45EST – Algoritmos e Estruturas de Dados

Busca em estruturas lineares

Prof. Marcelo de Souza

UDESC Ibirama
Bacharelado em Engenharia de Software

marcelo.desouza@udesc.br
Versão compilada em 13 de agosto de 2020

Leitura obrigatória:

- Capítulo 5 de Ziviani [2010] – Pesquisa em memória binária.

Leitura complementar:

- Capítulo 13 de Pereira [2008] – Ordenação e busca.

Algoritmos de busca

- Buscar um elemento consiste em verificar se o mesmo está armazenado em uma estrutura de dados.
- O retorno pode ser o próprio elemento, o índice onde ele se encontra, ou um valor booleano de sucesso.
- Busca em um vetor simples:
  - Dado um vetor e um valor, verifica se o valor está armazenado no vetor.
- Busca em uma lista (vetor ou encadeada) genérica:
  - Dada uma lista genérica e um valor genérico, verifica se o valor está armazenado na lista.
- Busca em uma estrutura de entradas (chave, valor):
  - Dada uma lista genérica de entradas e uma chave, devolve o valor correspondente, ou `null` caso a chave não seja encontrada.

Busca sequencial

- É a forma mais simples de busca.
- Percorre a estrutura até encontrar o elemento.
- Complexidade linear $O(n)$.

Busca sequencial simples:

```java
public class SimpleSequentialSearch {

   public int search(int[] array, int value) {
      for(int i = 0; i < array.length; i++)
      if(array[i] == value)
         return i;
      return -1;
   }

   public int search(String[] array, String value) {
      for(int i = 0; i < array.length; i++)
      if(array[i].equals(value))
         return i;
      return -1;
   }
}
```

Busca sequencial genérica:

```java
public class GenericSequentialSearch<E> {

   Comparator<E> comp;

   public GenericSequentialSearch() {
      this(new DefaultComparator<E>());
   }

   public GenericSequentialSearch(Comparator<E> c) {
```

```
10        comp = c;
11     }
12
13     public int indexOf(E[] array, E value) {
14        for(int i = 0; i < array.length; i++)
15        if(comp.compare(array[i], value) == 0)
16           return i;
17        return -1;
18     }
19
20     public int indexOf(List<E> array, E value) {
21        for(int i = 0; i < array.size(); i++)
22        if(comp.compare(array.get(i), value) == 0)
23           return i;
24        return -1;
25     }
26  }
```

Comentários:

- Classe usa um comparador para identificar o elemento buscado.
  1. Definir um comparador próprio; ou
  2. Definir a classe comparável e implementar o método `compareTo`.

Busca sequencial genérica para entradas:

```
1  public class GenericSequentialEntrySearch<K, V> {
2
3     Comparator<K> comp;
4
5     public GenericSequentialEntrySearch() {
6        this(new DefaultComparator<K>());
7     }
8
9     public GenericSequentialEntrySearch(Comparator<K> c) {
10       comp = c;
```

```
11      }
12
13      public V searchElement(List<Entry<K,V>> array, K key) {
14          for(int i = 0; i < array.size(); i++)
15              if(comp.compare(array.get(i).getKey(), key) == 0)
16                  return array.get(i).getValue();
17          return null;
18      }
19  }
```

Comentários:

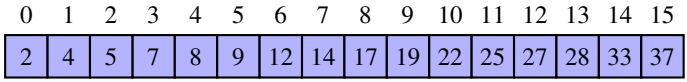
- Novamente, a classe usa um comparador para encontrar a chave.

Busca binária

- Busca mais eficaz, executada em $O(\log n)$.
- Pré-condições:
  - Necessita acesso aleatório à estrutura (vetores).
  - Dados devem estar ordenados.
- Funcionamento:
  1. Avalia o elemento central da lista.
  2. Caso seja o elemento buscado – sucesso.
  3. Caso contrário, avalia em qual sub-lista se o elemento pode estar.
  4. Repete a busca com a sub-lista correspondente.

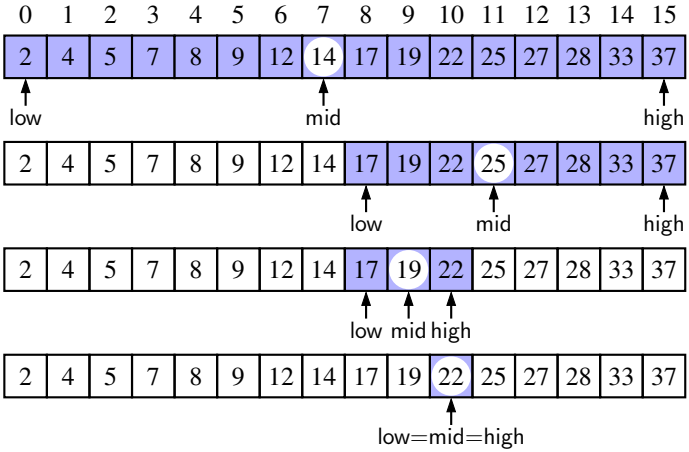
Exemplo

Vetor (já ordenado):



Elemento buscado: 22.

Funcionamento:



Conclusão:

- Encontra o elemento com 4 avaliações (14, 25, 19 e 22).
- Caso o elemento buscado fosse 23, na próxima iteração identificaria sua inexistência, pois `high < low`.

Busca binária simples:

```
1  public class SimpleBinarySearch {
2     public int search(int[] array, int value) {
3        int start = 0;
4        int end = array.length - 1;
5        int mid;
6
7        do {
8           mid = (start + end) / 2;
9           if(array[mid] < value)
10             start = mid + 1;
11          else
12             end = mid - 1;
13       } while(array[mid] != value && start <= end);
14
15       if(array[mid] == value)
16          return mid;
17       else
18          return -1;
19    }
20
21    public int search(String[] array, String value) {
22       int start = 0;
23       int end = array.length - 1;
24       int mid;
25
26       do {
27          mid = (start + end) / 2;
28          if(array[mid].compareTo(value) < 0)
29             start = mid + 1;
30          else
31             end = mid - 1;
32       } while(array[mid].compareTo(value) != 0 && start <= end);
33
34       if(array[mid].compareTo(value) == 0)
35          return mid;
36       else
```

```
37            return -1;
38        }
39    }
```

Comentários:

- Veriáveis `start` e `end` delimitam a sub-lista de busca (equivalente a `low` e `high`).
- Enquanto não encontra, atualiza o intervalo de índices e repete o processo.
- Quando `start > end`, finaliza a busca sem sucesso.

Busca binária genérica:

```
1  public class GenericBinarySearch<E> {
2
3     Comparator<E> comp;
4
5     public GenericBinarySearch() {
6        this(new DefaultComparator<E>());
7     }
8
9     public GenericBinarySearch(Comparator<E> c) {
10       comp = c;
11    }
12
13    public int indexOf(E[] array, E value) {
14       int start = 0;
15       int end = array.length - 1;
16       int mid;
17
18       do {
19          mid = (start + end) / 2;
20          if(comp.compare(array[mid], value) < 0)
21             start = mid + 1;
22          else
```

```
            end = mid - 1;
        } while(comp.compare(array[mid], value) != 0 && start <= end);

        if(comp.compare(array[mid], value) == 0)
            return mid;
        else
            return -1;
    }

    public int indexOf(List<E> array, E value) {
        int start = 0;
        int end = array.size() - 1;
        int mid;

        do {
            mid = (start + end) / 2;
            if(comp.compare(array.get(mid), value) < 0)
                start = mid + 1;
            else
                end = mid - 1;
        } while(comp.compare(array.get(mid), value) != 0 && start <=
        ↪  end);

        if(comp.compare(array.get(mid), value) == 0)
            return mid;
        else
            return -1;
    }
}
```

Busca binária genérica para entradas:

```
public class GenericBinaryEntrySearch<K,V> {

    Comparator<K> comp;

    public GenericBinaryEntrySearch() {
        this(new DefaultComparator<K>());
```

```
7       }
8
9       public GenericBinaryEntrySearch(Comparator<K> c) {
10          comp = c;
11      }
12
13      public V searchElement(List<Entry<K,V>> array, K key) {
14          int start = 0;
15          int end = array.size() - 1;
16          int mid;
17
18          do {
19              mid = (start + end) / 2;
20              if(comp.compare(array.get(mid).getKey(), key) < 0)
21                  start = mid + 1;
22              else
23                  end = mid - 1;
24          } while(comp.compare(array.get(mid).getKey(), key) != 0 &&
            ↪  start <= end);
25
26          if(comp.compare(array.get(mid).getKey(), key) == 0)
27              return array.get(mid).getValue();
28          else
29              return null;
30      }
31  }
```

Comentários:

- Veja as aplicações das buscas em diferentes estruturas nas classes `TestSequentialSearch` e `TestBinarySearch`.

Atividades

1. Desenvolva um software para gerenciar as contas de um banco. Primeiro, armazene os dados em uma estrutura de dados não-ordenada e utilize o algoritmo de busca sequencial para buscar contas. Após isso, implemente uma versão utilizando uma estrutura de dados ordenada, e aplique o algoritmo de busca binária para a pesquisa. Compare o tempo de execução de ambas abordagens para diferentes quantidades de contas armazenadas. Verifique a complexidade empírica dos algoritmos e compare com a complexidade no pior caso.

2. Implemente uma versão recursiva do algoritmo de busca binária. Em caso de dúvidas, consulte a Seção 5.1.3 de Goodrich et al. [2014].

3. Simule o algoritmo de busca binária para os seguintes casos:

   a) `x = 15`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

   b) `x = 33`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

   c) `x = 63`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

   d) `x = 81`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

   e) `x = 22`, `v = {15, 27, 33, 46, 51, 63, 71, 82, 90}`.

   Compare o número de avaliações realizadas com o número de avaliações que uma busca sequencial faria.

4. Quando o vetor está ordenado, a busca sequencial não precisa percorrer toda a lista para saber que o elemento buscado não existe. Ela pode parar quando o elemento analisado for maior que o buscado. Implemente as modificações necessárias para esta estratégia. Qual a complexidade no pior caso do novo algoritmo?

5. Implemente os algoritmos de busca sequencial e binária nas estruturas de dados estudadas em sala de aula.

6. Veja as demonstrações de buscas sequencial e binária disponíveis em https://www.cs.usfca.edu/~galles/visualization/Search.html.

Referências

Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). *Data structures and algorithms in Java*. John Wiley & Sons, 6th edition.

Pereira, S. d. L. (2008). Estruturas de dados fundamentais: Conceitos e aplicações.

Ziviani, N. (2010). *Projeto de Algoritmos com Implementa'ções em Java e C++*. Cengage Learning.