List

**Aplicações reais**

A Lista é uma estrutura de dados muito utilizada e podemos citar alguns sistemas ou rotinas em que elas são necessárias. Geralmente os programadores optam pela imlementação do *ArrayList* e apesar de ser bem flexível, veremos qual a mais adequada para cada rotina apresentada.

**1 – ArrayList**

**Cenário**

O sistema deve receber do Banco de Dados atributos resultado de uma busca feita em 2 tabelas onde pode haver itens repetidos e manipulá-los de acordo com os requisitos *XPTO* (hipotéticos).

Para este cenário, podemos escolher o *Arraylist* pois conseguiremos inserir elementos repetidos e acessá-los posteriormente de forma aleatoria para manipulação desses dados, seja atraves da busca pelo seu indice (metodo *get(index)*) com um custo computacional inferior se comparado ao LinkedList ou pela iteração de seus elementos (atraves da estrutura *for*, *foreach* ou da interface *Iterator*).

Para um sistema mais performático, recomenda-se ainda inicializar o *ArrayList* com o tamanho de elementos desejados, pois caso contrário, ele será iniciado com o tamanho de 10 e sempre que necessário esse tamanho aumentará em 50% do tamanho da lista. Para realizar essa operação, é criado uma nova estrutura com o tamanho novo e os elementos são copiados para esse novo *array*, elevando o custo da operação.

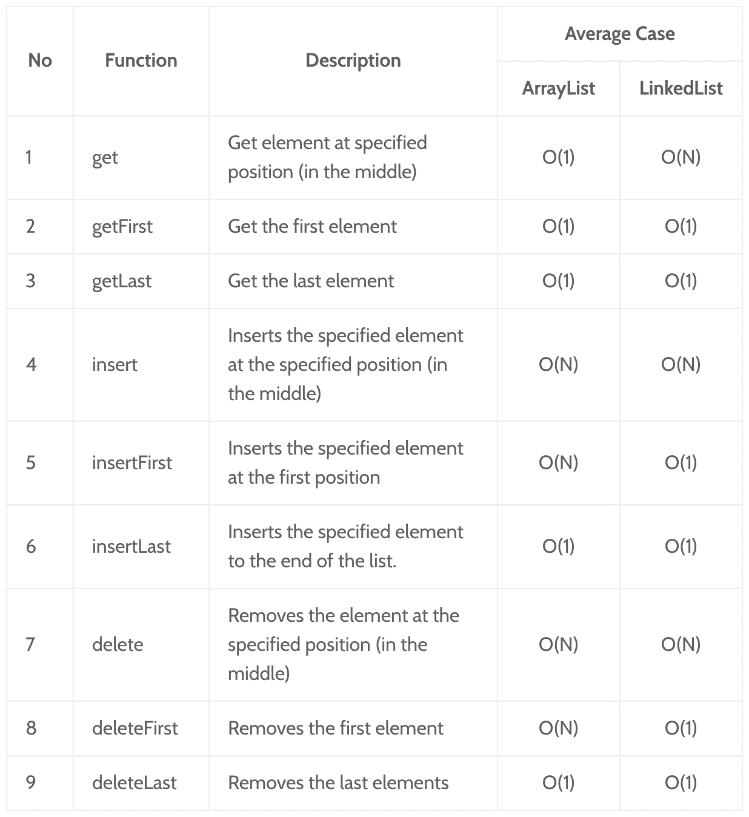
**2 – LinkedList**

**Cenário**

O sistema deverá armazenar os resltados de uma opração XPTO (hipotética) e posteriormente, disponibilizar no log da aplicação na sequencia em que foram inseridos, ou seja, o primeiro elemento armazemado, será o primeiro a ser entregue.

Para esse cenário, recomenta-se fortemente o uso da implementação *LinkedList* pois não sabemos quantos elementos serão adicionados e posteriormente serão acessados pelo topo da lista.

O *LinkedList* representa a estrutura de dados de Lista Cirlular Duplamente Encadeada, por este motivo torna-se a escolha ideal quando não há a necessidade de se acessar elementos aleatorios no meio da lista. Apesar de possuir metodos para tal ação, veremos na tabela de comparação da eficiência dos algorítimos em operações comuns a uma lista que temos uma perda de performace se comparado ao *ArrayList (O(1) – O(n)).*



**3 – Vector**

**Cenário**

O sistema deve armazenar os dados recebidos de replicas do sistema produtor XPTO (hipotético) e trata-lo conforme descrito no requisto X.

Para esse cenário teremos que lidar com mais de uma *thread* por vez acessando o codigo, adicionando e manipulando a lista que armazenará os dados vindo do sistema XPTO.

A implementação *Vector* pode ser caracterizada por ser um *ArrayList thread-safe*, o que siginifica que essa implementação é sincronizada e é capaz de trabalhar com *mult-threads*.

Apesar de ser a implementação mais adequada, sua utilização não é recomendada pois apresenta alguns pontos de falha tornando-a não tão segura para *mult-thread* dependendo do metodo invocado. Até mesmo na documentação tecnica da classe, há uma recomendação para que se dê preferência a utilização da classe *ArrayList*.

Como alternativa, podemos utilizar o metodo *syncronizedList()* da classe *Collections* ou a classe *CopyOnWriteArrayList().*

**4 – Comparação**

Se compararmos a performace principalmente do ArrayList com o LinkedList em ações básicas como adicionar, buscar e remover, veremos que o LinkedList é mais performático para adicionar e remover (na ordem sequencial), porem é muito lento quando se trata de buscar um elemento pelo indice, pois ele sempre iniciará a busca pelo primeiro item, percorrendo todos os elementos para chegar ao indice desejado, enquanto o ArrayList, acessa o elemento diretamente pelo seu indice.

Segue um codigo em que é testado essa diferença de performace e um gráfico com os resultados obtidos:

public class ApplicationMain {  
 public static void main(String[] args) {  
 List<Integer> arrayList = new ArrayList<>();  
 List<Integer> linkedList = new LinkedList<>();  
 List<Integer> vector = new Vector<>();  
  
 System.*out*.println(" TESTANDO A PERFORMACA DAS 3 IMPLEMENTACOES DE LIST");  
 System.*out*.println();  
 System.*out*.println();  
 System.*out*.println(" ---> Tempo gasto para ADICIONAR 100.000 itens:");  
  
 long startTime = 0;  
 long endTime = 0;  
 long duration = 0;  
  
// LinkedList add  
 startTime = System.*nanoTime*();  
  
 for (int i = 0; i < 100000; i++) {  
 linkedList.add(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - LinkedList (add)");  
  
// ArrayList add  
 startTime = System.*nanoTime*();  
  
 for (int i = 0; i < 100000; i++) {  
 arrayList.add(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - ArrayList (add)");  
  
// Vector add  
 startTime = System.*nanoTime*();  
  
 for (int i = 0; i < 100000; i++) {  
 vector.add(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - Vector (add)");  
  
 System.*out*.println();  
 System.*out*.println(" ---> Tempo gasto para BUSCAR os itens da lista (buscará todos, um por iteração):");  
  
// LinkedList get  
 startTime = System.*nanoTime*();  
  
 for (int i = 0; i < 10000; i++) {  
 linkedList.get(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - LinkedList (get)");  
  
// ArrayList get  
 startTime = System.*nanoTime*();  
  
 for (int i = 0; i < 10000; i++) {  
 arrayList.get(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - ArrayList (get)");  
  
// Vector get  
 startTime = System.*nanoTime*();  
  
 for (int i = 0; i < 10000; i++) {  
 vector.get(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - Vector (get)");  
  
 System.*out*.println();  
 System.*out*.println(" ---> Tempo gasto para REMOVER os itens da lista (excluira todos, um por iteração):");  
  
// LinkedList remove  
 startTime = System.*nanoTime*();  
  
 for (int i = 9999; i >=0; i--) {  
 linkedList.remove(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - LinkedList (remove)");  
  
// ArrayList remove  
 startTime = System.*nanoTime*();  
  
 for (int i = 9999; i >=0; i--) {  
 arrayList.remove(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - ArrayList (remove)");  
  
// Vector remove  
 startTime = System.*nanoTime*();  
  
 for (int i = 9999; i >=0; i--) {  
 vector.remove(i);  
 }  
 endTime = System.*nanoTime*();  
 duration = endTime - startTime;  
 System.*out*.println(duration + " - Vector (remove)");  
 }  
}

Saída console:

