Sumário

APIs usadas na disciplina

Este documento possui um resumo das estruturas que usamos em aula. A abordagem nesta seção se baseia de apresentar o uso de bibliotecas

Genéricos em Java

Genéricos em Java é uma técnica para fazer um estrutura genérica aceitar somente um tipo objeto pertencente a uma classe explicitamente declarada. Uma grande gama de classes permite esse tipo de uso, tais como listas, filas, pilhas, interfaces, etc.

Por exemplo. A lista abaixo é uma lista que somente vai aceitar objetos do tipo Acao.

```
List<Acao> acoes = new ArrayList<>();
```

Perceba que a especificação é feita colocando o nome da classe entre <Classe>. A parte curiosa é que em Java, se você não especificar o Genérico, o Java considerará que ele será o mais genérico possível, que é o Object.

Em outras palavras se você declarar uma variável: List acoes, ela será equivalente a você declarar List<Object> acoes.

A única coisa que muda é que esses objetos com genéricos vão aceitar somente objetos da classe especificada, ou na hierarquia de herança dela. As funções dos objetos serão inalteradas. Na nossa disciplina, entenda que somente precisamos entender que a classe colocada na lista serve de filtro. Não iremos criar classes com generico do zero.

Mais detalhes acesse os links:

- Devmedia: link.
- Um PDF aleatório que achei na internet: link.
- PUCRS: link.

Lista em Java

O que você precisa saber sobre listas em java para a cadeira de IA?

Considere a lista: List coisas = new ArrayList();

Basicamente, tu necessita saber como usar as funções básicas da biblioteca. São elas:

- Object coisa = coisas.get(int indice): retorna um objeto na posição índice.
- coisas.add(Object o): adiciona um objeto na lista;
- coisas.remove(int indice): remove um objeto na posição índice.
- coisas.size(): retorna o tamanho do da lista em valor inteiro.

Pense uma lista como um Array dinâmico, onde não há limite aparente no seu tamanho. Para a disciplina não importa como ela é programada internamente e sim como usar ela.

Mais detalhes de outras funções e como construir uma em:

- Texto DevMedia: link.
- Outro texto DevMedia: link.
- Slides IFSC: link.
- Introdução a lista de objetos (vídeo): link.
- Listas em Java com genéricos (outro vídeo): link.

Fila em Java

O que você precisa saber sobre fila para a cadeira de IA?

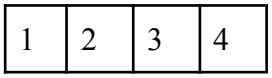
Basicamente, você precisa saber o que é uma fila e que funções tu precisa usar. Não há necessidade de saber como ela faz para executar internamento. Se tiveres curiosidade por isso sugiro olhar alguns dos materiais:

- Texto USP: link.
- Slides IFSC: link.
- Vídeo aleatório 1: link.

Então, vamos lá então.

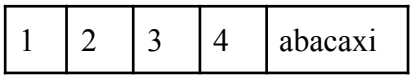
Filas são estruturas onde o primeiro objeto que você coloca nele, vai ser o primeiro a sair. Para isso, imagine os valores abaixo.
Há 4 valores, onde o 1 é a saída e o 4 é o último valor adicionado.

Saída Entrada



Digamos que eu queira adicionar nessa fila o valor “abacaxi”. A fila vai ficar da seguinte forma:

Saída Entrada

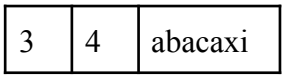


Se eu quiser remover um valor ela vai ficar assim:

Saída Entrada

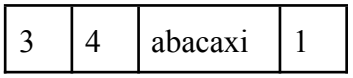
Se eu remover outro valor ele vai ficar assim:

Saída Entrada



Se eu adicionar o valor 1 novamente ele vai ficar assim:

Saída Entrada



Esse é o comportamento da fila em qualquer linguagem de programação. Em java há uma biblioteca programada com esse comportamento. O nome dela é **Queue**. Para declarar uma fila, você deve colocar no código algo assim:

```
Queue filaFronteira = new LinkedList();
```

Uma vez essa fila você vai ter acesso aos 3 métodos de seu comportamento:

- filaFronteira.add(valor); : adiciona um valor na fila.
- Object valor = filaFronteira.poll(); : remove um valor da fila.

- Object valor = filaFronteira.peek(); : pega o valor de saída da fila, mas não remove ele dela.

Pilha em Java

O que você precisa saber sobre pilha para a cadeira de IA?

Basicamente, você precisa saber o que é uma pilha e que funções tu precisa usar. Não há necessidade de saber como ela faz para executar internamente. Se tiveres curiosidade por isso sugiro olhar alguns dos materiais:

- Texto USP: link.
- Slides IFSC: link.
- Vídeo aleatório 0: link.
- Vídeo aleatório 1: link.

Então, vamos lá então.

Pilhas são estruturas onde o primeiro objeto que você coloca nele, vai ser o último a sair. Em outras palavras, o último valor a ser colocado, sai primeiro. Para isso, imagine os valores abaixo.

Há 4 valores, onde o 1 é a foi o primeiro a ser colocado e o 4 é o último valor adicionado.

Entrada e Saída

1	2	3	4

Digamos que eu queira adicionar nessa fila o valor “abacaxi”. A fila vai ficar da seguinte forma:

Entrada e Saída

1	2	3	4	abacaxi

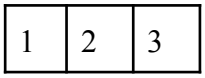
Se eu quiser remover um valor ela vai ficar assim:

Entrada e Saída

1	2	3	4

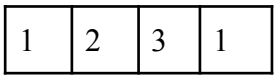
Se eu remover outro valor ele vai ficar assim:

Entrada e Saída



Se eu adicionar o valor 1 novamente ele vai ficar assim:

Entrada e Saída



Esse é o comportamento da pilha em qualquer linguagem de programação .Em java há uma biblioteca programada com esse comportamento. O nome dela é **Stack**. Para declarar uma pilha, você deve colocar no código algo assim:

Stack sequencia = new Stack<>();

Uma vez essa fila você vai ter acesso aos 3 métodos de seu comportamento básico:

- sequencia.push(valor); : adiciona o valor no final da pilha;
- Object valor = sequencia.pop(); : remove retira o último valor adicionado na pilha;
- Object valor = sequencia.peek(); : Pega o último valor adicionado na pilha, porém não remove ele.

For each em java

Mais infos em: link.

Em muitas linguagens de programação há simplificações de código para deixar a programação mais rápida. Em Java não é diferente. Uma dessas estruturas é o foreach.

Observe o código abaixo.

```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(String value : values) {
    System.out.print(value + ", ");
}
```

O for each no java basicamente tem dois parâmetros. for(valorIndividual : listaDeValores). Ele é uma simplificação do for clássico com valor inteiro de índice. Em outras palavras, ele é a forma simplificada do código abaixo:

```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
for(int i=0; i < values.size(); i++) {
    System.out.print(value.get(i) + ", ");
}
```

API para ordenamento sort

Link com maiores detalhes:

- Alura: link.
- Texto aleatório: link.
- Video: link.

O que você precisa saber sobre sort para a cadeira de IA?

A função sorte em java faz o ordenamento de uma lista/array. Para a cadeira é necessário saber o que a função faz e o resultado que ela tem. Não há necessidade de saber como a estrutura de dados e algoritmos internos funcionam.

Desta forma, vamos lá.

Todo array e lista pode ser ordenada. Para tanto o Java dispõe de uma classe especial. Collection. Vamos ver um exemplo.

```
// TODO code application logic here
    List<Integer> lista = new ArrayList<>();
    lista.add(5);
    lista.add(33);
    lista.add(1);
    lista.add(4);
    System.out.println("Antes de ordenar:");
    for(Integer i : lista){
       System.out.print(i + ", ");
    }
    // ordena
    Collections.sort(lista);
    // depois de ordenar
    System.out.println("Depois de ordenar:");
    for(Integer i : lista){
       System.out.print(i + ", ");
    }
```

Perceba que a operação **Collections.sort(lista);** está ordenando a lista. Ela não somente ordena valores numéricos, mas também String e objetos. Porém, em casos que as estruturas de ordenamento são mais complexas, é necessário implementar um comparador customizado.

Considere a classe abaixo.

```
public class Aluno {
   public String nome;
   public double nota;

   public Aluno(String nome, double nota) {
      this.nome = nome;
      this.nota = nota;
   }

   @Override
   public String toString() {
      return nome+" "+nota;
   }

}
```

A classe aluno possui nome e nota. A questão é que precisamos ordenar o aluno por nota. Para isso devemos implementar um comparador.

```
// ordena de forma decrescente
Comparator<Aluno> comparador = new Comparator<Aluno>() {
   @Override
   public int compare(Aluno o1, Aluno o2) {
      if(o1.nota > o2.nota){
         return -1; // se a nota 1 for maior que 2, coloca a nota 1 na antes de o2
      }
      if(o1.nota < o2.nota){
         return 1; // se a nota 1 for menor que 2, coloca a nota 1 depois de o2
      }
      return 0; // se forem iguais, não altera as posições
   }
};
```

Um comparador vai comparar todos os valores, dois por vez. A partir disso, o programador vai ter que dizer qual dos dois é maior ou menor que o outro. Para isso, é possível retornar 3 valores:

- -1 : se a nota 1 for maior que 2, coloca a nota 1 na antes de o2

- 1: se a nota 1 for menor que 2, coloca a nota 1 depois de o2
- 0: se forem iguais, não altera as posições

Para deixar crescente, basta inverter os símbolos > e <.

Após obter o comparador há duas formas de ordenar.

Forma 1: usando a classe Collections:

Podemos ordenar, passando a lista e o comparador no método sort do Collections.

```
List<Aluno> lista2 = new ArrayList<>();
        lista2.add(new Aluno("Orácio",75.0));
        lista2.add(new Aluno("Alice",95.0));
        lista2.add(new Aluno("Guilherme",35.1));
        lista2.add(new Aluno("Carlos",85.0));

        // ordena de forma decrescente
        Comparator<Aluno> comparador = new Comparator<Aluno>() {
            @Override
            public int compare(Aluno o1, Aluno o2) {
                if(o1.nota > o2.nota){
                    return -1; // se a nota 1 for maior que 2, coloca a nota 1 na antes de o2
                }
                if(o1.nota < o2.nota){
                    return 1; // se a nota 1 for menor que 2, coloca a nota 1 depois de o2
                }
                return 0; // se forem iguais, não altera as posições
            }
        };
        System.out.println("Antes de ordenar:");
        for(Aluno i : lista2){
            System.out.print(i + ", ");
        }
        // ordena
        Collections.sort(lista2,comparador);
        lista2.sort(comparador);
        // depois de ordenar
        System.out.println("Depois de ordenar:");
        for(Aluno i : lista2){
            System.out.print(i + ", ");
        }
```

Forma 2: usando a própria lista:

Toda lista possui um método sort interno. Nesse caso é possível usar diretamente o comparador na lista usando o método sort. Desta forma, não precisaremos do método Collestions.sort.

```
List<Aluno> lista2 = new ArrayList<>();
    lista2.add(new Aluno("Orácio",75.0));
    lista2.add(new Aluno("Alice",95.0));
    lista2.add(new Aluno("Guilherme",35.1));
    lista2.add(new Aluno("Carlos",85.0));

    // ordena de forma decrescente
    Comparator<Aluno> comparador = new Comparator<Aluno>() {
      @Override
      public int compare(Aluno o1, Aluno o2) {
        if(o1.nota > o2.nota){
          return -1; // se a nota 1 for maior que 2, coloca a nota 1 na antes de o2
        }
        if(o1.nota < o2.nota){
          return 1; // se a nota 1 for menor que 2, coloca a nota 1 depois de o2
        }
        return 0; // se forem iguais, não altera as posições
      }
    };
    System.out.println("Antes de ordenar:");
    for(Aluno i : lista2){
      System.out.print(i + ", ");
    }
    // ordena
    lista2.sort(comparador);
    // depois de ordenar
    System.out.println("Depois de ordenar:");
    for(Aluno i : lista2){
      System.out.print(i + ", ");
    }
```

Fila ordenada

Link com vídeo sobre: link.

A fila ordenada é uma classe especial de fila que reordena todos os valores após um novo valor ser adicionado. O seu funcionamento é igual a uma fila convencional com a adição de um comparador.

Em Java a classe **PriorityQueue** realiza esse comportamento. Observe o código abaixo que foi usado no algoritmo A* em aula (Classe BuscaAstar linha 23).

```
// cria uma fila ordenada de vertices, do com menor avaliação para o com maior
 PriorityQueue<Vertice> filaOrdenada = new PriorityQueue<>(new Comparator<Vertice>() {
        @Override
        public int compare(Vertice o1, Vertice o2) {
            // compara se primeiro é maior que o segundo
            if (o1.avaliacao > o2.avaliacao) {
                return 1; // se for retorna 1
            }
            // compara se primeiro é menor que o segundo
            if (o1.avaliacao < o2.avaliacao) {
                return -1; // se for retorn -1
            }
            // se os dois forem iguais retorna 0
            return 0;
        }
    });
```

No restante ela funciona como uma fila normal, onde você vai ter acesso aos 3 métodos de seu comportamento:

- filaOrdenada.add(valor); : adiciona um valor na fila. Ele será ordenado pelo comparador.
- Object valor = filaOrdenada.poll(); : remove um valor da fila.
- Object valor = filaOrdenada.peek(); : pega o valor de saída da fila, mas não remove ele dela.

Outras Estrutura de dados

Grafo

Grafo é uma das estruturas mais gerais que se usa em computação. A maior parte dos problemas podem ser abstraídos nesse tipo de estrutura, sejam mapas, máquinas de estado, movimentos, combinações de valores, etc.

Nesta representação nós possuímos 2 principais elementos:

- Vértices (Vertices): também conhecidos como nós, não estruturas que contém um significado para o problema.
- Arestas (Edges): ligações/relações entre os vértices. Elas podem ser direcionadas ou bidirecionais.
    - Em alguns problemas podemos usar pesos nas arestas. Ex.: A distância em Km entre duas cidades.

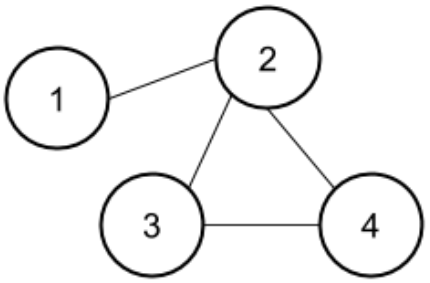
Esses vértices e arestas podem ser representados de forma gráfica e em forma escrita. A representação escrita é mais formal, e segue uma notação onde G representa um grafo constituído com um conjunto G = (V, E), onde V são vértices e E as arestas. Considere o exemplo:

Onde G = (V, E):

V = { 1, 2, 3, 4}

E = { 1–2, 2–3, 2–4, 3–4 }

Tal grafo é comumente representado como podemos ver abaixo:



De forma adicional, alguns grafos podem possuir mais um campo W (peso) de cada aresta que é representado por um campo adicional em G. Por exemplo:

Onde G = (V, E,W):

V = { 1, 2, 3, 4}

E = { 1–2, 2–3, 2–4, 3–4 }

W ={ 1,1, 2,1}

Além disso, há diversas variantes e forma de implementação. Para mais detalhes observe os vídeos e materiais abaixo.

- Vídeos sobre conceito:
    - UNESP: link.

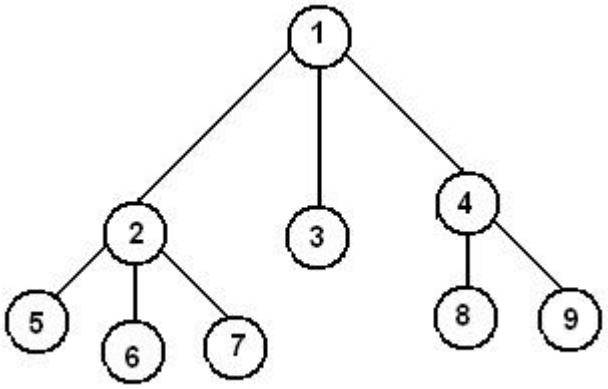
    - Canal Henrique Lima: link.
- Vídeos sobre implementação:
    - UNESP: link.
    - Canal Leandro Guarnido: link.
    - UFRGS (em C): link.
    - Canal Linguagem C descomplicada: link1 e link2.

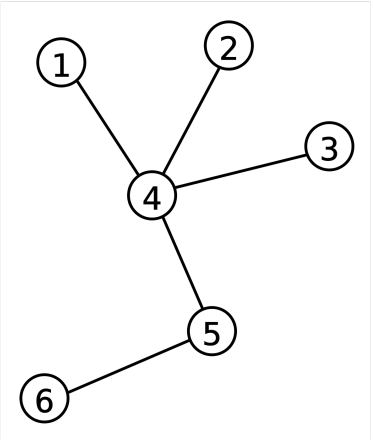
- Material grafos:
    - Slides Kleinberg e Tardos: link.
    - Slides Charles Ornelas Almeida e Nivio Ziviani:
        - Link 1
        - Link 2

Árvore

Árvore é uma estrutura de Grafo onde a estrutura expande a partir de um único nó até chegar em vértices sem relacionamentos. Um ramo não possui ligação com outro ramo.

Essa estrutura é muito usada em computação, possuindo diversos algoritmos que rodam de forma eficiente nessas estruturas. Abaixo alguns exemplos gráficos de árvores.





Para maiores detalhes assista os vídeos:

- Conceitos:
    - UNESP: link.
    - Airton Pereira: link;
- Implementação:
    - UNESP: link.
    - Árvore binária: link.
    - UFRGS (em C): link.