$ arestas (ou, ramos). A árvore do **Exemplo 3.1** tem $n = 9$ nós e, portanto, $8 (= n - 1)$ ramos.

- Se uma árvore binária tem uma altura h (isto é, a medida da maior profundidade dos seus elementos é h), então terá h elementos, no mínimo, e $2^{h+1}-1 (= 1 + 2 + 4 + \dots + 2^h)$, no máximo. Quando a quantidade máxima de elementos for atingida, a árvore binária é dita *cheia*. A árvore do exemplo acima não é cheia. Isto é, não atinge a quantidade máxima de elementos que é $2^{3+1}-1 = 15$ (já que sua altura é três).

Caminhamento em árvore binária

Uma vez estabelecida na memória do computador, uma estrutura de dados fica disponível para visitas permitindo que faça: consulta, alteração de dados, inserção, remoção, etc. Em sendo uma árvore, este gerenciamento é feito percorrendo-se caminhos. Chamaremos esta operação de *caminhamento*.

Vamos considerar que desejamos escrever os números que estão na memória do computador segundo a estrutura em árvore do **Exemplo 3.1**. A visita pode ser realizada de acordo com os seguintes critérios:

- *Caminhamento por níveis* – A partir da raiz, são visitados os nós de cada nível.

De acordo com este critério, os números serão assim escritos:

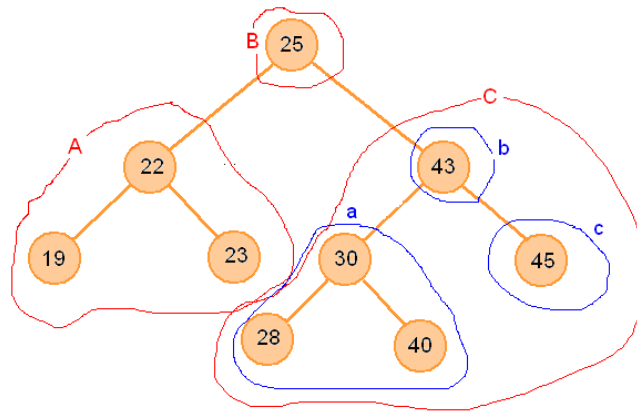
25, 22, 43, 19, 23, 30, 45, 28, 40

Ou seja, são impressos: a raiz (25), os elementos do nível 1 (22 e 43), os elementos do nível 2 (19, 23, 30 e 45) e os do nível 3 (28 e 40).

No caminhamento por níveis, o percurso é feito *em largura*. A seguir, são vistos os casos de percurso *em profundidade* (por ordem de subárvore).

- *Caminhamento em ordem* (ou, *caminhamento simétrico*) – Dado um nó, sua subárvore esquerda é simetricamente visitada em primeiro lugar. Em segundo lugar, o nó raiz é visitado (é efetuada a operação desejada, nesse caso, é apenas imprimir o número), e por último é visitada simetricamente a subárvore direita.

Os números da árvore do **Exemplo 3.1** serão escritos da maneira mostrada na figura abaixo:



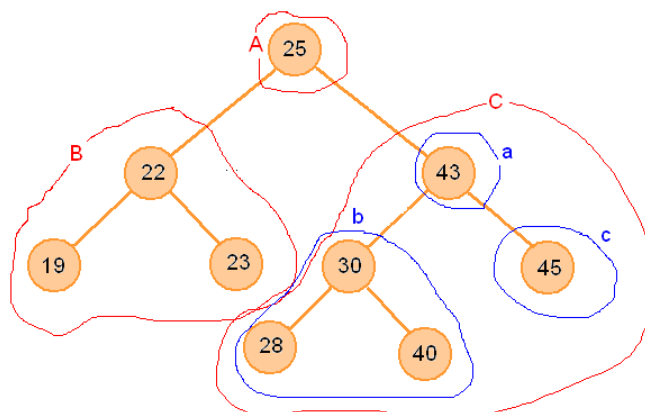
A subárvore **A** (da esquerda) será percorrida primeiramente. Logo depois, será visitado o nó **B** (imprimindo o valor 25) e em seguida será visitada a subárvore **C**. No caminhamento da subárvore **A** serão escritos: 19, 22 e 23 (notemos que o caminhamento simétrico é mantido). No caminhamento da subárvore **C** serão escritos: 28, 30 e 40 da subárvore **a**, o valor 43 do nó **b**, e, depois o valor 45 (do nó **c**, ao qual está reduzida subárvore da direita) . Finalmente, a saída será:

19, 22, 23, 25, 28, 30, 40, 43, 45

•*Caminhamento em pré-ordem* – O nó raiz é visitado logo inicialmente, em seguida é visitada em pré-ordem sua subárvore esquerda, e depois, a subárvore direita.

No exemplo em questão (ver figura abaixo), a árvore será percorrida partindo da visita ao nó **A** (imprimindo o valor 25). A subárvore **B** (esquerda) é percorrida em seguida, sendo escritos (mantido o caminhamento em pré-ordem) os números: 22, 19 e 23. A subárvore **C** (direita) é percorrida também em pré-ordem. Ou seja, o nó **a** é visitado (imprimindo o valor 43), depois, a subárvore **b** (imprimindo 30, 28 e 40 – em pré-ordem) e, finalmente, o nó **c** (imprimindo o valor 45). Ou seja, será escrito:

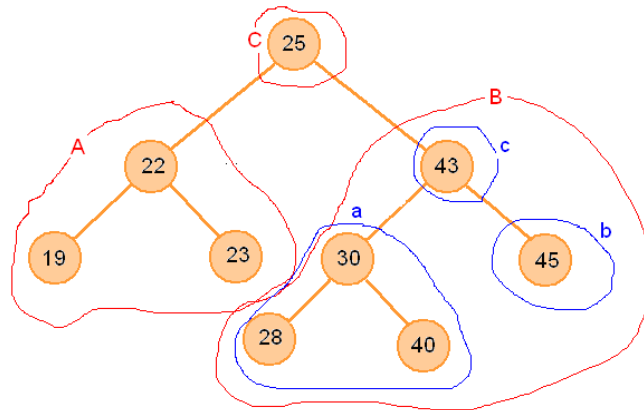
25, 22, 19, 23, 43, 30, 28, 40, 45



•*Caminhamento em pós-ordem* – Dado um nó, primeiramente é visitada em pós-ordem sua subárvore da esquerda, em seguida, a subárvore da direita. Somente ao final, é visitado o referido nó raiz.

A figura abaixo ilustra a sequência de caminhamento em pós-ordem da árvore numérica. As subárvores **A** e **B** são percorridas primeiramente e, por último, é visitado o nó **C**. Os nós da subárvore **A** são visitados em pós-ordem, sendo escritos os números: 19, 23 e 22. Da subárvore **B** é visitada sua subárvore **a** (são impressos os valores 28, 40 e 30 – em pós-ordem), o nó **b** (sendo escrito o valor 45) e o nó **c** (sendo escrito o valor 43). Finalmente, na visita ao nó **C** é impresso o valor 25. Ou seja:

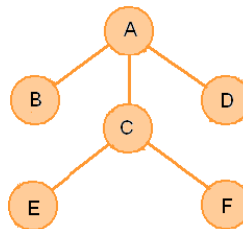
19, 23, 22, 28, 40, 30, 45, 43, 25



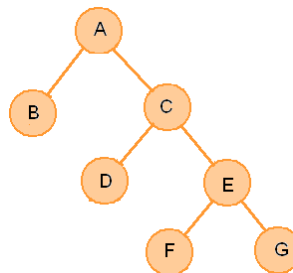
Exercício de autoavaliação

Resolva as questões abaixo e discuta no fórum dos conteúdos da semana.

1 - Converta a árvore abaixo numa árvore binária (faça o gráfico usando um editor de sua preferência):



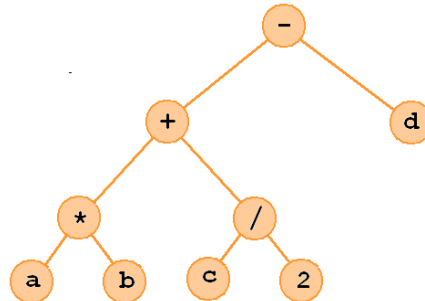
2 - A visita a um nó da árvore abaixo significa mostrar seu conteúdo. Execute os caminhamentos em ordem, pré-ordem e pós-ordem da citada árvore mostrando a sequência de letras encontrada.



3 - Expressões aritméticas (como aquelas estudadas na **Unidade II.2**) também podem ser representadas por árvores. Estas árvores, chamadas de *árvores de expressões*, possuem

operadores e operandos nos seus nós. Os operadores estão nos nós internos e os operandos (constantes ou variáveis) estão nas folhas.

Vejamos a expressão $a * b + c / 2 - d$ (do **Exemplo 2.5**). Obedecendo as prioridades das operações envolvidas, a correspondente árvore da expressão será:



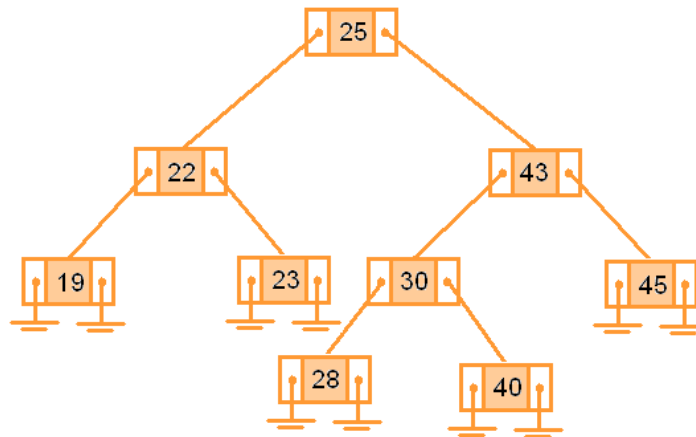
Realizando um caminhamento da árvore em ordem, encontramos a expressão da maneira indicada pela forma com parênteses seguinte: $((a * b) + (c / 2)) - d$. Execute caminhamentos em pré-ordem e em pós-ordem, e mostre a expressão resultante em cada caso (compare com as que foram encontradas no **Exemplo 2.5**).

3.1.20 TAD *Árvore*

Uma árvore pode ser implementada de várias maneiras. Adotaremos aqui uma representação encadeada. Isto é, cada nó da árvore binária será representado por um registro contendo o dado em si e, ainda, uma referência ao filho esquerdo e outra ao filho direito, da forma indicada na figura seguinte:



Portanto, um tipo abstrato *árvore* deverá fazer uso de um tipo *nó da árvore*. Assim, tratando-se de um encadeamento, para definir a árvore é bastante informar seu *nó raiz*. Um nó será uma folha quando suas referências aos filhos (esquerdo e direito) estiverem nulas (*None*, usando a notação em Python). A árvore do exemplo 3.1 pode então ser representada da seguinte maneira.



Um nó folha é indicado no gráfico por (com as duas referências nulas):



Nesse texto, o tipo nó terá interface com apenas a função para criação de um novo nó, mas, dependendo da aplicação, podem ser implementadas funções, por exemplo, para manipulação do conteúdo. A interface do TAD árvore poderá conter funções de criação da árvore, inserção de nós, remoção de nós, busca de valores, identificação das folhas, caminhamentos, etc. No nosso caso em particular, usaremos apenas as funções de inserção, caminhamento em ordem, pré-ordem e pós-ordem.

Exemplo 3.2

Consideremos a árvore exibida no **Exemplo 3.1**. Desejamos fazer os caminhamentos em ordem, pré-ordem e pós-ordem, conferindo os resultados discutidos na seção anterior. Neste exemplo, desenvolveremos o tipo abstrato *árvore* que faz uso do tipo `no_arv`, adotando-se uma representação encadeada da árvore em estudo.

O TAD `no_arv`

Os atributos serão definidos pelo tipo registro `no_arv`:

```
Tipo_registro no_arv:
    esq (tipo no_arv)
    dir (tipo no_arv)
    dado (inteiro)
Fim_registro
```

A interface será composta apenas da função `novo_no(valor)` – Esta função cria e retorna um registro do tipo `no_arv`, inserindo `valor` como `dado`.

```
Defina novo_no(valor):
    no ← no_arv()
    no.dado ← valor
    retornar no
Fim_função
```

O TAD *Árvore*

Uma árvore será construída pelo encadeamento de registros do tipo `no_arv`. Portanto, será definida pelo seu nó-raiz:

`no_raiz` (do tipo `no_arv`)

A interface será composta das funções recursivas: `insere()`, `caminhaEmOrdem()`, `caminhaPreOrdem()` e `caminhaPosOrdem()`.

Função `insere(valor, raiz)` - Insere um valor na árvore representada localmente por seu nó `raiz` mantendo sua ordem.

```
Defina insere(valor, raiz):  
  Se (raiz.dado está vazio) então:  
    raiz.dado ← valor {ou seja, insere na raiz se o campo estiver vazio}  
  senão:  
    Se (valor < raiz.dado) então: {insere à esquerda}  
      Se (raiz.esq não aponta outro nó) então:  
        raiz.esq ← novo_no(valor) {insere em novo nó}  
      senão: {chama insere() recursivamente na subárvore esq.}  
        insere(valor, raiz.esq)  
    Senão Se (valor > raiz.dado) então: {insere à direita}  
      Se (raiz.dir não aponta outro nó) então:  
        raiz.dir ← novo_no(valor) {insere em novo nó}  
      senão: {chama insere() recursivamente na subárvore dir.}  
        insere(valor, raiz.dir)  
    Senão: {valor já existe na árvore e não será inserido}  
      Escrever valor, "não será inserido!"  
Fim_função
```

Função `caminhaEmOrdem(raiz)` - Efetua o percurso em ordem.

```
Defina caminhaEmOrdem(raiz):  
  Se (raiz aponta algum nó) então:  
    caminhaEmOrdem(raiz.esq) {percorre a subárvore esquerda}  
    Escrever raiz.dado {visita a raiz, imprimindo o dado}  
    caminhaEmOrdem(raiz.dir) {percorre a subárvore direita}  
Fim_função
```

Função `caminhaPreOrdem(raiz)` - Efetua o percurso em pré-ordem.

```
Defina caminhaPreOrdem(raiz):  
  Se (raiz aponta algum nó) então:  
    Escrever raiz.dado {visita a raiz, imprimindo o dado}  
    caminhaPreOrdem(raiz.esq) {percorre a subárvore esquerda}  
    caminhaPreOrdem(raiz.dir) {percorre a subárvore direita}  
Fim_função
```

Função `caminhaPosOrdem(raiz)` - Efetua o percurso em pós-ordem.

```

Defina caminhaPosOrdem(raiz):
    Se (raiz aponta algum nó) então:
        caminhaPosOrdem(raiz.esq) {percorre a subárvore esquerda}
        caminhaPosOrdem(raiz.dir) {percorre a subárvore direita}
        Escrever raiz.dado          {visita a raiz, imprimindo o dado}
    Fim_função

```

O algoritmo seguinte faz uso dos tipos acima expostos para preencher uma árvore mantendo-a ordenada, e realizar os caminhamentos em ordem, pré-ordem e pós-ordem.

Algoritmo

```

{Declarar o tipo abstrato no_arv}
{Declarar o tipo abstrato arvore}
1 arvore = no_raiz {a árvore é representada pelo seu nó raiz}
2 Ler qtde {lê a quantidade e elementos para preenchimento da árvore}
3 Escrever "Digite os números (inteiros)"
4 Para i ← 1...qtde, faça:
    Ler n
    insere(n)
5 Escrever "Caminhamento em Ordem:"
6 caminhaEmOrdem(arvore)
7 Escrever "Caminhamento em Pré-Ordem:"
8 caminhaPreOrdem(arvore)
9 Escrever "Caminhamento em Pós-Ordem:"
10 caminhaPosOrdem(arvore)
Fim-Algoritmo

```

Laboratório - Árvores

Objetivos

Conferir os algoritmos apresentados nesta unidade;

Identificar e aplicar os recursos disponíveis na linguagem Python para implementar e testar o TAD Árvore.

Recursos e implementação

No experimento a seguir implementaremos os algoritmos desenvolvidos no **Exemplo 3.2**. Utilizaremos os recursos da linguagem Python para construir um encadeamento de nós conforme descrito no texto acima. Criemos o diretório L312 para armazenar os arquivos `no_arv.py`, `arvore.py` e `executaCamin.py` seguintes.

O arquivo `no_arv.py` contém apenas a definição de um nó da árvore:

```

# Tipo Abstrato 'no_arv'
#ATRIBUTOS:-----
class no_arv:
    esq = None
    dir = None
    dado = None

```

```

#INTERFACE:-----
def novo_no(valor):
    no = no_arv()
    no.dado = valor
    return no
#-----

```

O arquivo `arvore.py` a seguir especifica o tipo abstrato *árvore*, com seu `nó_raiz` e suas funções de manipulação. *Observação:* Esta implementação inclui a função `limpa()`. O objetivo é apenas ajudar nos testes ao se executar o programa repetidas vezes. A cada execução tem-se uma nova árvore de fato.

```

# Tipo Abstrato 'Arvore'
from no_arv import *
#ATRIBUTOS:-----
# Árvore
no_raiz = no_arv()
#INTERFACE:-----
def insere(valor, raiz = no_raiz):
    if raiz.dado == None: # se árvore estiver vazia, é iniciada
        raiz.dado = valor
    else: # se tem dado na raiz, insere valor...
        if valor < raiz.dado: # ... à esquerda
            if raiz.esq is None: raiz.esq = novo_no(valor)
            else: insere(valor, raiz.esq)
        elif valor > raiz.dado: # ... à direita
            if raiz.dir is None: raiz.dir = novo_no(valor)
            else: insere(valor, raiz.dir)
        else: print (valor, 'não será inserido!')

def caminhaEmOrdem(raiz = no_raiz):
    if raiz is not None: # se raiz aponta algum nó
        caminhaEmOrdem(raiz.esq)
        print (raiz.dado, end=' ')
        caminhaEmOrdem(raiz.dir)

def caminhaPreOrdem(raiz = no_raiz):
    if raiz is not None: # se raiz aponta algum nó
        print (raiz.dado, end=' ')
        caminhaPreOrdem(raiz.esq)
        caminhaPreOrdem(raiz.dir)

def caminhaPosOrdem(raiz = no_raiz):
    if raiz is not None: # se raiz aponta algum nó
        caminhaPosOrdem(raiz.esq)
        caminhaPosOrdem(raiz.dir)
        print (raiz.dado, end=' ')

def limpa(raiz = no_raiz): # limpa os campos do nó-raiz
    raiz.esq = None
    raiz.dir = None
    raiz.dado = None
#-----

```

Devemos notar que cada função da interface faz referência à árvore representada pelo seu `no_raiz`, localmente indicado pelo parâmetro `raiz = no_raiz`.

O arquivo `executaCamin.py` abaixo é o código da solução do problema do **Exemplo 3.2**.

```

from arvore import *
print ('Cria árvore e executa caminhamentos')
arvore = no_raiz #a árvore é representada pelo seu nó raiz
limpa(arvore)
qtde = int(input('Digite a quantidade de números: '))
print ('Digite os números (inteiros)')
for i in range(qtde):
    n = int(input('> '))
    insere(n)
print ()
print ('Caminhamento em Ordem:')
caminhaEmOrdem(arvore)
print ('\n')
print ('Caminhamento em Pré-Ordem:')
caminhaPreOrdem(arvore)
print ('\n')
print ('Caminhamento em Pós-Ordem:')
caminhaPosOrdem(arvore)
#-----

```

Testaremos a solução do problema executando o programa `executaCamin.py` com os dados indicados no **Exemplo 3.1**. Para que a árvore montada neste experimento seja idêntica àquela, digitaremos obedecendo a sequência dos dados por níveis, isto é: 25, 22, 43, 19, 23, 30, 45, 28, 40 (são nove elementos).

```

>>>
Cria árvore e executa caminhamentos
Digite a quantidade de números: 9
Digite os números (inteiros)
> 25
> 22
> 43
> 19
> 23
> 30
> 45
> 28
> 40

Caminhamento em Ordem:
19 22 23 25 28 30 40 43 45

Caminhamento em Pré-Ordem:
25 22 19 23 43 30 28 40 45

Caminhamento em Pós-Ordem:
19 23 22 28 40 30 45 43 25
>>>

```

Encontramos os resultados dos caminhamentos exatamente como esperávamos.

Exercício de autoavaliação

Resolva as questões abaixo e discuta no fórum dos conteúdos da semana.

1 - Faça uma adaptação do programa elaborado no laboratório L312 para construir uma árvore binária de busca em que os conteúdos dos nós são nomes de pessoas (no lugar dos números inteiros). Escolha arbitrariamente um conjunto de nomes, faça um teste com o programa e confira o resultado desenhando a árvore montada (usando um editor de sua preferência ou manualmente).

2 - A função abaixo lê um *arquivo do tipo texto*, cujo nome no disco é dado pelo parâmetro `nomeArq`, e retorna uma lista, `L`, com todas as linhas deste arquivo. Internamente (no corpo da função) o arquivo é chamado de `arq`. Se o arquivo (que deve estar no mesmo diretório do código do programa) não for encontrado, a função retorna `None`:

```
def leArq(nomeArq):
    try:
        # tenta abrir o arquivo para leitura
        arq = open(nomeArq, 'r')
    except:
        # se não conseguir, retorna None
        return None
    L = arq.readlines() # obtém a lista L das linhas do arq.
    arq.close()        # fecha o arquivo
    return L           # retorna a lista L
```

Para testar esta função, consideremos o arquivo “`nomes.txt`”, que pode ser criado utilizando-se o bloco de notas do *Windows* e contém certo número de linhas, cada uma com apenas um nome de uma pessoa. Tomemos este arquivo com o seguinte conteúdo:

```
Luiz
Ana
José
Joana
Bruna
```

O programa abaixo (chamemos de `leArquivo.py`) lê os nomes arquivados no disco (em “`nomes.txt`”), grava na memória do computador (na lista `lst`) e mostra a lista em seguida:

```
def leArq(nomeArq):
    try:
        # tenta abrir o arquivo para leitura
        arq = open(nomeArq, 'r')
    except:
        # se não conseguir, retorna None
        return None
    L = arq.readlines() # obtém a lista L com as linhas do arq.
    arq.close()        # fecha o arquivo
    return L           # retorna a lista L

lst = leArq('nomes.txt')
if lst==None:
    print ('Impossível abrir "nomes.txt"')
else:
    print('Lista com o conteúdo do arquivo:')
    print(lst)
```

A saída deste programa será:

```
>>>
```



```
Lista com o conteúdo do arquivo:  
['Luiz\n', 'Ana\n', 'José \n', 'Joana \n', 'Bruna']  
>>>
```

Use a função desenvolvida aqui para incrementar a solução do primeiro item desse exercício. Ou seja, a nova versão daquele programa deverá construir a árvore binária de busca lendo os nomes a partir de um arquivo do tipo texto.

Observação: A função acima copia exatamente o que está no arquivo texto e, pelo fato de ser um nome em cada linha, há marcas e final de linha ('\n') em alguns nomes da lista. Não sendo isto desejável, caracteres extras (e até espaços em branco) antes e depois de cada nome podem ser eliminados usando o método `strip()` da classe `str`. O trecho de código a seguir demonstra uma forma de resolver este problema:

```
for i in range(len(lst)):  
    lst[i] = lst[i].strip()
```

Se executarmos a impressão da lista após esta alteração, a nova saída será:

```
['Luiz', 'Ana', 'José', 'Joana', 'Bruna']
```

(Os nomes contidos na lista agora aparecem “limpos”, prontos para a construção da árvore).

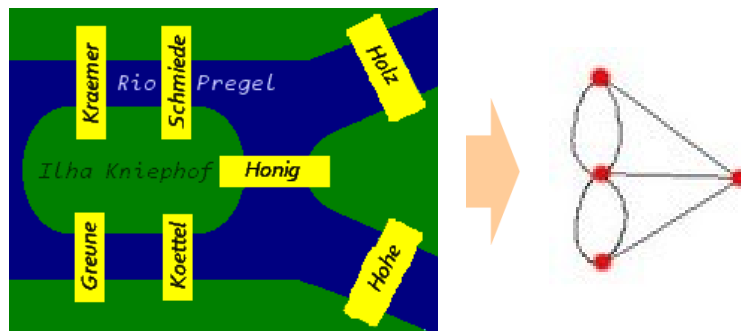
3 - Acrescente uma função no TAD árvore do laboratório desta subunidade (chame de `buscaFolhas(raiz)`), que escreve os dados dos nós que estão nas folhas da árvore (representada pelo seu nó `raiz`). Experimente a função elaborada. *Sugestão: A partir do nó corrente, iniciando pela raiz, a função verifica se as referências aos filhos, esquerdo e direito, são nulas. Se assim forem, isto significa que este é um nó folha e, então, a função escreve o dado contido no mesmo. Senão, a função é chamada recursivamente duas vezes. Uma, para buscar folhas na subárvore esquerda se a referência ao filho esquerdo for diferente de `None` e outra vez para buscar folhas na subárvore direita, se a referência ao filho direito for diferente de `None`.*

Unidade III.2

Grafos

Abordaremos nesta unidade a estrutura de dados *grafo*, com o modesto objetivo de conhecermos apenas as propriedades mais importantes e suas possibilidades de aplicação. Na verdade, o que há por trás da definição dessa estrutura é uma teoria matemática, a *Teoria dos Grafos*. Este fundamento matemático compõe a base teórica desta unidade, mas não será tratada aqui com profundidade, pois extrapola os objetivos deste curso.

A fim de termos uma visão contextual da área, tomaremos um exemplo clássico: O caso das *Pontes de Königsberg*². Conta-se que os habitantes da cidade russa de Königsberg desafiavam passear por toda cidade passando pelas sete pontes existentes naquela localidade apenas uma vez por cada uma delas. O fato é que não se conhecia qualquer pessoa que tivesse conseguido tal façanha. Esse problema logo despertou o interesse do matemático suíço *Leonhard Euler*. Euler usou uma estratégia simples para estudar a solução do problema. Reduziu as partes em terra a pontos simplesmente e representou por linhas os trajetos com passagem pelas pontes, chegando a um desenho de características peculiares. O gráfico, conseguido por Euler, corresponde a uma representação do que hoje chamamos de *grafo* (é citado na literatura como sendo o primeiro grafo da história).



Esboço das Pontes de Königsberg e o grafo correspondente construído por Euler (fonte: Adap. de <http://cognosco.blogs.sapo.pt/45217.html>, acessado em 07/09/2012)

Após sua análise, Euler chegou à conclusão de que, realmente, ninguém conseguiria vencer o desafio proposto. Somente seria possível passear por todos os caminhos, passando uma única vez em cada ponte, se houvesse, no máximo, dois pontos com saídas de linhas em quantidade ímpar.

A explicação é que cada ponto deve ter um número par de linhas (necessita-se de uma trajetória chegando e outra saindo). Os pontos que não precisam de uma trajetória para entrar

²Ler o texto de M. Maia, intitulado "As pontes de Königsberg" (blog "Cognosco", publicado em 25/05/2005). Disp. em <http://cognosco.blogs.sapo.pt/45217.html>. Acessado em 07/09/2012

e outra para sair são aqueles do início e final do percurso. Então, são estes os pontos com quantidades de linhas ímpares.

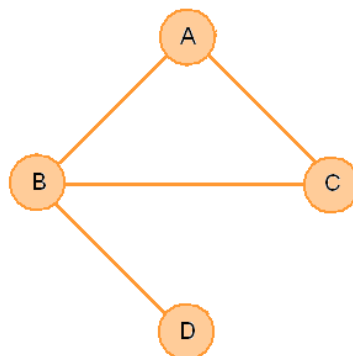
Os grafos servem para modelar uma grande quantidade de problemas do mundo real, normalmente quando está em jogo a conectividade de informação. Podemos citar a *Web*, as relações espaciais (mapas, redes viárias etc.) envolvidas nas redes físicas telefônicas, no estudo de rotas em logística, etc.

3.2.1 Conceituação

A partir do que foi exposto acima, podemos dizer que, para construir um grafo, precisamos de pontos e linhas que chegam ou saem desses pontos representando as relações entre eles. Os pontos são chamados de *vértices* ou *nós*, e as linhas são as *arestas* ou *arcos*. Portanto, um grafo é definido por dois conjuntos: o de vértices e o de arestas. Vejamos o caso a seguir, **Exemplo 3.3**, que aproveita essa ideia.

Exemplo 3.3

Certa pessoa 'A' recebeu um convite de uma pessoa 'B' para participar de uma certa rede social e aceitou. 'B' é um veterano dessa rede e tem como amigos as pessoas 'C' e 'D'. Após algum tempo, 'A' foi adicionada como amiga pela pessoa 'C'. Para representar o relacionamento entre essas quatro pessoas, podemos montar o seguinte gráfico (obviamente, este grupo é apenas uma pequena amostra do volume atual de amigos dessas pessoas):



'A', 'B', 'C' e 'D' são os vértices e reunimos no conjunto V : $V = \{ 'A', 'B', 'C', 'D' \}$. As arestas representam que pessoas estão relacionadas, formando o conjunto A : $A = \{ ('A', 'B'), ('A', 'C'), ('B', 'C'), ('B', 'D') \}$. Denotamos então o grafo por $G(V, A)$.

Terminologia

A Teoria dos Grafos possui uma vasta quantidade de termos, e alguns deles estão listados a seguir.

Vértices adjacentes

Dois vértices são *adjacentes* se estão ligados por uma aresta. Por exemplo, no **Exemplo 3.3**, os vértices 'A' e 'B', 'A' e 'C', 'B' e 'C', 'B' e 'D' são adjacentes.

Grau de um Vértice

O *grau de um vértice* é o número de arestas que incidem em cada vértice (isto é, chegam ou saem de cada vértice). No **Exemplo 3.3**, $\text{Grau}('A') = 2$, $\text{Grau}('B') = 3$.

Caminho

Um *caminho* é uma sequência de vértices de modo que cada um deles (exceto, naturalmente, o último da sequência) tem uma aresta para o vértice seguinte. No caminho, é definido o *vértice inicial* e o *vértice final*. Por exemplo, no **Exemplo 3.3**, uma possibilidade de 'A' conhecer 'D' é através do caminho: {'A','B','D'}. 'A' é o vértice inicial, e 'D' é o vértice final.

Comprimento de um caminho

O *comprimento de um caminho* é medido pela quantidade de arestas que ele contém (Obs.: Se isso envolver a passagem por certa aresta mais de uma vez, esta deve ser contada todas as vezes que se passar por ela). Exemplo: O comprimento do caminho citado acima ({'A','B','D'}), para 'A' conhecer 'D', é dois (Podemos observar na figura do **Exemplo 3.3**, que há um caminho mais longo que este, para o mesmo objetivo).

Caminho simples

Um caminho é dito *simples* quando não há vértices repetidos. Exemplo: O caminho citado logo acima, {'A','B','D'}, é também um caminho simples.

Ciclo

Um *ciclo* é um caminho simples com uma propriedade a mais: existe uma aresta ligando o último vértice ao primeiro vértice. Se quiséssemos mostrar no grafo do **Exemplo 3.3** um grupo de pessoas que se relacionam diretamente, podemos citar o ciclo {'A','B','C','A'}.

Subgrafo

Um *subgrafo* de um grafo dado é formado por um conjunto de vértices e um conjunto de arestas que são subconjuntos dos conjuntos de vértices e arestas do grafo original. Aproveitando o exemplo acima, o grupo dos amigos que se relacionam diretamente, {'A','B','C'}, pode ser destacado do grafo original {'A','B','C','D'} formando um subgrafo deste. Ou, então, o próprio grafo de relacionamento das quatro pessoas citadas pode ser considerado um subgrafo de todos os relacionamentos dessas pessoas na comunidade completa.

Grafo orientado (direcionado ou digrafo)

Um grafo é orientado quando as ligações dos vértices seguem uma determinada orientação. Por exemplo, no **Exemplo 3.3**, a relação é "...é amiga de...". Dizendo-se que as pessoas 'A' e 'B' são amigas, estamos dizendo que 'A' é amiga de 'B' e 'B' é amiga de 'A' (ou seja, não há escolha de um dos sentidos da relação). Apenas a título de comparação, se a relação fosse "...ama..." (do verbo "amar"), dizer 'A' ama 'B' não implica 'B' ama 'A'. Portanto, num grafo orientado, a relação 'A' ama 'B' seria representada por:



Observação: Num grafo orientado, se for verdade que também 'B' ama 'A', então outra aresta deve ser traçada no sentido oposto, da seguinte forma:

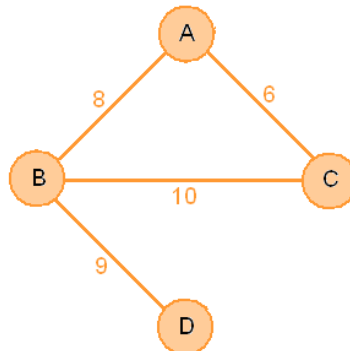


Grafo valorado

Um grafo é dito valorado se um valor, chamado de *peso*, for atribuído a um vértice ou aresta. Grafos valorados são de grande utilidade prática, onde os pesos podem representar custos, distâncias, capacidades, tempo (tempo em análise de trânsito, tempo de permanência etc.) etc.

Exemplo 3.4

Se pudéssemos medir o grau de amadurecimento da amizade entre as pessoas do **Exemplos 3.3**, atribuindo uma nota, digamos, de 0 a 10 (supondo-se haver relacionamentos pessoais e esse amadurecimento esteja relacionado com a quantidade de tempo em que são amigas), poderíamos construir um grafo valorado como o seguinte:



Representação de grafos

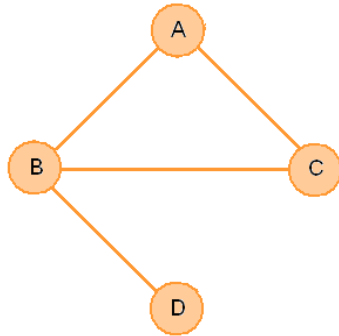
Determinar um grafo significa especificar seus vértices e suas arestas (da maneira como vimos no **Exemplo 3.3**). Os vértices podem ser letras, números etc. As arestas são determinadas, partindo da identificação dos vértices. Surgem então as diversas maneiras de representação de grafos e são definidas com objetivo de conduzir à implementação. Citamos aqui duas delas: a representação por *Matriz de Adjacência* e a representação por *Lista de Adjacência*. Aqui, não faremos uma análise mais profunda do assunto, quanto à aplicabilidade, eficiência etc.

Matriz de Adjacência

O texto mencionado na introdução, “*As Pontes de Königsberg*”, exibe uma animação de uma tabela que é, na verdade, a construção de uma matriz de adjacência, isto é, a relação de adjacência entre os vértices é mapeada numa matriz.

Exemplo 3.5

Podemos representar o grafo do **Exemplo 3.3** pela matriz de adjacência abaixo:

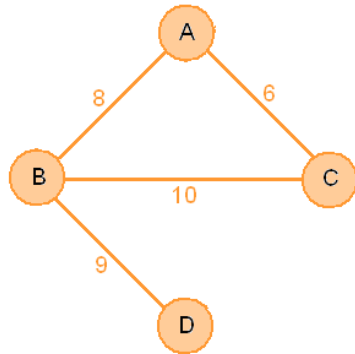


	'A'	'B'	'C'	'D'
'A'	0	1	1	0
'B'	1	0	1	1
'C'	1	1	0	0
'D'	0	1	0	0

Os índices das linhas e colunas apontam os vértices (as pessoas envolvidas) e os elementos da matriz indicam que vértices estão relacionados (que pessoas estão relacionadas).

Eventualmente, os elementos da matriz podem representar pesos num grafo valorado.

O grafo do **Exemplo 3.4** pode então ser representado pela matriz seguinte:



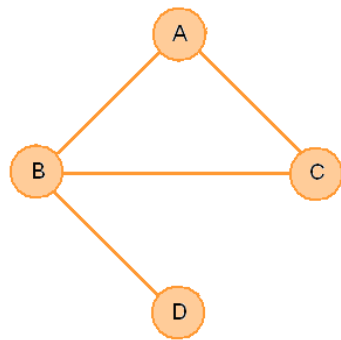
	'A'	'B'	'C'	'D'
'A'	0	8	6	0
'B'	8	0	10	9
'C'	6	10	0	0
'D'	0	9	0	0

Lista de adjacência

Esta representação consiste em listar todos os vértices do grafo e, em seguida, para cada um deles, anotar com que outro vértice este está relacionado.

Exemplo 3.6

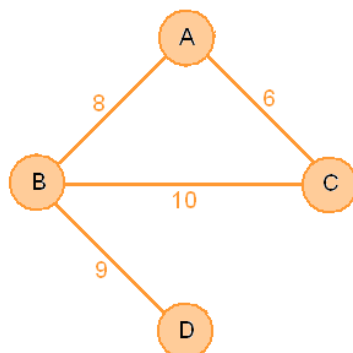
Podemos representar o grafo do **Exemplo 3.3** pela lista de adjacência seguinte:



Vértice:	Vértices adjacentes:
'A'	'B', 'C'
'B'	'A', 'C', 'D'
'C'	'A', 'B'
'D'	'B'

Cada item da lista corresponde a um vértice (uma pessoa). Ao lado, para cada vértice, existe a indicação dos vértices ao qual este se relaciona (aquelas com quem esta pessoa esteja relacionada).

No caso de grafos valorados, é bastante adotar uma maneira de indicar os pesos associados a vértices e/ou arestas. Uma dessas maneiras é usada abaixo para representar o grafo do **Exemplo 3.4** por uma lista de adjacência:



Vértice:	Vértices adjacentes:
'A'	('B',8),('C',6)
'B'	('A',8),('C',10),('D',9)
'C'	('A',6),('B',10)
'D'	('B',9)

Vê-se, na indicação dos vértices adjacentes, que também são indicados os pesos das arestas.

Exercício de autoavaliação

Realize os exercícios abaixo e discuta no fórum dos conteúdos da semana.

1 - A Teoria dos Grafos possui uma quantidade razoável de termos que não foram citados aqui. Então, dentre outros termos existentes, pesquise e escreva, inclusive dando exemplos, sobre o que são *arestas paralelas*, *laços*, *grafo simples* e *grafo completo*.

2 - Considere-se que se deseja construir um grafo valorado onde os vértices são as cidades de Santana do Ipanema, Olho D'Água das Flores, Maragogi, Maceió e Arapiraca (simbolizadas respectivamente por ST, OD, MR, MC e AR) e existem arestas ligando todos eles (trata-se de um grafo completo). Construa o tal grafo onde os pesos das arestas são as distâncias (km) entre as cidades citadas. *Observação:* Faça um desenho usando um editor de sua preferência ou, manualmente (escaneando o resultado, se desejar). Para estimar os pesos,

consulte o *Google Maps* (<http://maps.google.com.br>), anotando quilometragens de trajetos com as **menores distâncias possíveis** entre as citadas cidades.

3 - Represente o grafo montado no item anterior:

a) Por uma matriz de adjacência (Observe que cada elemento da matriz será a respectiva distância em *km* entre as cidades);

b) Por uma lista de adjacência (Observe que a cada cidade corresponderá a coleção de destinos possíveis e sua distância associada, ou seja, uma coleção de pares, cidade de destino, distância);

4 - As cidades envolvidas no item 2 destes exercícios são pólos da UAB, como sabemos. Suponhamos que um servidor da UFAL, partindo de Maceió, planeja visitar todas as outras cidades (passando uma única vez por cada uma) e voltar para Maceió ao final. Escreva **um caminho** que você considere que a distância percorrida por esse servidor seja a menor possível e mostre essa distância total. *Obs.: Note que esta questão não solicita qualquer tipo de código de programação.*

3.2.20 TAD Grafo

Normalmente, a construção do tipo abstrato Grafo envolve, como atributos, a quantidade de vértices, a quantidade de arestas e pesos (no caso de grafo valorado). As funções da interface envolvem:

Inicialização do grafo - criar um grafo vazio (adotando-se um tipo de representação);

Exibição do grafo (segundo a representação adotada);

Determinação do número de vértices;

Inserção de um vértice;

Remoção de um vértice;

Inserção de uma aresta;

Remoção de uma aresta;

Determinação do grau de um vértice;

Determinação de caminho, etc.

Diante do caráter introdutório da abordagem do assunto, aplicaremos um exemplo bastante simples (**Exemplo 3.7**, abaixo) baseado no **Exemplo 3.3**.

Exemplo 3.7

Consideremos o caso apresentado no **Exemplo 3.3**. Vamos tomar a seguinte questão: Se a pessoa 'A' entrou no grupo de amigos posteriormente, qual será um dos caminhos a se seguir para que 'A' venha a conhecer a pessoa 'D'?

Escolhemos representar o grafo (não-valorado) por uma lista de adjacência, tal como demonstra o **Exemplo 3.6** e será desenvolvido o tipo abstrato Grafo.

O TAD Grafo

Os atributos serão definidos por um vetor, chamado de `Grafo`, cujos índices são os vértices do grafo. Os vértices adjacentes serão localizados da seguinte maneira (segundo a tabela do **Exemplo 3.6**): `Grafo['A']` indica os vértices adjacentes ['B', 'C'], `Grafo['B']` indica os vértices adjacentes ['A', 'C', 'D'] e assim por diante

A interface será composta das funções `MostraGrafo()`, `insereVertice()` e `Caminho()` (*Observação*: Trata-se apenas de uma amostra de funções - como aquelas citadas na introdução desta subunidade, que o leitor pode incrementar facilmente usando estas como base lógica):

A função `MostraGrafo(grafo)` tem o objetivo de escrever o grafo por sua lista de adjacência, isto é, é mostrada a lista de vértices do grafo com seus vértices adjacentes.

```
Defina MostraGrafo(grafo):
    Escrever 'Lista de adjacência:'
    Para (cada vértice vert do grafo) faça:
        Escrever vert, '-', grafo[vert]
```

A função `insereVertice(vert, arestas, grafo)` insere no grafo um vértice (`vert`) e seus adjacentes reunidos na lista chamada `arestas`, se o mesmo ainda não existir.

```
Defina insereVertice(vert, arestas, grafo):
    Se vert (for um dos índices de grafo) então:
        Escrever 'O vértice', vert, 'já existe.'
        Retornar None {interrompe a função}
    Se (os adjacentes em arestas não forem vértices existentes) então:
        Escrever 'Adjacência de', vert, 'incorreta.'
        Retornar None {interrompe a função}
    grafo[vert] ← arestas {insere adjacências fornecidas como parâm.}
    Para (todos elementos v de arestas) faça:
        acrescentar vert como adjacência grafo[v]
```

Tomando o **Exemplo 3.3**, antes de a pessoa 'A' ser inserida no grupo de amigos, o grafo pode ser representado pela lista de adjacência seguinte:

Vértice:	Vértices adjacentes:
'B'	'C', 'D'
'C'	'B'
'D'	'B'

Para inserir o vértice 'A', devemos informar também a que vértices 'A' estará ligado, isto é:

'A'	'B', 'C'
-----	----------

Ao mesmo tempo, 'A' será incluído nas adjacências dos vértices 'B' e 'C'. O uso da função é da seguinte maneira `insereVertice(grafo, 'A', ['B','C'])`, com isso, conseguindo-se o resultado abaixo:

Vértice:	Vértices adjacentes:
'B'	'C', 'D', 'A'
'C'	'B', 'A'
'D'	'B'
'A'	'B', 'C'

A função `Caminho(inicio, final, caminho=[], grafo)` busca recursivamente no grafo um caminho começando com `inicio` e terminando com `final`:

```

Defina Caminho(inicio, final, caminho=[], grafo):
    Se (inicio e final não forem vértices existentes) então:
        retornar None {interrompe a função}
    Acrescentar inicio a caminho {inicialmente vazio}
    Se inicio = final então:
        retornar caminho
    Para (cada vert das adjacências grafo[inicio]) faça:
        Se vert (não estiver em caminho) então:
            novocaminho = Caminho(grafo, vert, final, caminho)
            Se novocaminho ≠ None:
                retornar novocaminho
    retornar None
    
```

Observação: Talvez não seja trivial o entendimento da função `Caminho()`. Porém, isso pode ser bem compreendido, fazendo o caminhamento graficamente. O objetivo aqui é apenas demonstrar alguns dos aspectos implementáveis desta teoria. Em linhas gerais, o procedimento é o seguinte. Uma vez fornecido o `inicio` e `final` de um caminho, a função toma (recursivamente) um dos vértices da adjacência do vértice `inicio` como novo início (de um novo caminho), mantendo o mesmo `final` (isto mantém a mesma lógica de busca da maneira como foi feita com o `inicio-final` de origem).

A pessoa 'A' entrou no grupo de amigos por último. Antes dela, o grupo era formado pelas pessoas 'B', 'C' e 'D'. No algoritmo seguinte, o grafo é inicializado considerando-se esse fato. Em seguida, a função `Caminho()` é usada a para encontrar um caminho em que 'A' venha a conhecer 'D' (veremos que este caminho não é o único).

Algoritmo

{Declarar o tipo abstrato Grafo}

1 {grafo inicial, representando o grupo de amizade sem a presença ainda da pessoa 'A'}

Grafo['B'] ← ['C', 'D'],

Grafo['C'] ← ['B'],

Grafo['D'] ← ['B']

```
2 MostraGrafo(Grafo)
3 insereVertice('A', ['B', 'C'], Grafo) {insere o vértice 'A' e sua adjacência}
4 Escrever 'Caminho para A conhecer D:', Caminho('A', 'D', Grafo) {tenta encontrar um caminho para 'A' e 'D' conhecerem-se}
Fim-Algoritmo
```

Laboratório - Grafos

Objetivos

Conferir os algoritmos apresentados nesta unidade;

Aplicar os recursos da classe *dict* da linguagem Python para implementar e testar o TAD Grafo.

Recursos e implementação

A linguagem Python possui uma estrutura pré-definida que se encaixa perfeitamente na implementação de grafos através de lista de adjacência. Essa estrutura é o dicionário e já a estudamos na segunda semana do curso. Dessa maneira, os atributos de um grafo serão extraídos dos atributos de um dicionário, e as funções da interface de um grafo serão manipulações dos métodos da classe *dict*.

Implementaremos neste laboratório o que está apresentado no **Exemplo 3.7**. Criemos o diretório L322 para armazenar os arquivos `grafo.py` e `AplicGrafo.py` seguintes.

O arquivo `grafo.py` abaixo contém o TAD Grafo:

```
# Tipo Abstrato 'Grafo'
#ATRIBUTOS:-----
Grafo = {}
#INTERFACE:-----
def insereVertice(vert, arestas=[], grafo=Grafo):
    if vert in grafo.keys():
        print ('O vertice', vert, 'já existe.')
        return None
    if not (set(arestas) <= set(grafo.keys())):
        print ('Adajcência de', vert, 'incorreta')
        return None
    grafo[vert] = arestas
    for v in arestas:
        grafo[v].append(vert)

def Caminho(inicio, final, caminho=[], grafo=Grafo):
    if not (inicio in grafo.keys() and final in grafo.keys()):
        return None
    caminho = caminho + [inicio]
    if inicio == final:
        return caminho
    for vert in grafo[inicio]:
        if vert not in caminho:
```

```

        novocaminho = Caminho(vert,final,caminho,grafo)
        if novocaminho != None:
            return novocaminho
    return None

def MostraGrafo(grafo=Grafo):
    print ('Lista de adjacência:')
    print ('vértices -> v.adjacentes')
    for vert in grafo.keys():
        print (vert, '- ', grafo[vert])
    print()
#-----

```

Armazenar no diretório L322 o arquivo `AplicGrafo.py` a seguir. Este programa implementa o algoritmo do **Exemplo 3.7**. Preenche o grafo de relacionamentos com as primeiras pessoas ('B', 'C' e 'D'), insere a pessoa 'A' neste grupo e, em seguida, busca um caminho em que 'A' passa a conhecer e 'D'.

```

from grafo import *
print('Grafo do relacionamento original')
Grafo['B'] = ['C', 'D']
Grafo['C'] = ['B']
Grafo['D'] = ['B']
MostraGrafo()
print("Grafo após a inserção de 'A' (com seus amigos 'B' e 'C')")
insereVertice('A', ['B', 'C'])
MostraGrafo()
print ('Um caminho para A conhecer D:', Caminho('A', 'D'))

```

Executando-se `AplicGrafo.py`, encontramos:

```

>>>
Grafo do relacionamento original
Lista de adjacência:
vértices -> v.adjacentes
C - ['B']
B - ['C', 'D']
D - ['B']

Grafo após a inserção de 'A' (com seus amigos 'B' e 'C')
Lista de adjacência:
vértices -> v.adjacentes
A - ['B', 'C']
C - ['B', 'A']
B - ['C', 'D', 'A']
D - ['B']

Um caminho para A conhecer D: ['A', 'B', 'D']
>>>

```

Ou seja, as pessoas 'A' e 'D' terminarão por se conhecerem, pelo menos através da pessoa 'B' (um amigo em comum). Esse caminho não é único. Por exemplo, outro caminho seria {'A', 'C', 'B', 'D'}. De fato, os resultados podem diferir dependendo do algoritmo utilizado. Um exemplo disso seria no caso de haver interesse em encontrar o caminho de comprimento mínimo. Então, uma nova versão deveria ser elaborada com esse objetivo.

Dado que o algoritmo segue uma ordem qualquer das adjacências, podemos verificar como ficará o resultado se invertermos a solicitação, isto é, tentemos encontrar um caminho para 'D' conhecer 'A'. Para tanto, executando `print(Caminho('D','A'))`, usando o interpretador interativamente na mesma seção em que foi executado o programa anterior, obtemos:

```
>>> print(Caminho('D','A'))
['D', 'B', 'C', 'A']
>>>
```

Isso significa que, a partir do início 'D', sua adjacência é somente 'B' (significando que, diretamente, 'D' conhece apenas 'B'). Ora, 'B' conhece 'A' e 'C' mas 'C' aparece primeiramente em sua adjacência. Finalmente, 'C' conhece 'A' e o caminho é concluído.

Exercício de autoavaliação

Os exercícios seguintes referem-se ao laboratório desta subunidade. Realize-os e discuta no fórum dos conteúdos da semana:

1 - Escreva uma sequência de chamadas da função `insereVertice()` que, a partir de `Grafo` inicialmente vazio, resulte no mesmo grafo do experimento desta subunidade em sua versão final (*Observação*: Podemos notar que, iniciando com o grafo vazio, a construção pode partir de qualquer vértice).

2 - Continue os experimentos do item anterior inserindo mais duas pessoas no grafo de relacionamentos (usando a função `insereVertice()`). No grafo resultante, determine dois caminhos de sua escolha usando a função `Caminho()`.

3 - Acrescente ao TAD Grafo a função `Grau(vert, grafo)`, que retorna o grau de um determinado vértice (fornecido no parâmetro `vert`) do `grafo` dado. Teste a função usando dados da versão do grafo obtida no item anterior. *Sugestão*: É bastante determinar a quantidade de vértices adjacentes do vértice passado como parâmetro.

4 - Elabore uma versão do programa desenvolvido no laboratório desta subunidade em que o grafo inicial tem apenas dois vértices, as pessoas 'B' e 'D'. Isto é,

```
Grafo['B'] = ['D']
Grafo['D'] = ['B']
```

O programa deverá permitir inserções de novas pessoas via teclado, exibição do estado atual do grafo e encerramento da execução, estando estas três opções em um menu. Experimente (e mostre os resultados obtidos) inserindo três ou mais pessoas além daquelas exibidas no texto (continue representando pessoas por letras maiúsculas).