Fundação Getulio Vargas
Estruturas de Dados
Prof. Jorge Poco

Lista 2/João Victor Mendes
Entrega: Março 20, 2022

Lista 2

Parte 1 - Questões Teóricas

Problema 1. (12 pontos)

1.1. (6 pontos) Considere a árvore enraizada da Figura 5(a). Desenhe uma figura mostrando sua representação na forma "primeiro filho/próximo irmão".

1.2. (6 pontos) Considere a árvore enraizada da Figura 5(b) representada na forma "primeiro filho/próximo irmão". Desenhe uma figura mostrando a árvore enraizada equivalente.

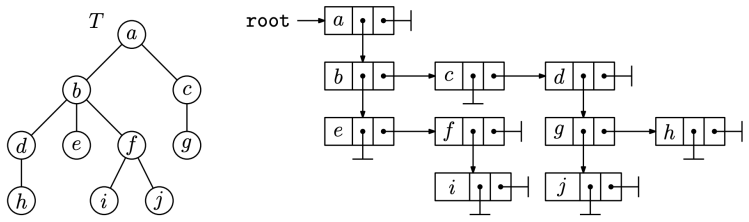

Figura 1: Árvore enraízada a partir de árvore "primeiro-filho/próximo irmão"e vice-versa.

Resposta 1:

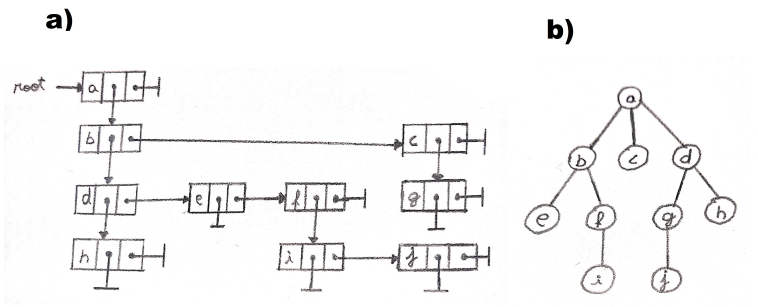

Figura 2: "primeiro filho/próximo irmão"da letra (a) e árvore enraizada da letra (b).

Problema 2. (8 pontos) Desenhe a árvore binária da Figura 3(a) com os fios da ordem (*inorder-threads*) adicionados.

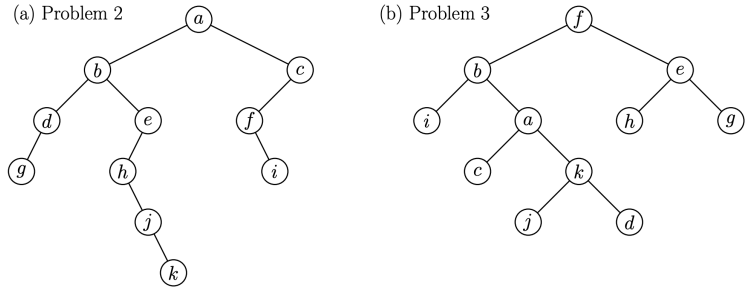

Figura 3: Adicionando *inorder threads* para uma árvore binária e uma árvore binária completa.

Resposta 2:

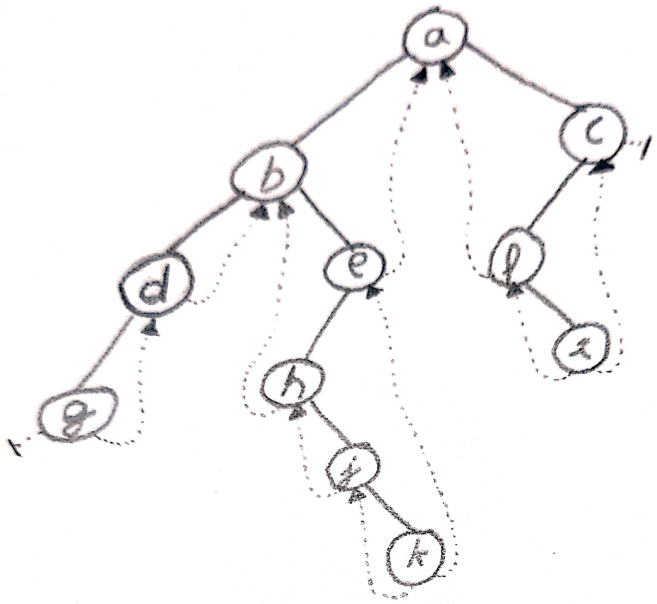

Figura 4: (a) da figura 3 com *inorder threads* adicionados.

Problema 3 (15 pontos) Você tem uma árvore binária completa, onde cada nó é rotulado com uma letra. (Lembre-se de que uma árvore binária está cheia se cada nó não-folha tiver exatamente dois filhos.) Ao longo deste problema, restringimos a atenção às árvores binárias completas.

3.1. (3 pontos) Alguém executou uma travessia *postorder* e forneceu uma lista com os nomes dos nós. (Por exemplo, na árvore mostrada na Figura 3(b), isso é $(i, c, j, d, k, a, b, h, g, e, f)$. É possível recuperar a estrutura da árvore binária completa exclusivamente da sequência *postorder*? Se sim, explique como apresentando um algoritmo para fazê-lo. Se não, desenhe duas árvores binárias completas rotuladas onde as listas de *postorder* são as mesmas.

3.2. (3 pontos) Repita (3.1), mas dessa vez a lista foi modificada para que cada o nó folha foi "marcado" para distinguir as folhas dos nós internos. (Por exemplo, na árvore mostrada na Fig. 3(b), se usarmos "$\star$"para indicar uma folha, isso seria $(i*, c*, j*, d*, k, a, b, h*, g*, e, f)$.

3.3. (3 pontos) Repita (3.1), mas desta vez para um temos a lista do percurso *inorder* de uma árvore binária completa. (Para exemplo, na árvore mostrada na Figura 3(b), isso seria $(i, b, c, a, j, k, d, f, h, e, g)$.

3.4. (4 pontos) Repita (3.2), mas desta vez para um temos a lista do percurso *inorder* de uma árvore binária completa. (Para exemplo, na árvore mostrada na Figura 3(b), isso seria $(i*, b, c*, a, j*, k, d*, f, h*, e, g*)$.

3.5. (4 pontos) Não pediremos que você resolva o caso restante (com uma sequência de um percurso *pre-order*), mas suponha que você discuta o caso (3.1) com seu melhor amigo (sequência *preorder* sem os nós interno marcados). (Vocês dois suspeitam que o malvado professor pode colocar essa questão em um exame futuro.) Este amigo anuncia que a resposta é "não" e informa que existe um contra-exemplo simples de 6 nós. Sem sequer ver o contra-exemplo, você diz ao seu amigo que isso está errado! Como é isso possível? (Assuma para este problema que você não é um médium.)

Resposta 3:

3.1. Não é possível recuperar a estrutura da árvore binária completa exclusivamente da sequência *postorder*, contra exemplo na (a) da próxima figura;

3.2.

3.3. Pela resposta da (3.4), se um percurso inorder marcado é ambíguo, então um percurso não marcado também é.

3.4. Não é possível recuperar uma árvore somente pelo seu percurso inorder, mesmo com as folhas marcadas. Veja o contra exemplo na (b) da próxima figura.

3.5. Sabemos que uma árvore binária completa com 'n' nós internos tem n + 1 nós externos (folhas), o que faz o número de nós em uma árvore binária completa ser sempre ímpar. Assim, o contra-exemplo de contagem de 6-nós de seu amigo não pode ser uma árvore binária completa válida.

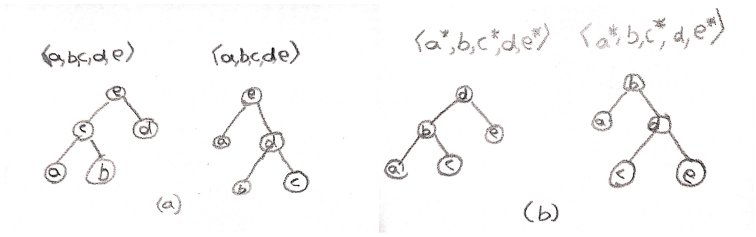

Figura 5: Recuperando árvore binária completa.

Problema 4 (15 pontos) São dadas duas matrizes $n \times n$ A e B, onde (seguindo a convenção de C++) as linhas e colunas são indexadas de 0 a $n-1$. Seu produto $A \cdot B$ é uma matriz $n \times n$ C, onde para $0 \leq i$, $j \leq n-1$, $C[i,j] = \sum_{k=0}^{n-1} A[i,k] \cdot B[k,j]$.

4.1. (5 pontos) Assuma que A e B são representadas por matrizes esparsas (ver Lecture 4 e Figura 6). Apresente um algoritmo eficiente para o cálculo do produto $A \cdot B$.
Para simplificar, você pode assumir que a matriz de saída C é representada como uma matriz bidimensional $n \times n$, que foi inicializada com zeros. Para tornar possível a generalize sua solução para o caso esparso, você deve preencher as entradas não zero de C em ordem seqüencial (por exemplo, de cima para baixo e da esquerda para a direita).

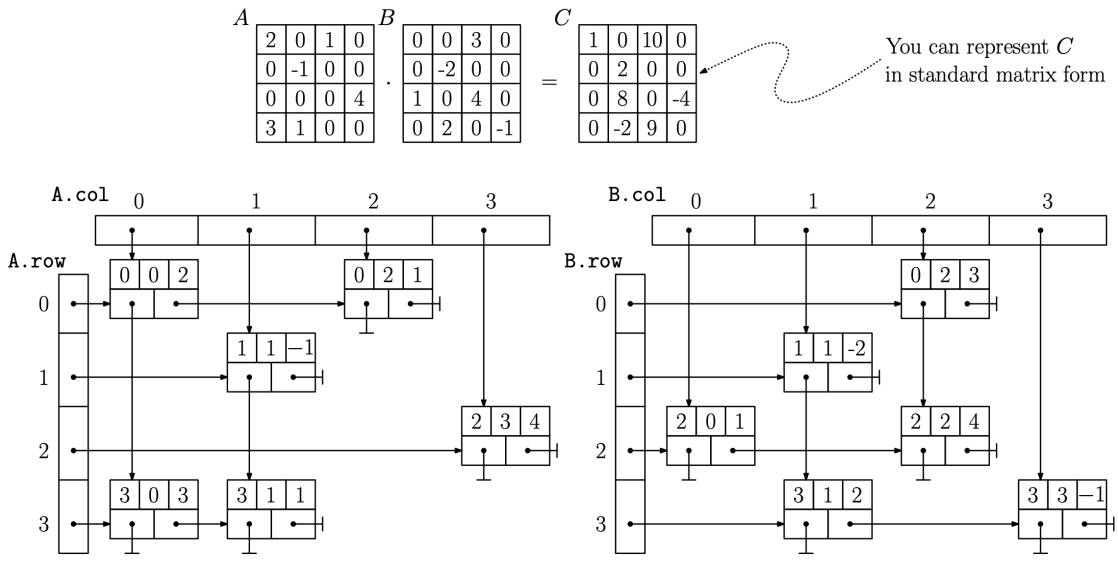

Figura 6: Multiplicação de matriz esparsa.

4.2. (3 pontos) Dê o tempo de execução de seu algoritmo em termos das seguintes quantidades: n, N_A e N_B, onde N_A e N_B são os números de entradas não zeradas nas matrizes A e B, respectivamente. (Ou seja, declare qual é o tempo de execução assimptótico e apresente uma prova ou explicação convincente de sua limitação. **Dica**: No caso especial quando as matrizes são densas, ou seja, $N_A = N_B = n^2$, o tempo de execução deve ser $O(n^3)$).

Resposta 4:

Parte 2 - Tarefa de programação

Quake Heaps: Este é a primeira tarefa (dividida em mais partes) para implementar uma estrutura de dados interessante chamada de *Quake Heap*. Tal como acontece com *heaps* padrão, esta estrutura de dados implementa uma prioridade fila. Tal estrutura de dados armazena pares chave-valor, onde as chaves são de um tipo ordenável (como `ints`, `floats` ou `strings`). No mínimo, uma fila de prioridade suporta o operações de *insert* (adicionar um novo par chave-valor) e *extrair-min* (remover a entrada com o menor chave e retornar seu valor associado).

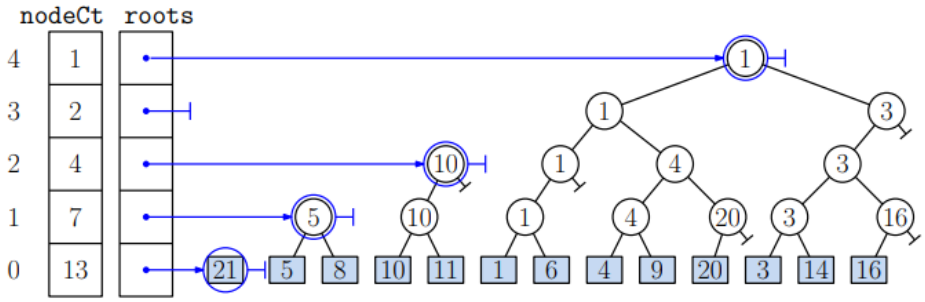

Figura 7: Uma *Quake Heap* armazenando as chaves $(21, 5, 8, 10, 11, 1, 6, 4, 9, 20, 3, 14, 16)$.

O exemplo mais famoso de heap é a b*inary heap*, que é a estrutura de dados usada por *HeapSort*. Existem inúmeras variantes, que fornecem desempenho aprimorado para vários operações, notadamente a de diminuir uma chave. A *Quake Heap* é uma dessas variantes. Foi desenvolvido por Timothy Chan (descrito neste artigo) como uma alternativa mais simples para a *Fibonacci Heap*. Suporta inserção e chaves decrescentes em tempo $O(1)$, e suporta *extract-min* em $O(logn)$ tempo amortizado. Na primeira parte da tarefa, implementaremos apenas uma parte das funcionalidades do *Quake Heap*.
A classe `QuakeHeap` funcionará com tipos genéricos, um tipo de chave e um tipo de valor. O tipo de chave deve ser ordenável, e assumimos que ambos podem ser convertidos para strings com `to_string`. O `QuakeHeap` é representado como uma coleção de árvores binárias, onde cada nó armazena um par de valores-chave. Os nós dessas árvores são organizados em níveis. Todas as folhas residem no nível 0, e cada chave na *heap* é armazenada em exatamente uma folha de alguma árvore. (Por exemplo, na Figura 7, as chaves consistem nas 13 chaves nos nós de folha azul.)
Cada nó interno tem um filho esquerdo e um filho direito opcional. O valor da sua chave é igual ao do seu filho da esquerda. Se o filho direito existir, seu valor de chave será maior ou igual ao filho esquerdo. Cada raíz contém o menor valor de chave sobre todas as folhas, que (pela nossa regra que a chave esquerda é menor) é o da folha mais à esquerda. Segue que a menor chave no *heap* será armazenada em uma das raízes (mas geralmente não sabemos qual).
Cada nó armazena um par chave-valor, ponteiros filho esquerdo e direito, ponteiro pai e seu nível.
Mantemos dois `arrays` adicionais organizados por nível:

- `roots[lev]`: Uma lista encadeada contendo referências às raízes da árvore do nível `lev`. (Nós recomendo implementar isso como um lista encadeada de nós (`std::list`)).
- `node_counter[lev]`: Armazena o número total de nós no nível `lev`.

`Nodes` e `Locators`: Os principais objetos que estão sendo manipulados são os nós da `QuakeHeap`. Como mencionado acima, cada nó armazena um par chave-valor, links filho esquerdo e direito, link pai, e seu nível na árvore. Os nós no nível 0 são folhas, portanto, ambos os links filho são nulos. Um nó raiz (em qualquer nível) tem um link pai de nulo. C++ fornece uma maneira elegante de definir uma classe aninhada a outra, assim faremos o `Node`, dentro da classe `QuakeHeap`.

Um detalhe complicado em qualquer estrutura de *heap* que suporte diminuir o valor de uma chave é que precisamos de um mecanismo para identificar a entrada cuja chave desejamos diminuir. Quando inserimos um valor-chave par, criamos um novo nó folha. Como o `Node` é um objeto protegido dentro do `QuakeHeap`, não podemos retornar um ponteiro diretamente para ele. Em vez disso, criamos um objeto público especial, chamado `Locator`, para incluir uma referência a este nó folha recém-inserido. A função `insert` retorna um localizador referenciando o nó recém-criado.
Operações: Para esta parte do projeto, começaremos implementando as funções básicas necessárias inserir chaves. Aqui está uma lista das operações que você deve implementar.

- `QuakeHeap(int n_levels)` (10 pontos) : Constrói uma `QuakeHeap` vazia. O parâmetro `n_Level` indica o número de níveis a serem alocados em seus `arrays roots` e `node_counter`. Você deve inicializar essas estruturas e outros atributos que sua classe use.
- `void Clear()` (10 pontos): Isso redefine a estrutura para seu estado inicial. Em particular, ele redefine a contagem `node_counter` para zero e limpa `roots`.
- `Locator Insert(T1 key, T2 value)` (20 pontos): Isso insere o par chave-valor (*key, value*) na *heap*. Isso cria uma árvore "trivial" que consiste em um único nó raiz no nível 0, que armazena esse par chave-valor. Ele insere este nó no array `roots[0]`. Ele retorna um `Locator` (veja acima) referenciando o nó folha recém-criado.
- `vector<string> ListHeap()` (10 pontos): Esta operação lista o conteúdo de sua estrutura em um `vector<string>`. O formato preciso é importante, pois verificamos a correção comparando com os nossos resultados. Enumere os níveis da árvore de baixo para cima. Para cada nível, faça o seguinte:
  - Se a contagem de nós para este nível for zero, pule este nível e vá para o próximo. Caso contrário, ordene os nós raíz deste nível por seu valor de chave.
  - Gere um cabeçalho de nível na forma de uma string`"lev: xxx nodeCt: yyy"` e adicione no `vector`. Aqui, "xxx" é o índice de nível e "yyy" é a contagem de nós para este nível. Por exemplo, se houver quatro nós no nível dois, isso gera a string, "`lev: 2 nodeCt: 4`".
  - Para cada nó raiz `r` na lista de raízes para este nível, enumere os nós deste árvore com base em uma travessia de *preorder*. Para cada nó `u` visitado nesta travessia, fazemos:
    - Se `u.level >= 1` gere a string `"("+ u.key + ")"`. Visite recursivamente `u.left` e `u.right`.
    - Se `u.level = 0` gere a string `"["+ u.key + + u.value + "]"` e retorne.
    - Se `u = nullptr`, retorne a string `"[null]"`.

Código base: Como na tarefa anterior, forneceremos o código esqueleto na classe no *Github Classroom*. Iremos fornecer os arquivos *headers* e o nosso *tester* e os arquivos de texto de teste. Você só precisa implementar a `QuakeHeap` com as funções listadas acima.
Nós vamos avaliar seu trabalho utilizando o seguinte comando para compilar:

```
g++ -std=c++11 tester.cc -o tester
```

Caso o código não compile, a nota será 0. Após a compilação, nós iremos realizar testes baseados em arquivos de texto (que estão dentro da pasta `src/tests`), utilizaremos o comando:

```
tester <tests/test01-input.txt >tests/test01-output.txt
```

Com isso, será gerado um arquivo dentro da pastas tests de output, você deve comparar o `tests/test01-output.txt` com o arquivo `tests/test01-expected.txt` com o comando:

```
diff tests/test01-output.txt tests/test01-expected.txt
```

Nós iremos considerar como um erro caso o comando `diff` aponte diferenças entre os arquivos (incluindo espaços em branco). O número 01 pode ser trocado por 02 e 03 para testar com os outros arquivos de teste. Uma outra avaliação será o estilo do código. Você deve formatar seu código baseado no estilo recomendo pelo Google, de acordo com a referência no seguinte link. Utilizaremos *cpplint* para verificar o seu código. A cada *alerta* referente a formatação, será retirado 5 pontos da nota. Você também pode verificar com as instruções no link. Para executar, em um terminal use o comando:

```
cpplint --extensions=cc,hpp,h --header=h --repository=. tester.cc quake_heap.h quake_heap.hpp
```

A Detalhes avaliação

Uma pergunta comum é "quanto detalhe é esperado nas respostas?", algumas orientações são:

Provar vs. Mostrar: Se lhe pedirmos para "provar" algo, estamos à procura de uma prova bem estruturada. Se você estiver aplicando a indução, tenha cuidado para distinguir seu(s) caso(s) básico(s) e indicar qual é a sua hipótese de indução. Se lhe pedirmos para "mostrar", "explicar" ou "justificar", estaremos geralmente apenas esperando uma explicação em português. Se você não tiver certeza, por favor, verifique.

Algoritmo vs. Pseudocódigo: Quando pedimos um "algoritmo" estamos esperando uma descrição em alto nível de algum processo computacional, geralmente em uma combinação de português e notação matemática (por exemplo, "classifique as n chaves e localize x usando busca binária"). Para pseudocódigo, nós estão esperando uma descrição passo a passo mais detalhada que se pareça muito mais com C++ (por exemplo, "`Node q = p.left`"). Lembre-se de que você está escrevendo seu código para ser lido por um humano, e não por um compilador. Por favor, omita detalhes irrelevantes que são sintaxes de C++. Mesmo que não solicitemos explicitamente, sempre que você fornecer um algoritmo ou pseudocódigo, você deve sempre fornecer uma breve explicação em português. Isso ajuda o avaliador a entender quais são suas intenções, e se houver um pequeno erro em seu código, muitas vezes podemos usar seu explicação para entender quais eram suas reais intenções.

B Adição de Matriz Esparsa

Os estudantes muitas vezes me perguntam quantos detalhes eu espero para perguntas que envolvam dar um algoritmo. Sempre que lhe for pedido que apresente um "algoritmo", você deve apresentar o seguinte (mesmo que eu não peça tudo isso explicitamente):

- Uma breve explicação em português
- Apresentar o próprio algoritmo, tipicamente em pseudo-código
- Se não for óbvio, justifique brevemente a correção do algoritmo
- Faça uma breve análise do tempo de funcionamento

A seguir, apresento uma solução de amostra para o problema da adição de matriz com a matriz esparsa representação. Vamos supor que nos são dadas duas matrizes esparsas A e B (ver Lecture 4), e nosso objetivo é calcular sua soma matricial C. Para simplificar, vamos supor que C é dada como uma matriz $n \times n$ padrão, que é inicializada a zero, e tudo o que precisamos fazer é preencher são as entradas não zeradas. Seguindo as convenções de C++, assumimos que as linhas e colunas são indexados de 0 a $n-1$. Lembramos que na representação de matriz esparsa, nos são dados dois n-element arrays, chame-lhes `row[]` e `col[]`, onde `row[i]` é o chefe (*head*) de uma lista encadeada das entradas na linha `i`, e `col[j]` é o cabeçalho (*head*) de uma lista encadeada de nós na coluna `j`. Estas listas encadeadas são ordenadas por ordem crescente de índices. Cada entrada de matriz diferente de zero é representada por um nó contendo as seguintes informações:

```
Class Node {
  int row;        // indice da linha
  int col;        // indice da coluna
  float value;    // o valor desta entrada da matriz
  Node *rowNext;  // a proxima entrada (da esquerda para a direita) nesta linha
  Node *colNext;  // a proxima entrada (de cima para baixo) nesta coluna
}
```

O algoritmo itera através de cada linha $0 \leq i \leq n-1$, e depois itera de forma coordenada através das duas listas ligadas `A.row[i]` e `B.row[i]`. Se ambas as entradas estiverem na mesma coluna, calculamos sua soma e a armazenamos em `C`. Caso contrário, copiamos o valor que se encontra no índice menor da coluna. Após

processarmos uma entrada, avançamos para o próximo elemento da lista encadeada. Quando chegamos ao final de qualquer uma das listas, simplesmente copiamos as demais entradas da outra lista para as entradas apropriadas em `C`.
Aqui está o pseudo-código. Omitiremos tanto quanto possível as especificações do tipo.

```
void sparseAddition(SparseMatrix A, SparseMatrix B, float C[][])
  for (i = 0 to n-1) {                    // iterate through all the rows
    ap = A.row[i]                         // head of A's ith row
    bp = B.row[i]                         // head of B's ith row
    while (ap != NULL && bp != NULL) {    // while entries remain in both rows
      if (ap.col < bp.col) {              // A's entry comes next?
        C[i][ap.col] = ap.value           //copy to C and advance
        ap = ap.rowNext
      } else if (bp.col < ap.col) {       // B's entry comes next?
        C[i][bp.col] = bp.value // copy to C and advance
        bp = bp.rowNext
      } else { // (ap.col == bp.col)      // both in the same column?
        C[i][ap.col] = ap.value + bp.value // put sum in C and advance
        ap = ap.rowNext
        bp = bp.rowNext
      }
    }
    // At this point, only one list has elements remaining
    while (ap != null) {                  // copy any remainder from A to C
      C[i][ap.col] = ap.value
      ap = ap.rowNext
    }
    while (bp != null) {                  // copy any remainder from B to C
      C[i][bp.col] = bp.value
      bp = bp.rowNext
    }
  }
```

A correção decorre do fato de que os dois indicadores `ap` e `bp` se movem em coordenação, de modo que um nunca fica muito à frente do outro. Para obter o tempo de execução, observe que visitamos cada um dos nós das representações de matriz esparsa para `A` e `B` exatamente uma vez (quando estamos processando sua fila). Como há N_A nós em A e N_B em B, isto leva tempo $O(N_A + N_B)$. Entretanto, mesmo que estas quantidades sejam ambas zero, ainda assim será necessário acessar cada uma das n entradas de `A.row` e `B.row`. Assim, o tempo total de execução é $O(n + N_A + N_B)$.