Lista de Exercícios - CAP-241 2018

Dr. Gilberto Ribeiro de Queiroz
gilberto.queiroz@inpe.br

Instituto Nacional de Pesquisas Espaciais - INPE
São José dos Campos, SP, Brasil

15 de maio de 2018

Capítulo 1

Estruturas de Dados Elementares

Atenção:

1. Os exercícios práticos deste capítulo devem ser desenvolvidos em C++ padrão. Escreva a documentação que achar pertinente dentro do próprio código fonte, que deverá utilizar a codificação de caracteres UTF-8. Opcionalmente, você poderá incluir um documento no formato PDF com explicações e figuras que achar necessário. Mas não envie o PDF com o código desenvolvido.

2. O texto com a discussão sobre os exercícios teóricos deverão ser entregues no formato PDF.

3. Envie por e-mail **um único** arquivo no **formato zip**, chamado `lista.zip`, contendo o código fonte dos exercícios e o relatório da parte teórica.

4. O título do e-mail deve seguir o seguinte padrão[1]:
   `[cap241-2018][lista] nome-aluno`.

5. Seja objetivo nos exercícios.

6. **Prazo para entrega:** 01/06/2018 - 17:00

[1]Não use acentos ou caracteres especiais no nome do arquivo.

1.1 Exercícios

Exercício 1. Projete e implemente uma estrutura de dados do **tipo árvore** cujos nós possam conter um número qualquer de descendentes (árvore multivias). Os elementos de dados desta árvore poderão ser de qualquer tipo de dado primitivo de C++ ou de estruturas/classes definidas pelo usuário que atendam aos requisitos que você definir.

Apesar de parecer teórico este exercício, esse tipo de estrutura possui diversas aplicações. Por exemplo, todos os sistemas operacionais modernos possuem algum tipo de aplicativo para mostrar de forma gráfica o sistema de arquivos, que possui uma estrutura hierárquica. A estrutura de um documento XML é outro exemplo prático de árvore, que começa com um elemento raiz podendo conter vários nós descendentes, que podem representar novos elementos, atributos e textos. Em um Sistema de Informação Geográficas (SIG) as camadas de informação (ou planos de informação) podem ser organizadas através de uma árvore.

Exercício 2. Projete e implemente uma **árvore binária de pesquisa** cujos elementos de dados possam ser qualquer tipo de dados primitivo de C++. Essa árvore deverá suportar as operações de busca, inserção e remoção.

Exercício 3. Implemente as seguintes operações de travessia na árvore do *exercício 2*:

- $pre\_order(tree, functor) \rightarrow tree$: aplica uma determinada função nos elementos da árvore durante uma travessia do tipo *pré-ordem*.
- $in\_order(tree, functor) \rightarrow tree$: aplica uma função nos elementos da árvore durante uma travessia do tipo *ordem central*.
- $pos\_order(tree, functor) \rightarrow tree$: aplica uma função nos elementos da árvore durante uma travessia do tipo *pós-ordem*.

Exercício 4. Aumente o conjunto de operações da árvore do *exercício 2* com a seguinte operação:

- $height(tree) \rightarrow int$: operação que fornece a altura da árvore.

Exercício 5. Escreva um programa que ajude a verificar se a inserção de valores de forma aleatória em uma árvore binária de pesquisa produz uma árvore cuja altura seja proporcional a $O(log_2 n)$. Para isso, utilize um gerador de números aleatórios com distribuição uniforme para realizar a inserção de elementos na árvore do *exercício 2*. Use uma sequência de tamanho $n = (10^1, 10^2, ..., 10^6)$. Apresente um gráfico com a razão $altura/log_2 n$ em função do tamanho n. Analise seu resultado.

Exercício 6. Suponha que você tenha um conjunto de registros de empregados de uma grande empresa, onde cada registro é formado por diversos atributos, entre eles, salário e ano de nascimento. Agora, suponha que você esteja trabalhando em um sistema de informação que lide com esse tipo de dado, tendo que atender a consultas como:

> *"Encontrar todos os empregados nascidos entre 1950 e 1955 que recebem entre R$3.000,00 e R$5.000,00."*

Qual estrutura de dados você poderia utilizar para responder a este tipo de consulta?

Discutimos diversas estruturas de dados no curso, mas nenhuma delas seria a mais adequada a esse problema. Se armazenarmos esses registros em um vetor ou em uma lista encadeada, teremos que varrer toda a sequência de elementos para recuperar os registros de interesse, ou seja, cada consulta será respondida em tempo $O(n)$, onde n é o número de registros de funcionários. Organizar esses registros com a ajuda de uma árvore binária de pesquisa para a finalidade da consulta acima também não seria viável, pois a busca não será por um valor de chave específico e sim por intervalos ao longo de dois conjuntos de valores distintos: ano nascimento e salário.

Dessa forma, estamos interessados em uma estrutura de dados que nos ajude a responder consultas de intervalos. Se pensarmos que cada atributo corresponde a uma dimensão do dado, é como se tivéssemos os registros dos funcionários espalhados num plano cartesiano como o mostrado na Figura 1.1. Nessa figura, um eixo corresponde ao salário e o outro, à idade. Os registros de nosso interesse correspondem à área retangular destacada em laranja nessa figura. Logo, dados que possuem múltiplas chaves ou partes, podem ser vistos como pontos num espaço multidimensional.

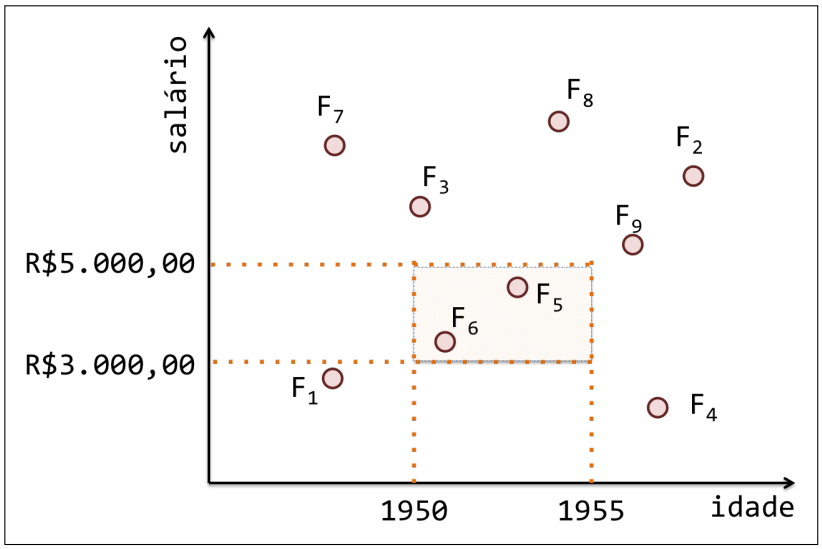

Figura 1.1: Conjunto de registros de funcionários visto em um plano cartesiano.

Uma estrutura de dados que foi proposta para problemas desse tipo é a *k-d-tree*, apresentada por Bentley em 1975[2]. Uma *k-d-tree* é uma árvore binária, ou seja, cada nó interno possui dois descendentes, não importando a dimensionalidade k do espaço envolvido. Esta árvore foi proposta como uma alternativa a solução de um problema mais genérico do que o geométrico, que é o de *busca baseada em chaves com vários atributos*.

No caso de conjuntos de pontos no espaço bidimensional, podemos usar esta estrutura para ajudar a responder consultas de retângulo ou janela, também conhecidos como *range query* ou *window query*.

Uma *k-d-tree* no espaço bidimensional pode ser construída facilmente a partir de um conjunto de pontos. Em cada nível da árvore, os nós descendentes são divididos ao longo de uma das dimensões do dado, de forma perpendicular ao eixo correspondente. Por exemplo, considere o conjunto de pontos mostrado na Figura 1.2. Veja que o valor de abscissa do ponto P_5, isto é, $P_5.x$, divide o conjunto de pontos em dois subconjuntos, um a esquerda de $P_5.x$ e outro a direita da abscissa desse ponto: $S_{esquerda} = \{P_1, P_3, P_6, P_7\}$ e $S_{direita}\{P_2, P_4, P_8, P_9\}$. Para o subconjunto $S_{esquerda}$, a ordenada do ponto P_3 ($P_3.y$) divide seus elementos em outros dois subconjuntos, um abaixo desse valor de ordenada e outro acima: $S_{abaixo} = \{P_1, P_6\}$ e $S_{acima} = \{P_7\}$.

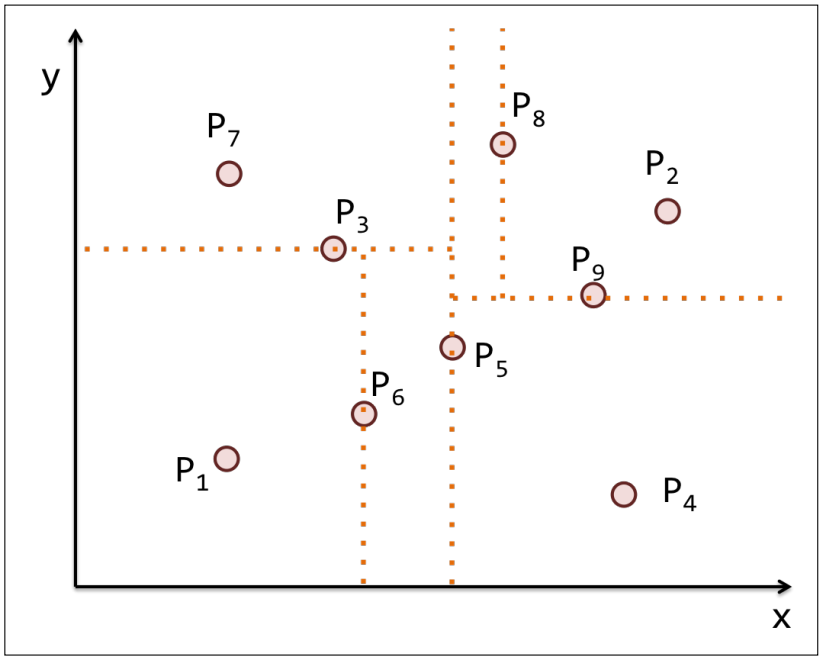

Figura 1.2: Conjunto de pontos no espaço bidimensional.

[2]Bentley, J. L. **Multidimensional binary search trees used for associative searching**. Communications of the ACM, v. 18, n. 9, p. 509-517, 1975.

Para construir a *k-d-tree* mostrada na Figura 1.3 a partir da sequência de pontos: $(P_5, P_9, P_3, P_6, P_1, P_7, P_4, P_8, P_2)$, procedemos da seguinte forma:

1. O primeiro ponto a ser inserido, no caso P_5, será definido como nó raiz dessa árvore. Todos os demais pontos da sequência serão inseridos como elementos descendentes da raiz, sendo divididos com base no valor da abscissa (x), que é a primeira dimensão considerada. Assim, todos os demais pontos da sequência com valores de x menor do que $P_5.x$, serão inseridos na sub-árvore esquerda, enquanto os pontos com valor de x maior do que $P_5.x$, serão inseridos na sub-árvore direita.

2. Seguindo a *regra do item 1* acima, o próximo da sequência, o ponto P_9, será inserido à direita do nó P_5, uma vez que seu valor de abscissa é maior do que a de P_5.

3. De manera similar, o ponto P_3, será inserido à esquerda do nó P_5, uma vez que seu valor de abscissa é menor do que a de P_5.

4. O ponto P_6 possui um valor de abscissa menor do que a do ponto P_5 e, portanto, será inserido na sub-árvore esquerda do nó definido pelo ponto P_5. No nível seguinte, a comparação se dará pela outra componente, a das ordenadas (y). Neste caso, o ponto P_6 possui um valor de ordenada menor do que a de P_3, logo, o ponto P_6 será inserido na sub-árvore esquerda de P_3.

Esse processo será aplicado para todos os pontos. Observe que em cada nível da árvore usaremos uma das componentes do dado para dividir os elementos, retomando a primeira dimensão assim que todas as outras tenham sido consideradas.

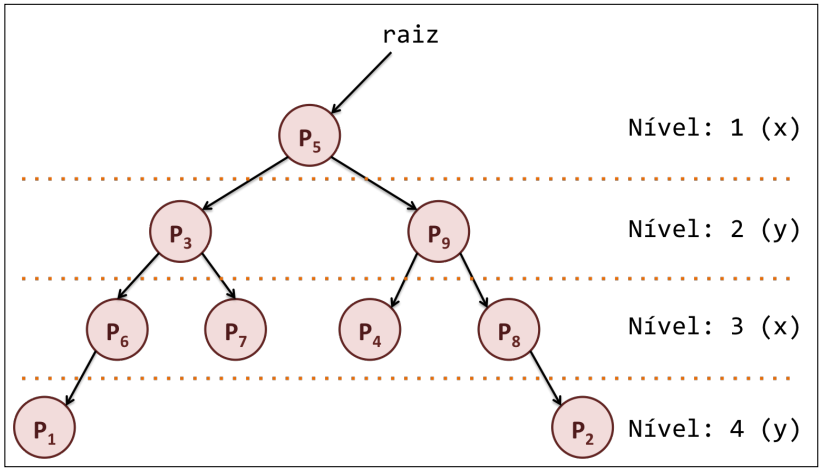

Figura 1.3: *k-d-tree* para os pontos da Figura 1.2.

A pesquisa por um valor exato de chave, isto é, pelos valores de x e y de um ponto, pode ser realizada da seguinte forma:

- Suponha que você esteja procurando por um valor de chave, com valores de x_p e y_p iguais aos do ponto P_2.
- Começando pelo nó raiz, comparamos o valor de x_p e $P_5.x$. Como o valor de x_p é maior do que $P_5.x$, a busca continua na sub-árvore direita.
- No segundo nível, comparamos os valores de ordenada, y_p e $P_9.y$. Como o valor de y_p é maior do que $P_9.y$, a busca continua na sub-árvore direita.
- No terceiro nível, usaremos novamente os valores de abscissa, x_p e $P_8.x$. Como o valor de x_p é maior do que $P_8.x$, a busca continua na sub-árvore direita.
- No quarto nível, usaremos novamente os valores de ordenada, y_p e $P_2.y$. Como esses valores são iguais, comparamos a próxima componente, x_p e $P_2.x$, verificando que as chaves são iguais e, consequentemente, reportando esse ponto.

A pesquisa por um retângulo de busca pode ser realizada da seguinte forma:

1. Suponha o retângulo de busca $(x_{min}, x_{max}, y_{min}, y_{max})$, mostrado na Figura 1.4.

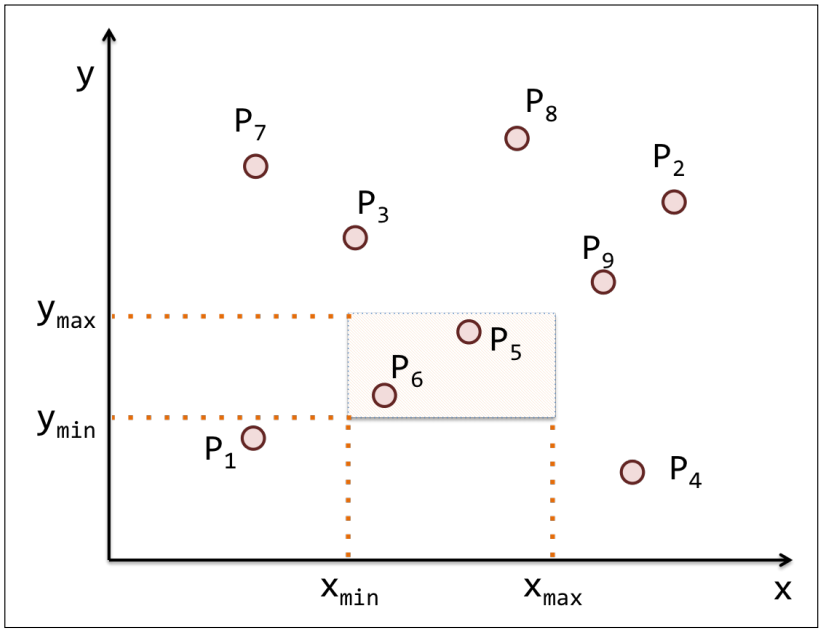

Figura 1.4: Busca por intervalo sobre a *k-d-tree* da Figura 1.3.

2. Na Figura 1.3, começando pelo nó raiz, verificamos se a chave x desse nó encontra-se no intervalo $[x_{min}, x_{max}]$. Se o intervalo estiver completamente a esquerda, toda a sub-árvore direita pode ser desprezada da busca pois apenas elementos à esquerda desse nó estarão no intervalo desejado. De maneira similar, se o intervalo estiver completamente a direita, toda a sub-árvore esquerda pode ser desprezada da busca. Quando o intervalo contém o valor do nó, temos que verificar os demais intervalos e componentes do nó. Nesse caso o valor de $P_5.y$ encontra-se no intervalo $[y_{min}, y_{max}]$, sendo esse ponto reportado. Além disso, nesse caso a busca continuará dos dois lados da árvore, o que claramente nos arremete a um algoritmo de busca recursivo.

3. No segundo nível da árvore, $P_3.y$ encontra-se acima do intervalo $[y_{min}, y_{max}]$, de forma que os nós da sub-árvore direita ($\{P_7\}$), são descartados da busca. No entanto, a sub-árvore esquerda precisa ser examinada.

4. No terceiro nível, o ponto P_6 encontra-se dentro do intervalo procurado e deve ser reportado. As sub-árvores de P_6 também deverão ser examinadas pois $P_6.x$ enconra-se no intervalo $[x_{min}, x_{max}]$.

5. Retomando a busca do lado direito do nó P_5, o intervalo $[y_{min}, y_{max}]$ encontra-se abaixo do valor de $P_9.y$, de forma que a sub-árvore direita não precisará ser verificada, mas a sub-árvore esquerda precisará.

6. Como o nó P_4 está fora do intervalo e não possui descendentes, encerramos a busca.

Para este exercício, você deverá implementar uma *k-d-tree* para o espaço bidimensional, considerando as seguintes operações:

- $new() \rightarrow kdtree$: cria uma árvore do tipo kd-tree vazia.
- $insert(kdtree, point) \rightarrow kdtree$: insere um ponto no espaço bidimensional na kd-tree e retorna o elemento da árvore no qual o ponto foi inserido.
- $find(kdtree, point) \rightarrow kdtree$: retorna o nó da árvore que contém o ponto procurado.
- $search(kdtree, rectangle) \rightarrow [kdtree]$: retorna o conjunto de todos os nós cujos pontos estão contidos no retângulo de busca.
- $clear(kdtree) \rightarrow kdtree$: libera todos os recursos.

Exercício 7. Monte uma bateria de testes e compare os seguintes algoritmos de ordenação: *Ordenação por seleção*, *Ordenação por inserção*, *Heapsort*, *Mergesort*, *Quicksort*. Utilize vetores com números gerados de forma aleatória. Também rode os testes para vetores completamente ordenados de forma crescente e decrescente. Tome cuidado em suas análises, obtendo o tempo médio de um grande número de execuções. Faça um relatório

explicando a metodologia detalhada dos testes e os resultados obtidos. Você deverá entregar o código fonte da sua bateria de testes incluindo os algoritmos de ordenação utilizados, mesmo que não tenham sido modificados. Tente propor modificações nos códigos para torná-los mais eficientes.