Módulo 3

Estruturas de Dados

Lição 7

Listas

Versão 1.0 - Mai/2007

Autor
Joyce Avestro

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Reginald Hutcherson
Rebecca Ong
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Anna Carolina Ferreira da Rocha
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bárbara Angélica de Jesus Barbosa
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alexandre de Sene
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Everaldo de Souza Santos
Fabrício Ribeiro Brigagão
Fernando Antonio Mota Trinta
Frederico Dubiel
Givailson de Souza Neves
Jacqueline Susann Barbosa
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Kefreen Ryenz Batista Lacerda
Leonardo Leopoldo do Nascimento
Lucas Vinícius Bibiano Thomé
Luciana Rocha de Oliveira
Luís Carlos André
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Marcus Borges de S. Ramos de Pádua
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Renato Alves Félix
Renê César Pereira
Reyderson Magela dos Reis
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Fernandes Suguiura
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Silvio Sznifer
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Lista é uma estrutura de dado que é baseada numa seqüência de itens. Nessa lição, iremos cobrir os dois tipos de listas – *linear* e *generalizada* – e suas diferentes representações. Listas **encadeadas simples**, **circulares** e **encadeadas duplas** também serão exploradas. Além disso, duas aplicações irão ser apresentadas – aritmética polinomial e alocação dinâmica de memória. *Aritmética polinomial* inclui a representação de operações polinomiais e aritméticas definidas nela. *Alocação dinâmica de memória* cobre ponteiros e estratégias de alocação. Também haverá uma breve discussão sobre conceitos de fragmentação.

Ao final desta lição, o estudante será capaz de:

- Explicar definições e conceitos básicos de listas
- Usar as diferentes **representações de lista**: seqüencial e encadeada
- Diferenciar lista **encadeada simples**, lista **encadeada dupla**, lista **circular** e lista com ***header nodes***
- Explicar como as listas são aplicadas na **aritmética polinomial**
- Discutir as **estruturas de dados** usadas na **alocação dinâmica de memória** usando métodos ***sequential-fit*** e métodos ***buddy-system***

2. Definições e conceitos relacionados

Uma **Lista** é um conjunto finito com nenhum ou "n" elementos. Os elementos de uma lista podem ser *átomos* ou *listas*. Um **átomo** é distinguível de uma lista. Listas são classificadas em dois tipos – lineares e generalizadas. **Listas lineares** contém apenas elementos átomos, e são ordenadas seqüencialmente. **Listas generalizadas** podem conter ambos elementos átomo e lista.

2.1. Lista Linear

O ordenamento linear dos átomos é a propriedade essencial de uma lista linear. A seguir a notação para esse tipo de lista:

$$L = (i_1, i_2, i_3, ..., i_n)$$

Várias operações podem ser feitas com uma lista linear. Inserção pode ser feita em qualquer posição. Similarmente, qualquer elemento pode ser deletado de qualquer posição. A seguir estão as operações que podem ser feitas nas listas lineares:

- Inicialização (a lista é igual a NULL)
- Determinar se a lista é vazia (checando se L != NULL)
- Achar o tamanho (obtendo o número de elementos)
- Acessar o j-ésimo elemento, $0 \leq j \leq n-1$
- Atualizar o j-ésimo elemento
- Deletar o j-ésimo elemento
- Inserir um novo elemento
- Combinar duas ou mais listas em uma única lista
- Dividir uma lista em duas ou mais listas
- Duplicar uma lista
- Apagar uma lista
- Buscar por um valor
- Ordenar a lista

2.2. Lista Generalizada

Uma lista generalizada pode conter elementos átomo e lista. Ela tem profundidade e tamanho. A lista generalizada é também conhecida como **lista estruturada**, ou simplesmente **lista**. A seguir um exemplo:

L = ((a, b, (c, ( ))), d, ( ), (e,( ), (f, (g, (a))), d))

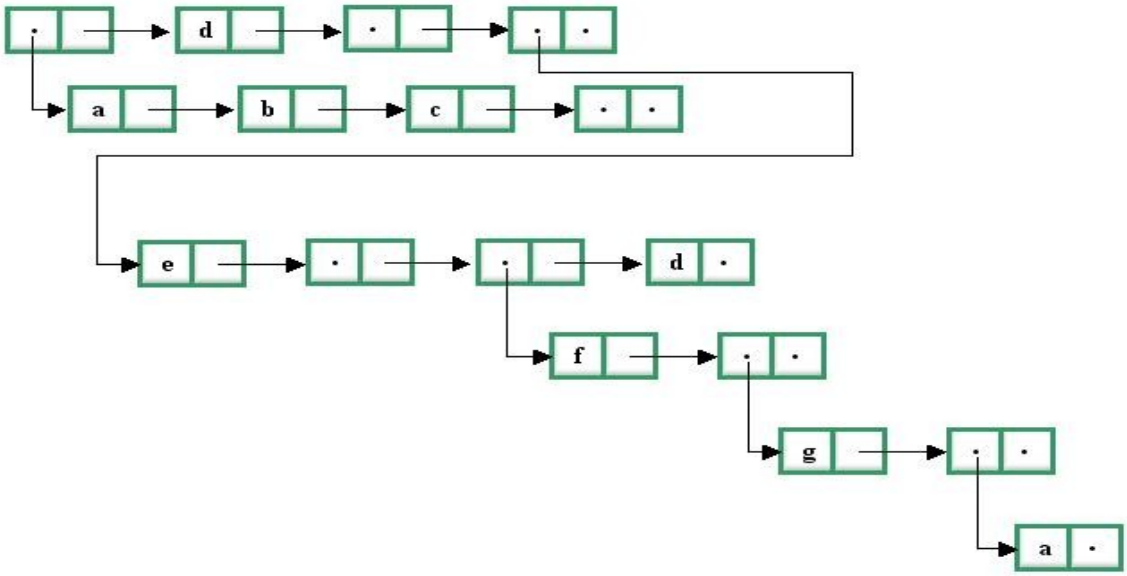


Figura 1. Uma lista generalizada

No exemplo, a lista **L** possui quatro elementos. O primeiro elemento é a lista (a, b, (c, ( ))), o segundo é o átomo **d**, o terceiro é o conjunto null () e o quarto é a lista (e, ( ), (f, (g, (a))), d).

3. Representações de lista

Uma forma de representar uma lista é organizar os elementos um após o outro em uma estrutura seqüencial como um *array*. Outra forma para implementar isso, é o encadeamento de *nodes* contendo os elementos da lista usando uma representação encadeada.

3.1. Representação seqüencial de lista linear encadeada simples

Na representação seqüencial, os elementos são armazenados **contiguamente**. Há um ponteiro para o último item na lista. A seguir é mostrada uma lista usando representação seqüencial:

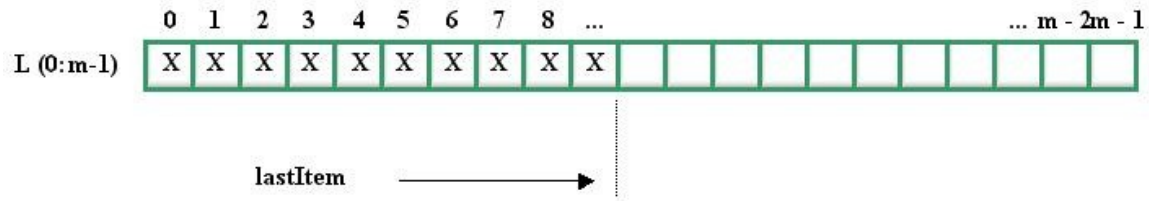


Figura 2. Representação Seqüencial de Lista

Com essa representação, poderá ser tomado O(1) tempo para acessar e atualizar o j-ésimo elemento na lista. Por fazer o caminho até o último item, poder ser tomado O(1) tempo para dizer se a lista é vazia e para achar o tamanho da lista. Há uma condição, no entanto, em que o primeiro elemento deve ser sempre armazenado no primeiro índice L(0). Nesse caso, inserir e deletar podem requerer desvio de elementos para assegurar que a lista satisfaz essa condição. No pior caso, isso pode obrigar desvio de *todos* os elementos no array, resultando na complexidade de tempo de O(n) para inserir e excluir **n** elementos. Ao combinar duas listas, um array mais largo é requerido se o tamanho combinado não couber em alguma das duas listas. Isso pode obrigar a copiar todos os elementos das duas listas na nova lista. Duplicar uma lista pode requerer atravessar a lista inteira, portanto, uma complexidade de tempo de O(n). Buscar um valor em particular pode tomar tempo de O(1) se o elemento for o primeiro na lista; de outra forma, o pior caso é quando o elemento pesquisado é o último, onde a passagem da lista inteira é necessária. Nesse caso, a complexidade de tempo é O(n).

A alocação seqüencial, sendo estática por natureza, é uma desvantagem para listas de tamanho incerto, por exemplo, o tamanho não é sabido no momento da inicialização, e com várias inserções e remoções, pode eventualmente precisar crescer ou encolher. Copiando a lista transbordada em um array mais largo e descartando o antigo pode funcionar, mas isso pode ser um desperdício de tempo. Nesse caso, é melhor usar a representação encadeada.

3.2. Representação encadeada de lista linear encadeada simples

Uma corrente de *nodes* encadeados pode ser usada para representar uma lista.

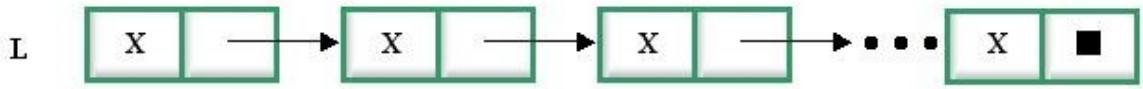


Figura 3. Representação Encadeada de Lista

Para acessar o j-ésimo elemento com alocação encadeada, a lista tem que ser percorrida do primeiro elemento até o j-ésimo elemento. O pior caso é quando i = n, onde n é o número de elementos. Portanto, a complexidade de tempo para acessar o j-ésimo elemento é O(n). Similarmente, encontrando o tamanho pode obrigar a percorrer a lista inteira, sendo a complexidade de O(n). Se inserções forem feitas no início da lista, isso pode levar ao tempo O(1). De outra forma, com

remoção e atualização, a busca tem que ser realizada resultando na complexidade de tempo de O(n). Para dizer se a lista é vazia, pode ser tomado um tempo constante, como na representação seqüencial. Para copiar uma lista, cada *node* é copiado enquanto a lista original é percorrida.

A tabela seguinte resume as complexidades de tempo das operações realizadas em listas com os dois tipos de alocação:

Operação	***Representação Seqüencial***	***Representação Encadeada***
Determinar se uma lista é vazia	O(1)	O(1)
Encontrar o tamanho	O(1)	O(n)
Acessar o j-ésimo elemento	O(1)	O(n)
Atualizar o j-ésimo elemento	O(1)	O(n)
Deletar o j-ésimo elemento	O(n)	O(n)
Inserir um novo elemento	O(n)	O(1)

Tabela 1: Representação Seqüencial x Encadeada

A representação seqüencial é apropriada para listas que são estáticas por natureza. Se o tamanho é desconhecido, o uso de alocação encadeada é recomendado.

Ainda no encadeamento simples linear, há mais variedades de representações encadeadas de listas. Encadeamento simples circular, encadeamento duplo e lista com *header nodes* são as variedades mais comuns.

3.3. Lista circular encadeada simples

Uma lista circular encadeada simples é formada pelo envio do *link* do último *node* e apontar de volta ao primeiro *node*. Isso é ilustrado na seguinte figura:

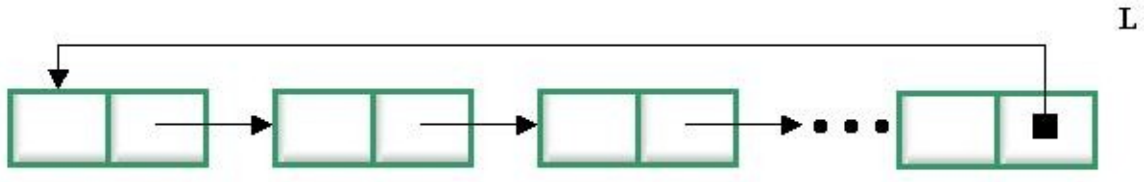


Figura 4. Lista Circular Encadeada Simples

Observe que o ponteiro para a lista nessa representação aponta para o último elemento na lista. Com lista circular, existe a vantagem de ser possível acessar um *node* de qualquer outro *node*.

Iniciamos a classe que exemplificará a lista circular encadeada, com dois nodes de referência, para o primeiro e o último elemento:

```
public class CircularList {
  private Node F;
  private Node L;
}
```

Lista circular pode ser usada para implementar uma ***stack***. Nesse caso, inserção (*push*) pode ser feita na ponta esquerda da lista, e remoção (*pop*) na mesma ponta. Similarmente, ***queue*** também pode ser implementada permitindo inserção na ponta direita da lista e remoção na ponta esquerda.

3.3.1. Inserção a Esquerda

O seguinte método inserido na classe realiza o procedimento de inserir um elemento X na ponta da esquerda da lista circular:

```
public void insertLeft(Object x) {
  Node alpha = new Node(x,null);
  if (F == null) {
    alpha.link = alpha;
    F = alpha;
  } else {
    alpha.link = F.link;
    F.link = alpha;
  }
  L = alpha;
}
```

No método, se a lista é inicialmente vazia, resultará a seguinte lista circular:

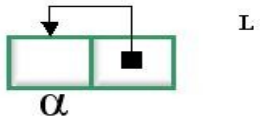


Figura 5. Inserção em uma lista vazia

De outra forma, será realizado, conforme o seguinte diagrama:

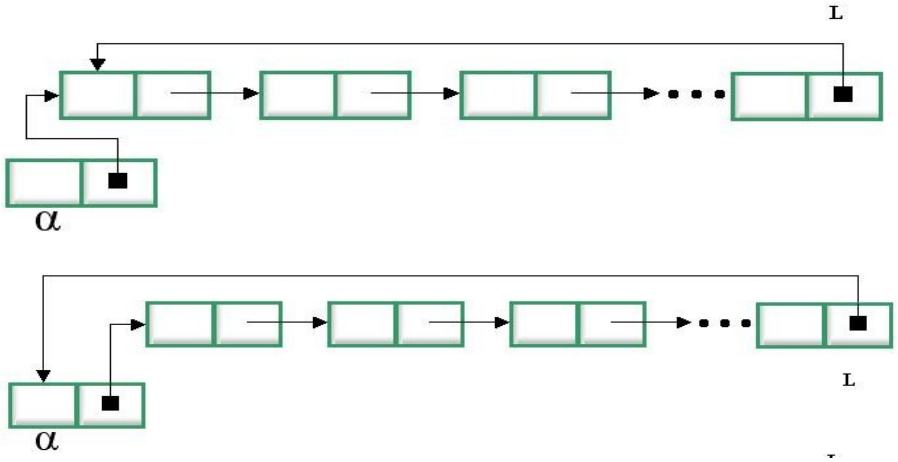


Figura 6. Inserção em uma lista não vazia

3.3.2. Inserção a Direita

O seguinte método inserido na classe realiza o procedimento de inserir o elemento X na ponta da direita da lista circular:

```
public void insertRight(Object x) {
  Node alpha = new Node(x,null);
  if (L == null) {
    alpha.link = alpha;
    F = alpha;
  } else {
    alpha.link = L.link;
    L.link = alpha;
```

```
    }
    L = alpha;
  }
```

Se a lista é inicialmente vazia, o resultado de *insertRight* é similar ao de *insertLeft*, entretanto para uma lista não vazia,

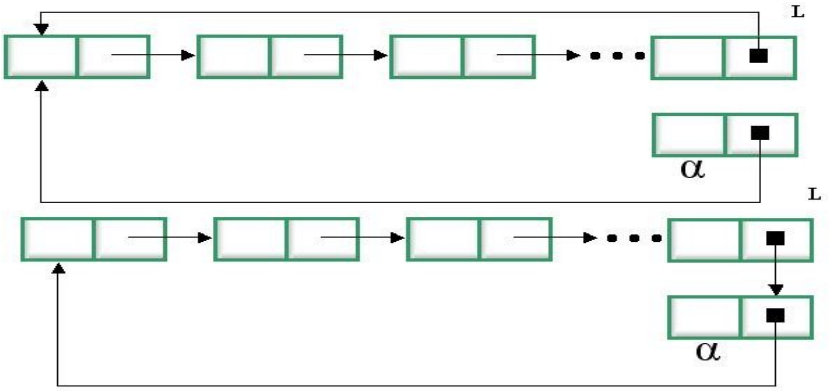


Figura 7. Inserção à Direita

3.3.3. Exclusão a Esquerda

O seguinte método inserido na classe realiza o procedimento de eliminar o elemento mais à esquerda da lista circular L:

```
public void deleteLeft() throws Exception {
  Node alpha;
  if (L == null)
    throw new Exception("CList empty.");
  else {
    F.link = L.link;
    alpha = L.link;
    if (alpha == L)
      L = null;
    else
      L = alpha;
  }
}
```

Execução de *deleteLeft* em uma lista não vazia é ilustrada conforme a **figura 8**.

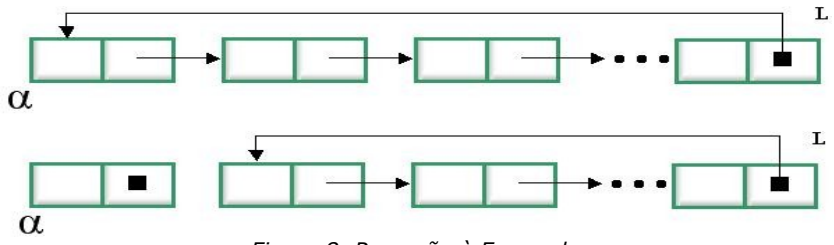


Figura 8. Remoção à Esquerda

Todos esses três procedimentos têm complexidade de tempo de O(1).

3.3.4. Concatenação de duas listas

Outra operação que possui tempo O(1) em uma lista circular é a **concatenação**. Isto é, dadas duas listas:

$L_1 = (a_1, a_2, ..., a_m)$ e $L_2 = (b_1, b_2, ..., b_n)$

Onde m, n ≥ 0, as listas resultantes são:

$L_1 = (a_1, a_2, ..., a_m, b_1, b_2, ..., b_n)$

$L_2 = null$

Isso pode ser feito por simples manipulações do campo de ponteiro dos *nodes* realizando uma junção de duas listas:

```
public void concatenate(Node List2) {
  if ((List2 != null) && (L != null)) {
    Node alpha = L.link;
    L.link = List2.link;
    List2.link = alpha;
  }
}
```

Se L2 é não vazia, o processo de concatenação é ilustrado abaixo:

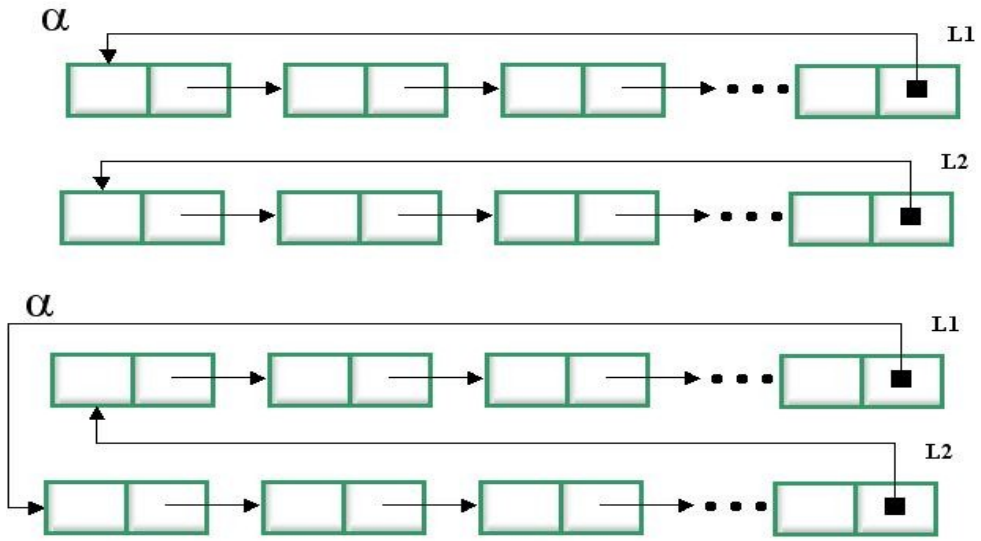


Figura 9. Concatenação de duas listas

Terminamos esta classe com um método para mostrar o conteúdo sequencial da lista (lembrando que a mesma é circular, então devemos listar uma determinada quantidade de elementos, pois não é possível localizar o fim da lista).

```
public void showList(Node n, int t) {
  if (t-- == 0)
    return;
  System.out.println(n.info);
  showList(n.link, t);
}

public static void main(String[] args) throws Exception {
  // Cria uma lista com 4 elementos inseridos pela esquerda
  CircularList cL = new CircularList();
  for (int i = 1; i < 5; i++)
    cL.insertLeft("Element " + i + "L");
  System.out.println("Left");
  cL.showList(cL.F, 5);
```

```
    // Elimina o elemento mais a esquerda
    System.out.println("After DeleteLeft");
    cL.deleteLeft();
    cL.showList(cL.F, 4);

    // Cria uma lista com 4 elementos inseridos pela direita
    CircularList cR = new CircularList();
    for (int i = 1; i < 5; i++)
      cR.insertLeft("Element " + i + "R");
    System.out.println("Right");
    cR.showList(cR.F, 5);

    // Concatena a lista da esquerda com a da direita
    cL.concatenate(cR.F);
    System.out.println("Concate");
    cL.showList(cL.L, 8);
  }
```

E como resultado será produzido:

```
Left
Element 1L
Element 4L
Element 3L
Element 2L
Element 1L
After DeleteLeft
Element 1L
Element 3L
Element 2L
Element 1L
Right
Element 1R
Element 4R
Element 3R
Element 2R
Element 1R
Concate
Element 3L
Element 4R
Element 3R
Element 2R
Element 1R
Element 2L
Element 1L
Element 3L
```

observe que sempre listamos um elemento a mais para mostrar a circularidade da lista.

3.4. Lista encadeada simples com header nodes

Um *header node* pode ser usado para armazenar informação adicional sobre a lista, como o número de elementos. O *header node* é também conhecido como ***list header***. Para uma lista circular, ele serve como *sentinela* para indicar passagem completa pela lista. Também pode indicar tanto o começo como o ponto final. Para uma lista encadeada dupla, ele é usado para apontar para ambos

os finais, para fazer inserção ou armazenamento de realocação rapidamente. O cabeçalho da lista é também usado para representar a lista vazia por um objeto não nulo em uma linguagem de programação orientada a objeto como Java, então esses métodos como inserção podem ser invocados em uma lista vazia. A próxima figura ilustra uma lista circular com um cabeçalho de lista:

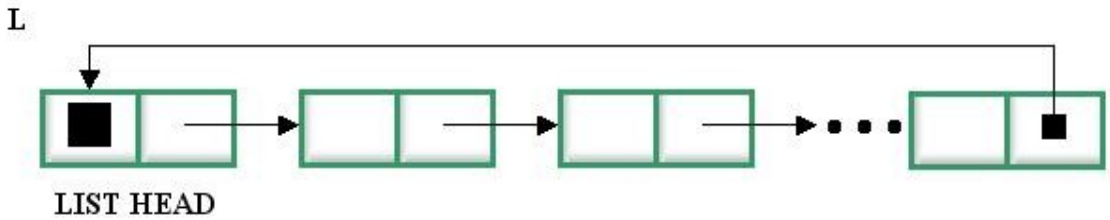


Figura 10. Lista Encadeada Simples com Cabeçalho de Lista

3.5. Lista encadeada dupla

Listas encadeadas duplas são formadas de *nodes* que possuem ponteiros para ambos vizinhos (esquerdo e direito) na lista. A próxima figura mostra a estrutura de *nodes* para uma lista encadeada dupla.

Figura 11. Estrutura de nodes de Lista Encadeada Dupla

A seguir um exemplo de uma lista encadeada dupla:

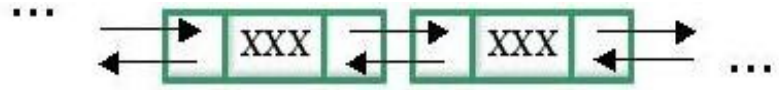


Figura 12. Lista Encadeada Dupla

Com lista encadeada dupla, cada *node* tem dois campos de ponteiro – LLINK e RLINK que apontam para os vizinhos da esquerda e direita respectivamente. A lista pode ser percorrida em ambas as direções. Um *node* **i** pode ser deletado sabendo apenas a informação sobre o *node* **i**. Ainda, um *node* **j** pode ser inserido tanto antes quanto depois de um *node* **i** sabendo a informação sobre **i**. A figura seguinte ilustra a remoção do *node* **i**:

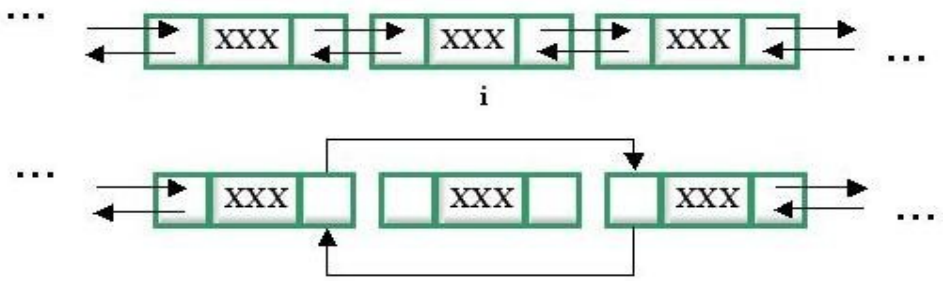


Figura 13. Remoção em uma Lista Encadeada Dupla

Uma lista encadeada dupla pode ser constituída das seguintes formas:

- Lista linear encadeada dupla
- Lista circular encadeada dupla
- Lista circular encadeada dupla com cabeçalho de lista

Propriedades de lista encadeada dupla

- LLINK(L) = RLINK(L) = L significa que a lista L é vazia.
- Podemos deletar qualquer *node*, como o *node* α, em L no tempo O(1), sabendo apenas o endereço α.
- Podemos inserir um novo *node*, como o *node* β, à esquerda (ou direita) de qualquer *node*, como o node α, no tempo O(1), sabendo apenas α sendo que só precisa de ajustes de ponteiro, como ilustrado abaixo:

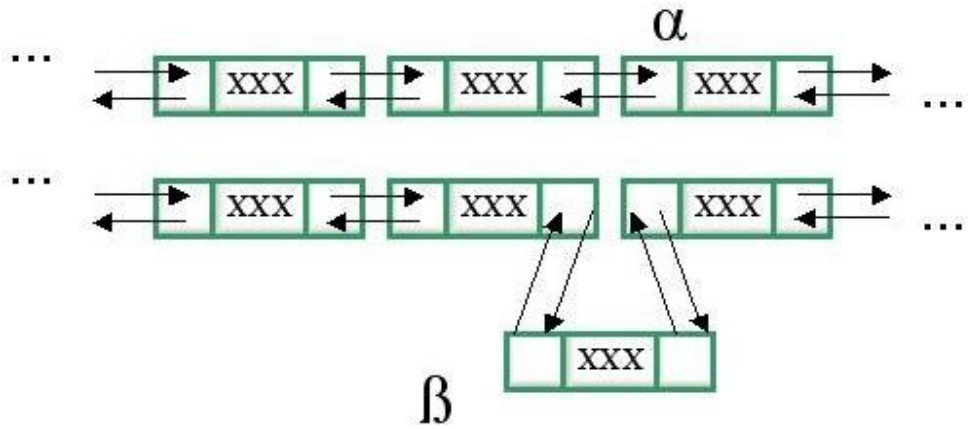


Figura 14. Inserção em uma Lista Encadeada

4. Aplicação: Aritmética Polinomial

As listas podem ser usadas para representar polinômios nos quais podem ser aplicadas operações aritméticas. Há dois assuntos que têm que ser tratados:

- Um caminho para representar os termos de um polinômio, tal que cada termo pode ser acessado e processado facilmente pelas entidades que os incluem.
- Uma estrutura dinâmica, isto é, para crescer e diminuir conforme necessário.

Para tratar estes assuntos, lista circular simplesmente ligada com cabeça de lista pode ser usada, com uma estrutura de *node* como ilustrado abaixo:

Figura 15. Estrutura do node

onde,

- O campo EXPO é dividido em um subcampo de sinal (S) e três subcampos de expoentes para as variáveis x, y, z.
  - S é negativo (-) se for a cabeça de lista, caso contrário é positivo
  - e_x, e_y, e_z são para as potências das variáveis x, y e z respectivamente
- O campo COEF pode conter qualquer número real, com ou sem sinal.

Por exemplo, a figura seguinte é a representação da lista do polinômio $P(x,y,z) = 20x^5y^4 - 5x^2yz + 13yz^3$:

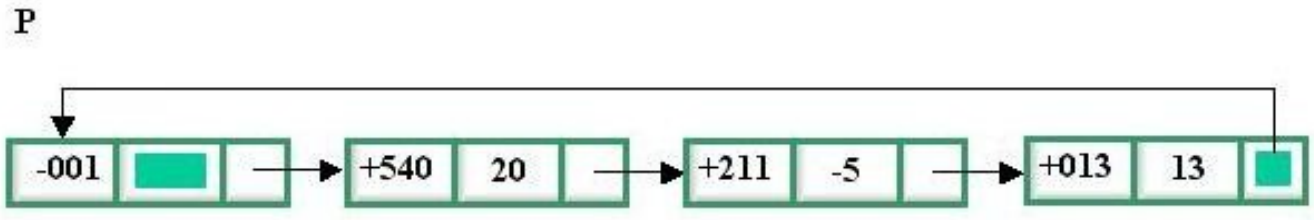


Figura 16. Um Polinômio Simbolizado, usando Representação Ligada

Neste aplicativo, há uma regra em que os *nodes* devem ser arranjados em valor decrescente do triplo ($e_xe_ye_z$). Um polinômio satisfazendo esta propriedade é dito estar em **forma canônica**. Esta regra torna o desempenho de executar as operações aritméticas do polinômio mais rápida, em comparação aos termos estarem organizados em nenhuma ordem particular.

Desde que a estrutura de lista tem uma cabeça de lista, para representar o zero polinomial, temos o seguinte:

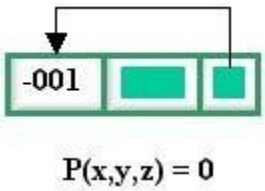


Figura 17. Zero Polinomial

Em Java, o seguinte é a definição de um termo polinomial:

```
class PolyTerm {
  int expo;
  int coef;
  PolyTerm link;

  // Cria um termo novo que contém o expo -001 (cabeça de lista)
  public PolyTerm() {
    expo = -1;
    coef = 0;
    link = null;
  }
  // Cria um termo novo com expo e o coeficiente
  public PolyTerm(int e, int c) {
    expo = e;
    coef = c;
    link = null;
  }
}
```

e o seguinte é a definição de um Polinômio:

```
class Polynomial {
  PolyTerm head = new PolyTerm(); // list head
  public Polynomial() {
    head.link = head;
  }
  // Cria um novo polinômio com head h
  public Polynomial(PolyTerm h) {
    head = h;
    h.link = head;
  }
}
```

Para inserir termos de **forma canônica**, o seguinte é um método da classe *Polynomial*:

```
/* Insere um termo para [this] polinômio inserindo
em seu próprio local, para manter a forma canônica */
public void insertTerm(PolyTerm p) {
  PolyTerm alpha = head.link;  // ponteiro móvel
  PolyTerm beta = head;
  if (alpha == head) {
    head.link = p;
    p.link = head;
    return;
  } else {
    while (true) {
      /* Se o termo corrente é menor do que alpha
         ou é o menor no polinômio, então insire */
      if ((alpha.expo < p.expo) || (alpha == head)) {
        p.link = alpha;
        beta.link = p;
        return;
      }

      // Avançar alpha e beta
      alpha = alpha.link;
      beta = beta.link;
```

```
            // Se o círculo está completo, retorna
            if (beta == head)
              return;
          }
        }
      }
```

4.1. Algoritmos de Aritmética Polinomial

Esta seção cobre como adição polinomial, subtração e multiplicação podem ser implementadas, usando a estrutura há pouco descrita.

4.1.1. Adição Polinomial

Somando dois polinômios P e Q, a soma é retida em Q. Três ponteiros de execução são necessário:

- α para apontar para o termo corrente (*node*) em P polinomial
- β para apontar para o termo corrente em Q polinomial
- σ apontar para o *node* atrás de β. Isto é usado durante a inserção e a deleção em Q para obter a complexidade do tempo O(1).

O estado de α e β cairá em um dos seguintes casos:

- EXPO (α) < EXPO (β)

  Ação: avançar os ponteiros para o polinômio Q um *node* acima

- EXPO (α) > EXPO (β)

  Ação: copiar o termo corrente em P e insira-o antes do termo corrente em Q, então avançar α

- EXPO (α) = EXPO (β)

  - Se EXPO (α) < 0, ambos os ponteiros α e β têm uma volta completa no círculo e estão agora apontando para as cabeças de lista

    Ação: terminar o procedimento

  - Senão, α e β estão apontando para dois termos que podem ser adicionados

    Ação: adicionar os coeficientes. Quando o resultado é zero, deletar o *node* de Q. Mover P e Q para o termo seguinte.

Por exemplo, adicionar os dois polinômios seguintes:

$P = 67xy^2z + 32x^2yz – 45xz^5 + 6x – 2x^3y^2z$

$Q = 15x^3y^2z - 9xyz + 5y^4z^3 - 32x^2yz$

Já que os dois polinômios não estão em **forma canônica**, seus termos devem ser reordenados antes deles serem representados, como:

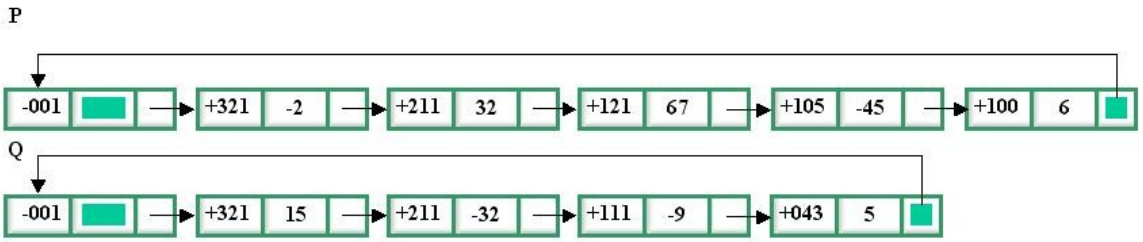
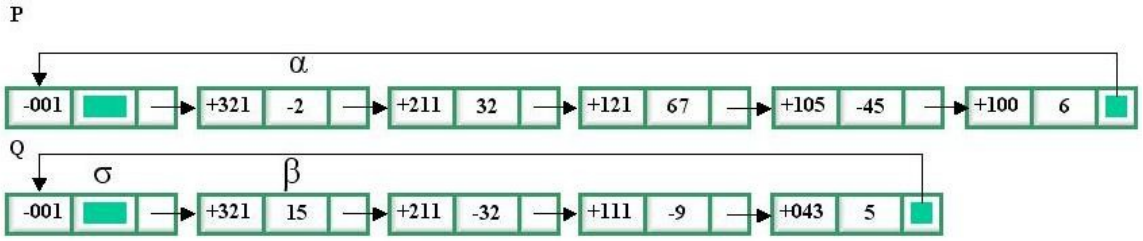


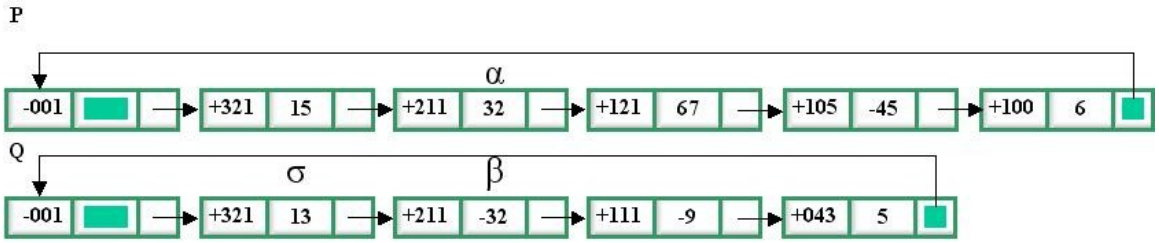
Figura 18. Polinômios P e Q

Adicionar os dois,



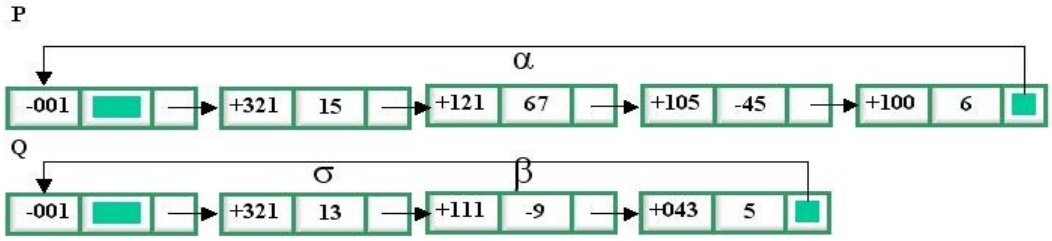
$\text{Expo}(\alpha) = \text{Expo}(\beta)$

adicionar α e β:



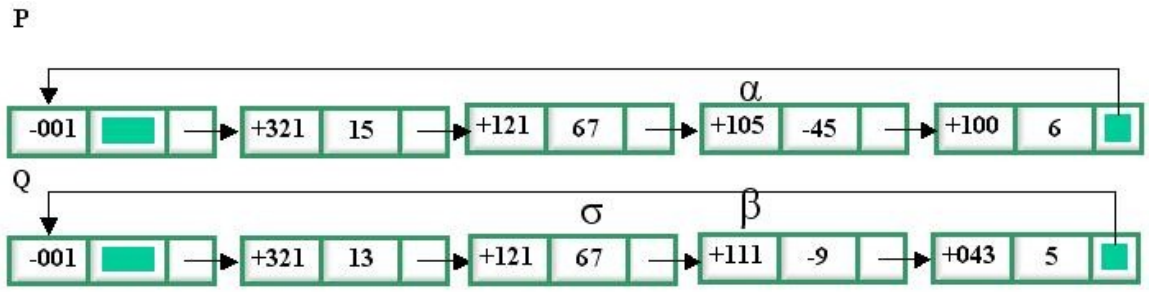
$\text{Expo}(\alpha) = \text{Expo}(\beta)$

adicionar α e β, resultados para excluir o *node* apontado por β:



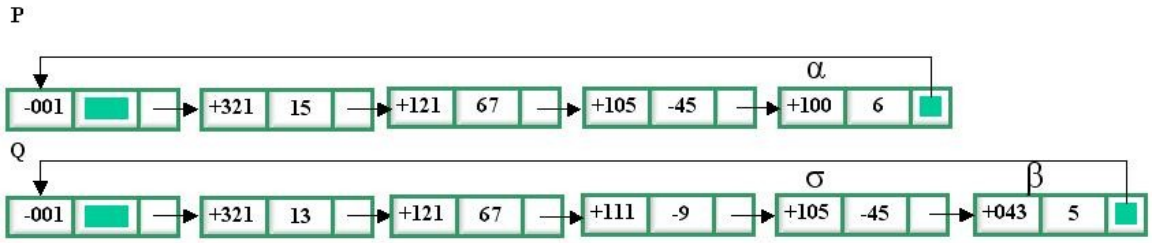
$\text{Expo}(\alpha) > \text{Expo}(\beta)$

inserir α entre σ ε β:



Expo(α) < Expo(β), avançar σ e β:

Expo(α) > Expo(β), inserir α em Q:



Expo(α) > Expo(β), inserir α em Q:

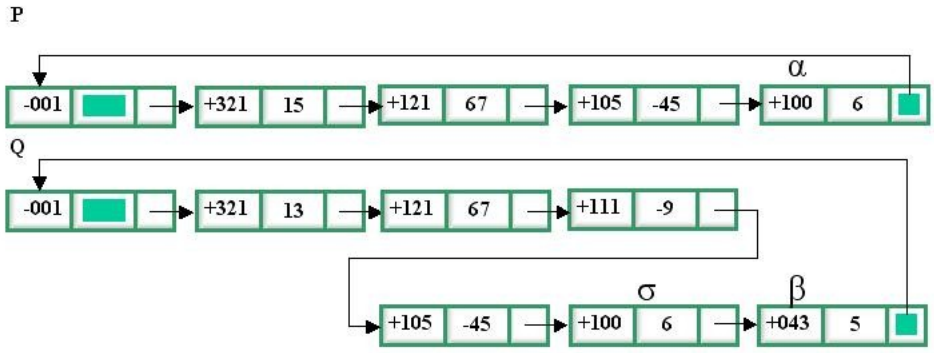


Figura 19. Adição dos Polinômios P e Q

Desde que ambos P e Q estejam em **forma canônica**, uma passagem é suficiente. Se os operando não estiverem em **forma canônica**, o procedimento não produzirá o resultado correto. Se P tem m termos e Q tem n termos, a complexidade de tempo do algoritmo é O(m+n).

Com este algoritmo, não há necessidade para manipulação especial do zero polinomial. Ele trabalha com zero P e/ou Q. Porém, desde que a soma é retida em Q, ele tem que ser duplicado, se a necessidade de usar Q depois da adição poderá surgir. Poderia ser feito chamando o método adicionar(Q, P) da classe Polinomial, onde P é inicialmente o zero polinomial e contém a duplicata de Q.

O seguinte é a implementação de Java deste Procedimento:

```
// Executa a operaçãoQ = P + Q, Q é [this] polinômio
public void add(Polynomial P) {
  // Ponteiro móvel em P
  PolyTerm alpha = P.head.link;
  // Ponteiro móvel em Q
  PolyTerm beta = head.link;
```

```
  // Ponteiro para o node atrás de beta, usado na inserção para Q
  PolyTerm sigma = head;
  PolyTerm tau;
  while (true) {
    // Termo corrente em P > termo corrente em Q
    if (alpha.expo < beta.expo) {
      // Avançar os ponteiros em Q
      sigma = beta;
      beta = beta.link;
    } else if (alpha.expo > beta.expo) {
      // Inserir o termo corrente em P para Q
      tau = new PolyTerm();
      tau.coef = alpha.coef;
      tau.expo = alpha.expo;
      sigma.link = tau;
      tau.link = beta;
      sigma = tau;
      alpha = alpha.link; // Avançar o ponteiro em P
    } else {  // Termos em P e Q podem ser adicionados
      if (alpha.expo < 0)
        return; // A soma já está em Q
      else {
        beta.coef = beta.coef + alpha.coef;
        // Se adicionando causará cancelamento do termo
        if (beta.coef == 0) {
          // tau = beta;
          sigma.link = beta.link;
          beta = beta.link;
        } else { // Avançar os ponteiros em Q
          sigma = beta;
          beta = beta.link;
        }
        // Avançar o ponteiro em P
        alpha = alpha.link;
      }
    }
  }
}
```

4.1.2. Subtração Polinomial

Subtração de um polinomial Q de P, i.e., Q = P-Q, é simplesmente uma adição polinomial com cada termo em Q negado: Q = P + (-Q). Isto pode ser feito percorrendo Q e negando os coeficientes no processo antes de chamar *polyAdd*.

```
// Executa a operação Q = Q-P, Q é [this] polinômio
public void subtract(Polynomial P){
  PolyTerm alpha = P.head.link;
  // Negar todos os termos em P
  while (alpha.expo != -1) {
    alpha.coef = - alpha.coef;
    alpha = alpha.link;
  }
  // Adicionar P para [this] polinômio
  this.add(P);
  // Restaurar P
```

```
  while (alpha.expo != -1) {
    alpha = alpha.link;
    alpha.coef = - alpha.coef;
  }
}
```

4.1.3. Multiplicação Polinomial

Para multiplicar dois polinômios P e Q, um polinômio R inicialmente zero é necessário para conter o produto, isto é, R = R + P*Q. No processo, todos os termos em P são multiplicados com todos os termos em Q.

O seguinte é a implementação Java:

```
/* Executar a operação R = R + P*Q, onde T é inicialmente
   um zero polinomial e R é this polinômio */
public void multiply(Polynomial P, Polynomial Q) {
  // Criar o polinômio temporário T, para conter termo do produto
  Polynomial T = new Polynomial();
  // Ponteiro móvel em T
  PolyTerm tau = new PolyTerm();
  // Conter o produto
  Polynomial R = new Polynomial();
  // Ponteiro móvel em P e Q
  PolyTerm alpha, beta;
  // Inicializar T e tau
  T.head.link = tau;
  tau.link = T.head;
  // Multiplicar
  alpha = P.head.link;
  // Para todos os termos em P
  while (alpha.expo != -1) {
    beta = Q.head.link;
    // multiplicar com todos os termos em Q
    while (beta.expo != -1) {
      tau.coef = alpha.coef * beta.coef;
      tau.expo = expoAdd(alpha.expo, beta.expo);
      R.add(T);
      beta = beta.link;
    }
    alpha = alpha.link;
  }
  this.head = R.head; // Fazer [this] polinômio ser R
}
/* Executar a adição de expoentes do triplo(x,y,z)
   Método auxiliar usado por multiply */
public int expoAdd(int expo1, int expo2) {
  int ex = expo1/100 + expo2/100;
  int ey = expo1%100/10 + expo2%100/10;
  int ez = expo1%10 + expo2%10;
  return (ex * 100 + ey * 10 + ez);
}
```

5. Alocação Dinâmica de Memória

Alocação Dinâmica de Memória (*DMA - Dynamic Memory Allocation*) refere-se ao gerenciamento de uma área contínua de memória, chamada ***memory pool***, usando técnicas para alocar e desalocar blocos. Assume-se que o *memory pool* consiste de unidades individuais endereçáveis chamadas ***words***. O tamanho de um bloco é mensurado em words. A alocação dinâmica de memória também é conhecida como alocação dinâmica de armazenamento ou *dynamic storage allocation*. Na DMA, blocos existem em tamanho variável, daqui, *getNode* e *retNode*, como discutido no lição 1, não serão suficientes para gerenciar alocação de *blocos*.

Existem duas operações relacionadas à DMA: reserva e liberação. Durante a **reserva**, um *bloco* de memória é alocado para uma tarefa de requisição (*requesting*). Quando um *bloco* não é mais necessário, ele está pronto para ser liberado. **Liberação** é o processo para retorná-lo ao *memory pool*.

Existem duas técnicas gerais na DMA – método ***sequential fit*** e ***buddy-system***.

5.1. Gerenciando o Memory Pool

É necessário gerenciar o ***memory pool*** para que os *blocos* sejam alocados e desalocados conforme a necessidade. Problemas aparecem após uma seqüência de alocar e desalocar blocos. Isto é, quando o ***memory pool*** consiste de blocos livres e dispersos todos entre blocos reservados do *pool*. Nestes casos, as *linked lists* podem ser utilizadas para organizar blocos livres tornando mais eficientes os processos de **reserva** e **liberação**.

No método ***sequential-fit***, todos os blocos livres estão constituídos em uma lista ***singly-linked*** chamada de **lista disponível**. No método ***buddy-system***, blocos são alocados em tamanhos *quantum* apenas, i.e. 1, 2, 4, 2^k apenas *words*. Assim, algumas listas disponíveis são mantidas, uma para cada tamanho permitido.

5.2. Método Sequential-Fit: Reserva

Uma forma de constituir listas disponíveis é usar a primeira *word* de cada bloco livre como uma ***control word*** (palavra de controle). Consiste de dois campos: SIZE e LINK. *SIZE* contém o tamanho do bloco livre, enquanto *LINK* aponta para o próximo bloco livre no *memory pool*. A lista deve ser ordenada de acordo com o tamanho, endereço, ou deve estar *left unsorted*.

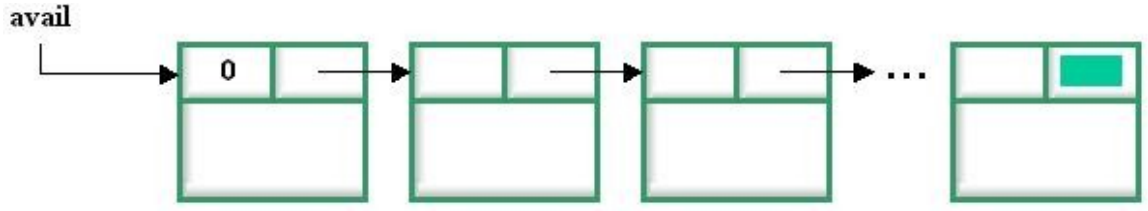


Figura 20. lista disponível

Para satisfazer uma necessidade de n *words*, a *lista disponível* é escaneada por blocos que reúnem um critério apropriado:

- ***first fit*** – o primeiro bloco com *words* m ≥ n
- ***best fit*** – o bloco *best-fitting* (mais apropriado), i.e. o menor bloco com *words* m≥n
- ***worst fit*** – o maior bloco

Após encontrar um bloco, n *words* das que estão reservadas e as restantes m-n são mantidas na **lista disponível**. Todavia, se as *words* restantes m-n forem muito pequenas para satisfazer qualquer pedido, devemos optar por alocar o bloco inteiro.

A abordagem acima é simples mas sofre de dois problemas. Primeiro, retorna para a **lista disponível** o que quer que esteja à esquerda do bloco após a **reserva**. Isto remete a longas buscas e muito do espaço livre não usado é dispersado na **lista disponível**. Segundo, a busca sempre começa no início da **lista disponível**. Daqui, pequenos blocos a esquerda da parte conduzida da lista resultando em buscas muito grandes.

Para resolver o primeiro problema, podemos alocar o bloco inteiro, se o que restar for pequeno para satisfazer o pedido. Podemos definir um valor mínimo como ***minsize***. Usando esta solução, será necessário armazenar o tamanho do bloco desde que o tamanho reservado não coincida com o tamanho atual do *bloco* alocado. Isto deverá ser feito adicionando-se um campo *size* para o bloco reservado, que será usado durante a **liberação**.

Para resolver o segundo problema, poderemos manter a trilha do final da última busca e começar a próxima busca na conexão do mesmo *bloco* atual, i.e. se chegarmos ao fina do bloco A, poderemos iniciar a próxima busca no LINK(A). Um ponto sem destino, dizemos ***rover***, é necessário para manter a trilha deste bloco. O seguinte método implementa estas soluções.

Buscas curtas na ***lista disponível*** fazem a segunda abordagem mais rápida que a primeira. A última abordagem é a que iremos usar para o primeiro método de ajuste da **reserva** de ajuste seqüencial.

Por exemplo, dado o estado do *memory pool* com um tamanho de 64K como ilustrado abaixo,

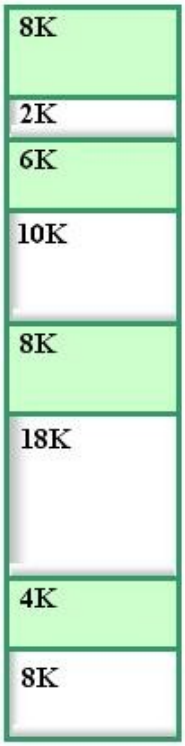


Figura 21. Um Memory Pool

reserve espaço para o seguinte pedido retornando o que quer que esteja a esquerda da alocação.

Task	Request
A	2K
B	8K
C	9K
D	5K

Task	Request
E	7K

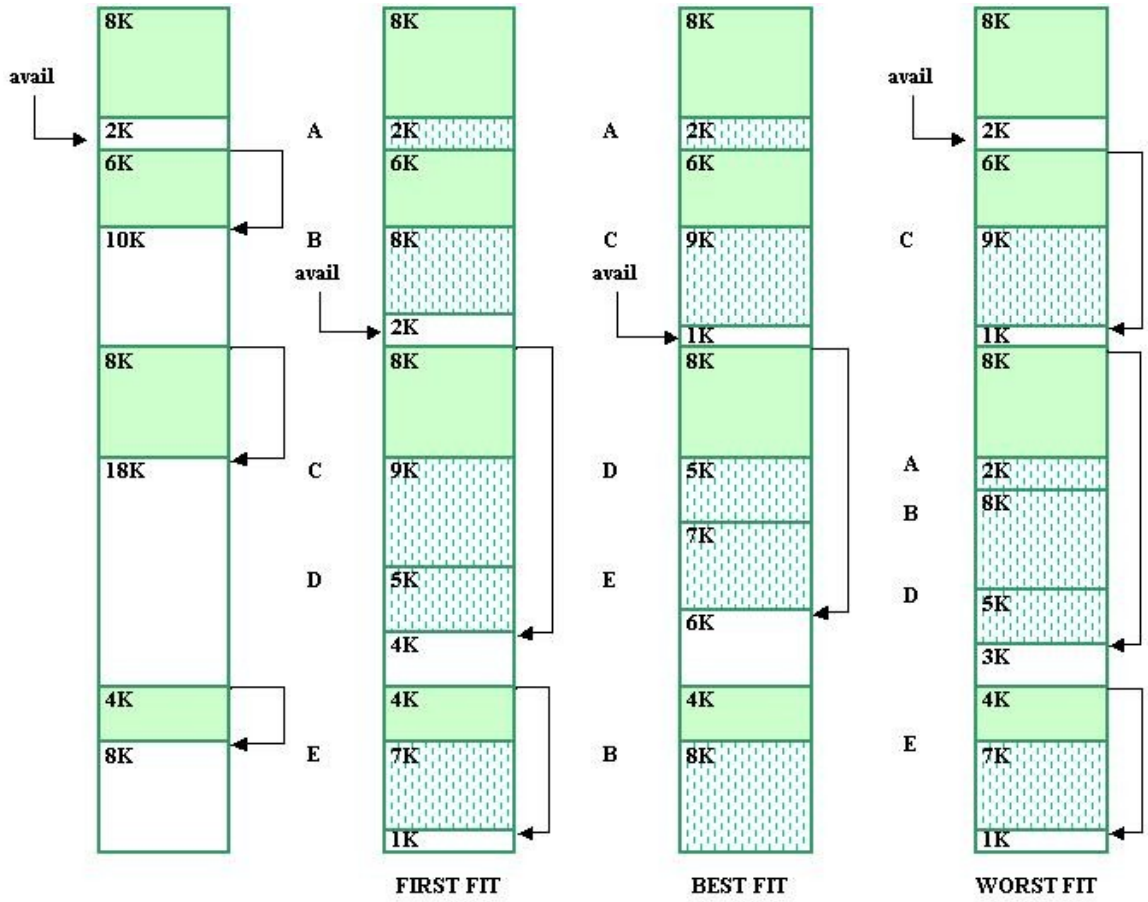


*Figura 22. Resultado de aplicação de três métodos **Sequential Fit***

O método ***best-fit*** reserva largos blocos para futuras requisições enquanto que na **worst-fit**, um largo bloco é alocado para retornar uma grande disponibilidade de blocos para a lista disponível. No método ***best-fit***, há uma necessidade em procurar a lista inteira para achar o melhor bloco ajustado. Há pouco semelhança com o primeiro ajuste, e também existe uma tendência em se acumular blocos muito pequenos. Entretanto, isso pode ser minimizado utilizando ***minsize***. ***Best-fit*** não significa necessariamente que produzirá resultados melhores no primeiro ajuste. Algumas pesquisas mostram que há muitos casos na qual há um resultado melhor que supera o primeiro ajuste. Em algumas aplicações, produto de pior ajuste produz os melhores resultados entre os três métodos.

5.3. Método Sequential-Fit: Liberação

Quando um bloco reservado não é necessário, deve ser liberado imediatamente. Durante a liberação, deve-se desmontar os blocos livres adjacentes para se formar um bloco maior (problema de desmontagem). Esta é uma consideração importante em liberação de ***sequential-fit***.

As figuras seguintes demonstram os quatro possíveis cenários durante liberação de um bloco:

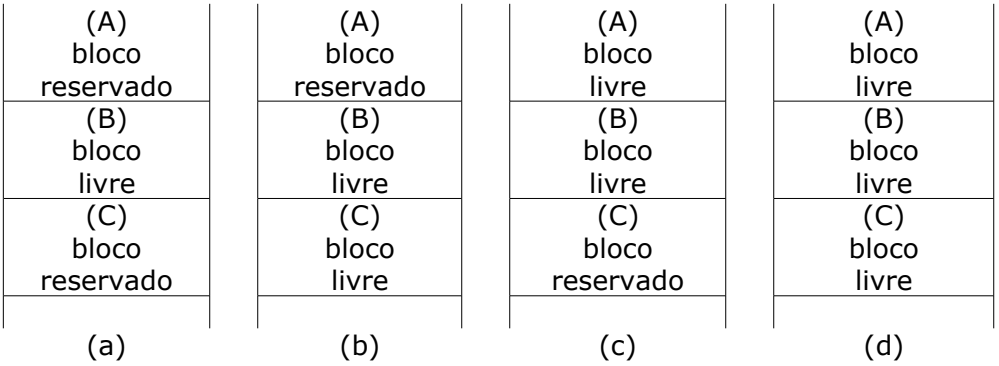


Figura 23. Casos possíveis na Liberação

Na figura, o *bloco* B está liberado. (a) mostra dois blocos adjacentes liberados, (b) e (c) mostram um bloco adjacente liberado e (d) mostra dois *blocos* adjacentes liberados. Para liberar, o bloco liberado deve estar mesclado com o bloco adjacente livre, se existir algum.

Existem dois métodos para mesclar na **liberação**: a técnica ***sorted-list*** e a técnica ***boundary-tag***.

5.3.1. A técnica Sorted-List

Na técnica ***sorted-list***, a **lista disponível** é convertida em uma lista *singly-linked* e é considerada para ser ordenada no aumento de endereços e memória. Quando um bloco está liberado, são necessárias as seguintes interações:

- O *bloco* recentemente liberado vem antes de um bloco liberado;
- O *bloco* recentemente liberado vem após um bloco livre; ou
- O *bloco* recentemente liberado antes e após blocos livres.

Para saber se um bloco liberado está adjacente a quaisquer dos blocos livres na **lista disponível**, usamos o tamanho do bloco. Para mesclar dois blocos, o campo SIZE do *bloco* mais baixo, que seu endereço conhecido, é simplesmente atualizado para conter a soma dos tamanhos dos blocos combinados.

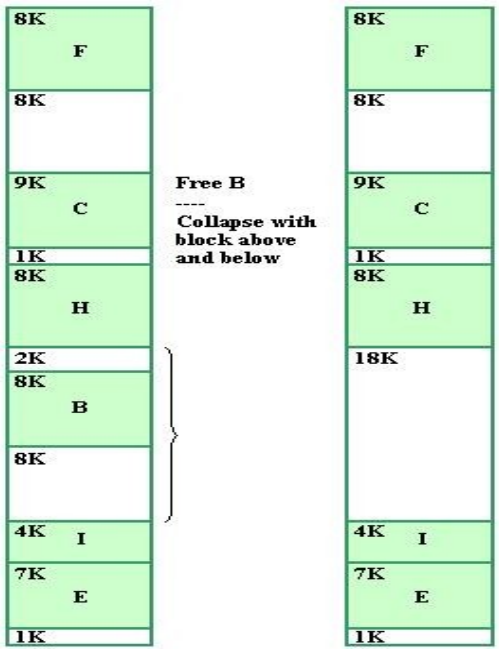


Figura 24. Exemplos de Liberação

5.3.2. A Técnica *Boundary-Tag*

A técnica ***Boundary-Tag*** utiliza duas *words* de controle e uma dupla ligação. A primeira e a última palavra contém detalhes de controle. A figura seguinte mostra a estrutura de ligação e os dois estados de um *bloco* (reservado e livre):

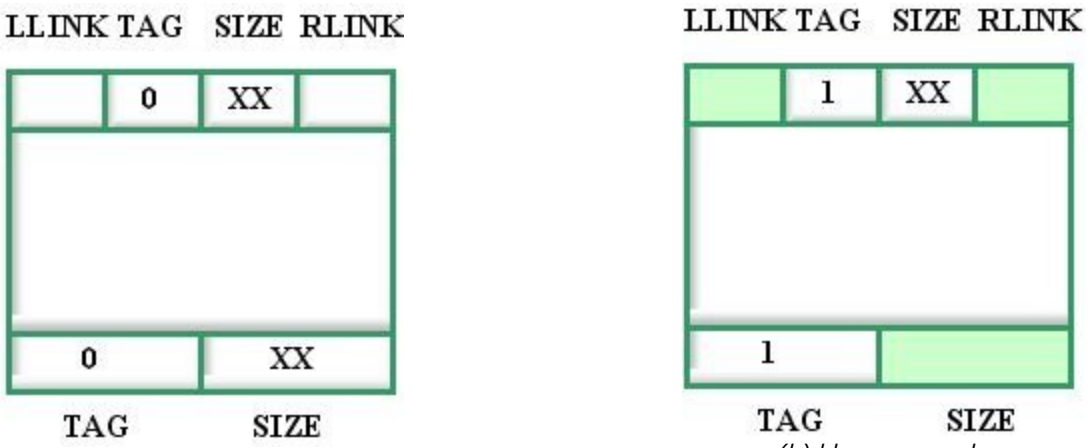


Figura 25. node de estrutura na técnica Boundary-Tag

O valor da TAG é 0 se o bloco está livre, de outra maneira será 1. Ambos os campos TAG e SIZE estão presentes para mesclar blocos livres executados no tempo O(1).

A *lista disponível* é concebida como uma lista ***doubly-linked*** como a lista principal (de cabeçalho). Inicialmente, a *lista disponível* é formada por apenas um bloco, o *memory pool* inteiro, limitado abaixo e acima pelos blocos de memória não disponíveis para DMA. A figura seguinte mostra o estado inicial de uma lista disponível:

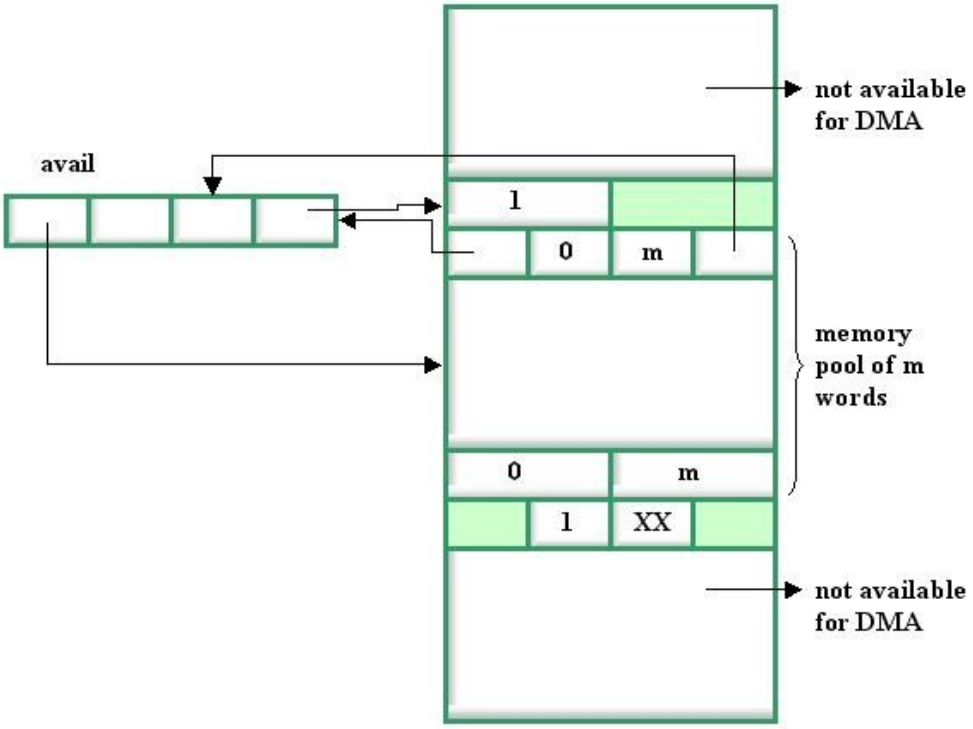


Figura 26. O estado inicial do Memory Pool

Após usar a memória por algum tempo, ficará com segmentos descontínuos, assim teremos a seguinte lista disponível:

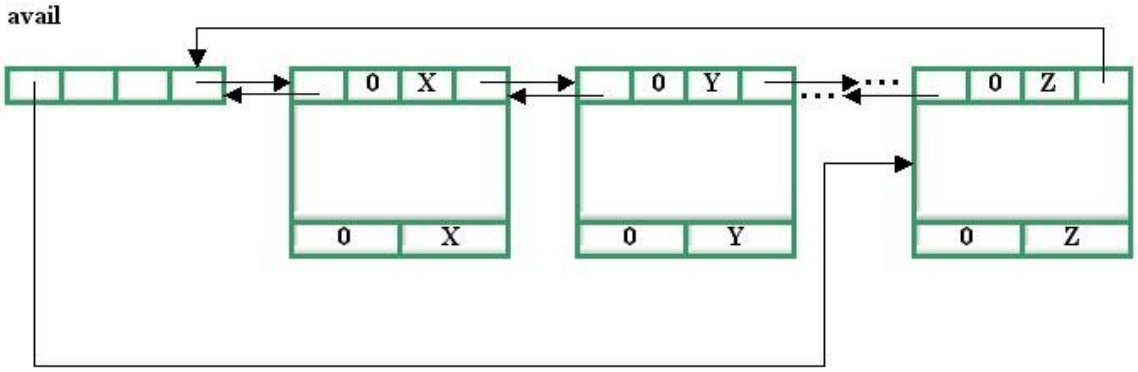


Figura 27. lista disponível após algumas alocações e desalocações

A técnica *Sorted-list* está em O(m), onde m é o número de *blocos* na lista disponível. A técnica ***Boundary-tag*** tem tempo de complexidade O(1).

5.4. Método Buddy-System

Nos métodos *buddy-system*, *blocos* estão alocados em tamanhos *quantum*. Algumas listas disponíveis são mantidas, uma para cada tamanho permitido. Existem dois métodos *buddy-system*:

- O método binary *buddy-system* – os *blocos* são alocados em tamanhos baseados nas potências de 2: 1, 2, 4, 8, …, 2^k *words*
- o método *Fibonacci buddy-system* – os *blocos* são alocados em tamanhos baseados na seqüência numérica *Fibonacci*: 1, 2, 3, 5, 8, 13, … palavras (k-1)+(k-2)

Nesta sessão, iremos cobrir apenas o método ***binary buddy-system***.

5.4.1. O método *Binary Buddy-System*

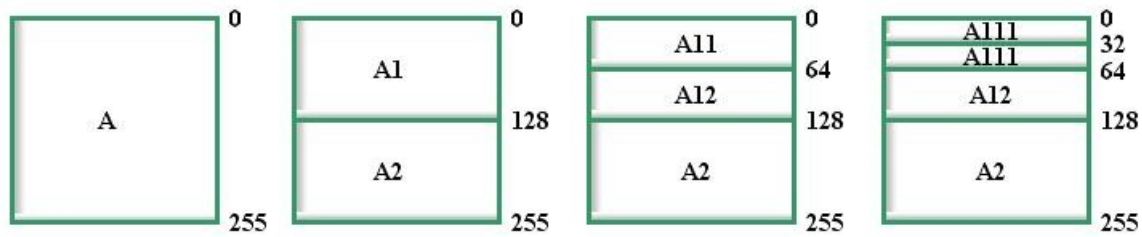


Figura 28. buddies no Binary Buddy-System

5.4.2. Reserva

Dado um bloco com tamanho 2^k, o que segue é o algoritmo para reservar um bloco para um pedido para n *words*:

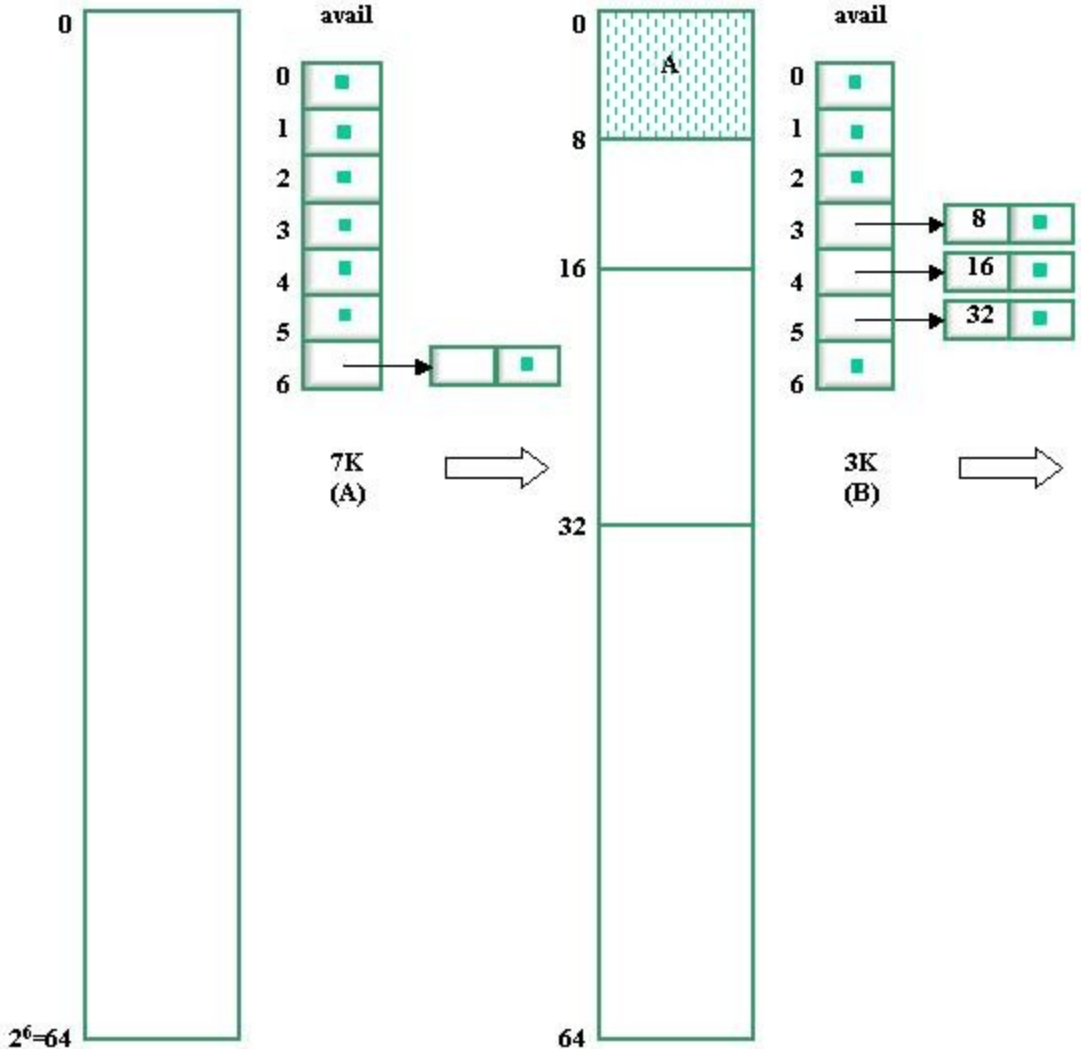
1. Se o tamanho do bloco atual é < n:

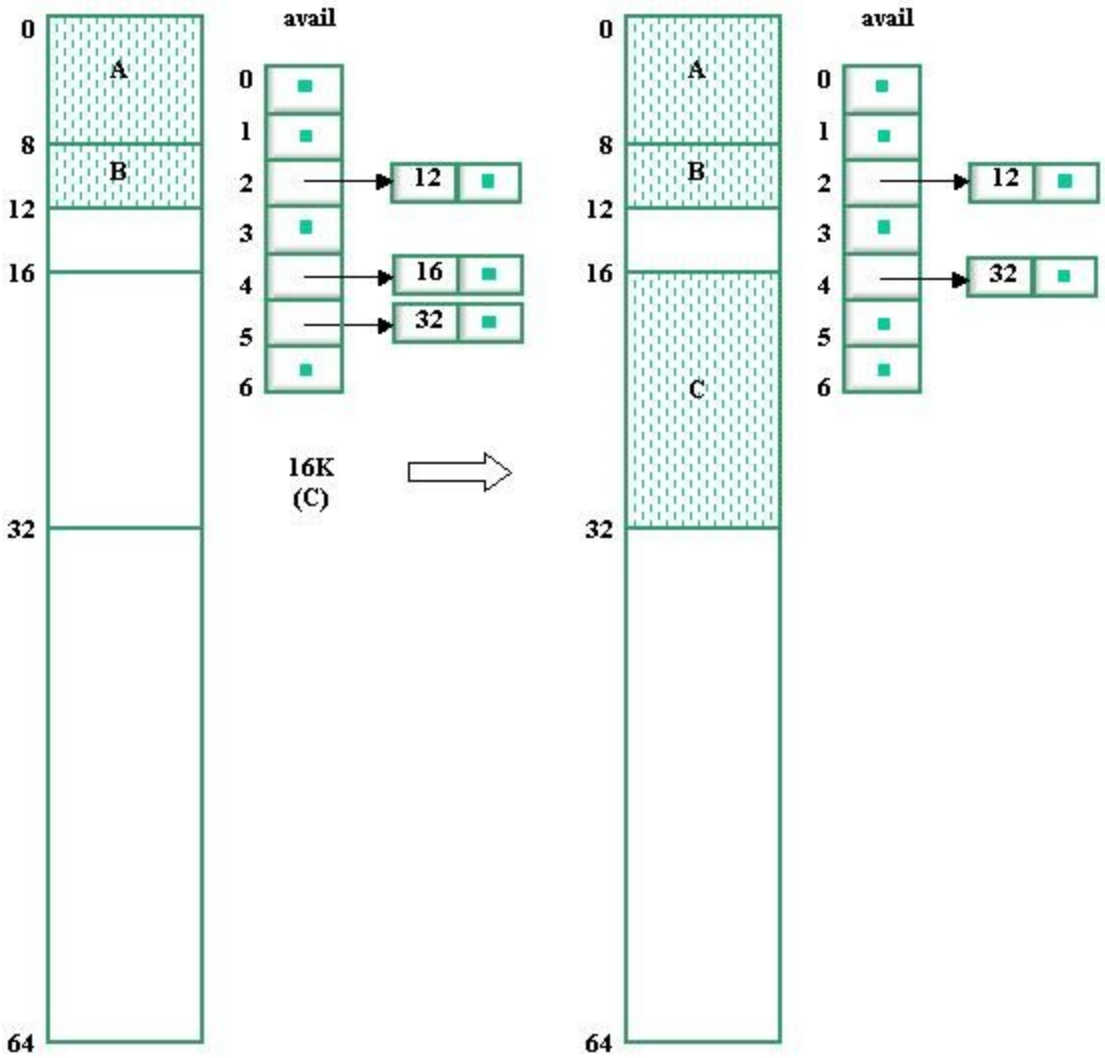
   - Se o bloco atual é o de maior tamanho, retorna: um bloco não suficientemente grande está disponível
   - Caso contrário vai para a lista disponível do próximo bloco no tamanho. Vai para 1
   - Caso contrário, vai para 2

2. Se o tamanho do bloco é o menor múltiplo de 2 ≥ n, então retorna o bloco reservado para a tarefa de requisição
   Caso contrário vai para 3

3. Divide o bloco em duas partes. Estas duas partes são chamados ***buddies***. Vai para 2, tendo a metade superior dos novos limite de corte como o bloco corrente

Por exemplo, reserve espaço para os pedidos A (7K), B (3K), C (16K), D (8K), e E (15K) a partir de um *memory pool* não usado de tamanho 64K.




0
A
8
B
12
16
32
64
avail
0
1
2
3
4
5
6
12
16
32
16K
(C)
0
A
8
B
12
16
C
32
64
avail
0
1
2
3
4
5
6
12
32

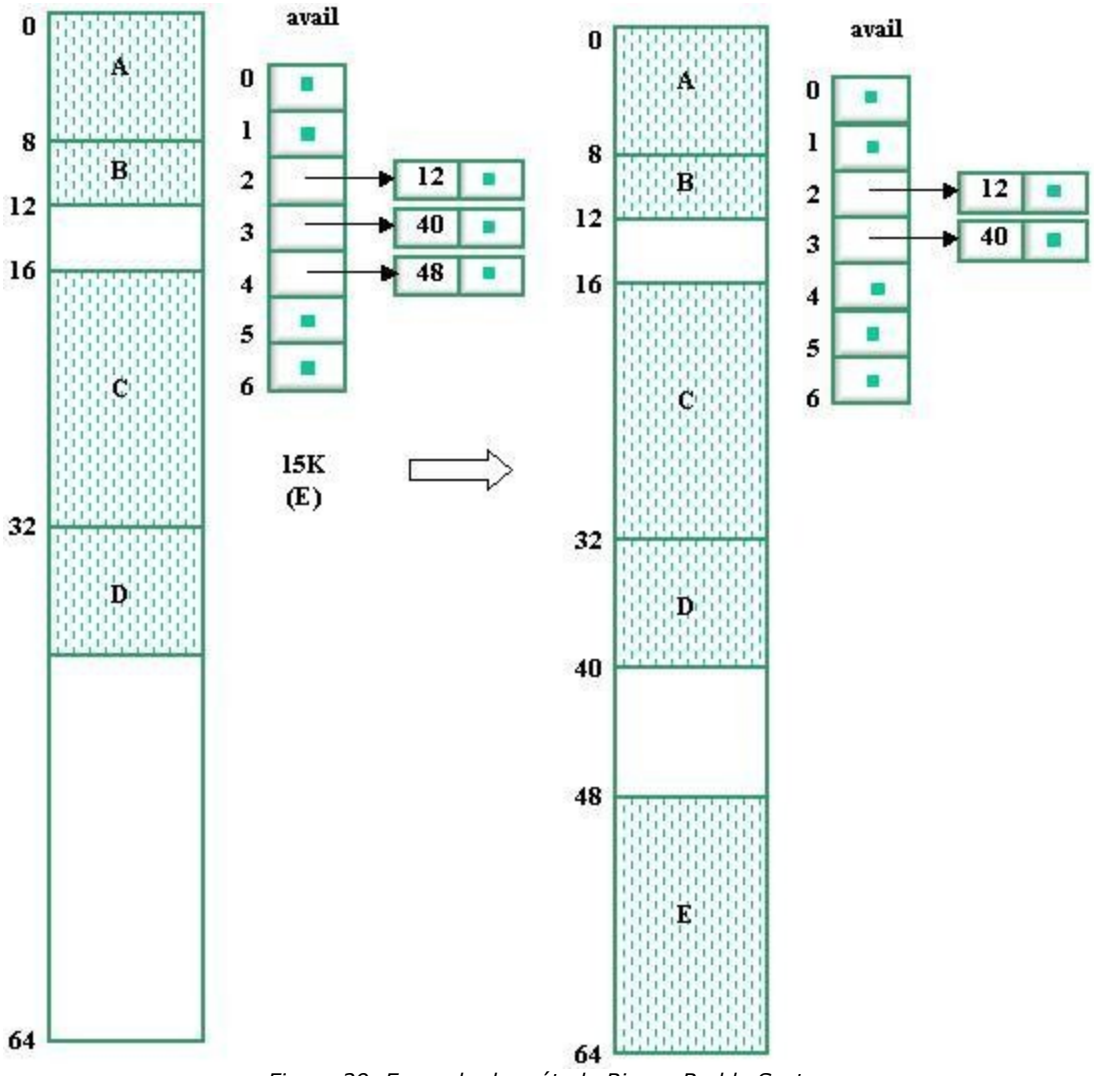


Figura 29. Exemplo de método Binary Buddy System

5.4.3. Liberação

Quando um bloco está liberado de uma tarefa e se o *buddy* do bloco inicialmente liberado está livre, é necessário mesclar os *buddies*. Quando o *buddy* dos blocos recentemente mesclados também está livre, executa-se novamente uma mescla. Isto é feito repetidamente até que mais nenhum *buddy* possa ser mesclado.

Localizar o *buddy* é um passo crucial na operação de **liberação** e é feito pela computação:

Deixe $\beta(k:\alpha)$ = endereço do *buddy* do bloco de tamanho 2^k no endereço α

$\beta(k:\alpha) = \alpha + 2^k$ se $\alpha \bmod 2^{k+1} = 0$

$\beta(k:\alpha) = \alpha - 2^k$ de outro modo.

Se o *buddy* localizado estiver livre, ele pode ser mesclado com o bloco mais recentemente liberado. Para o método *buddy-system* ser eficiente, é necessário manter uma lista disponível para cada

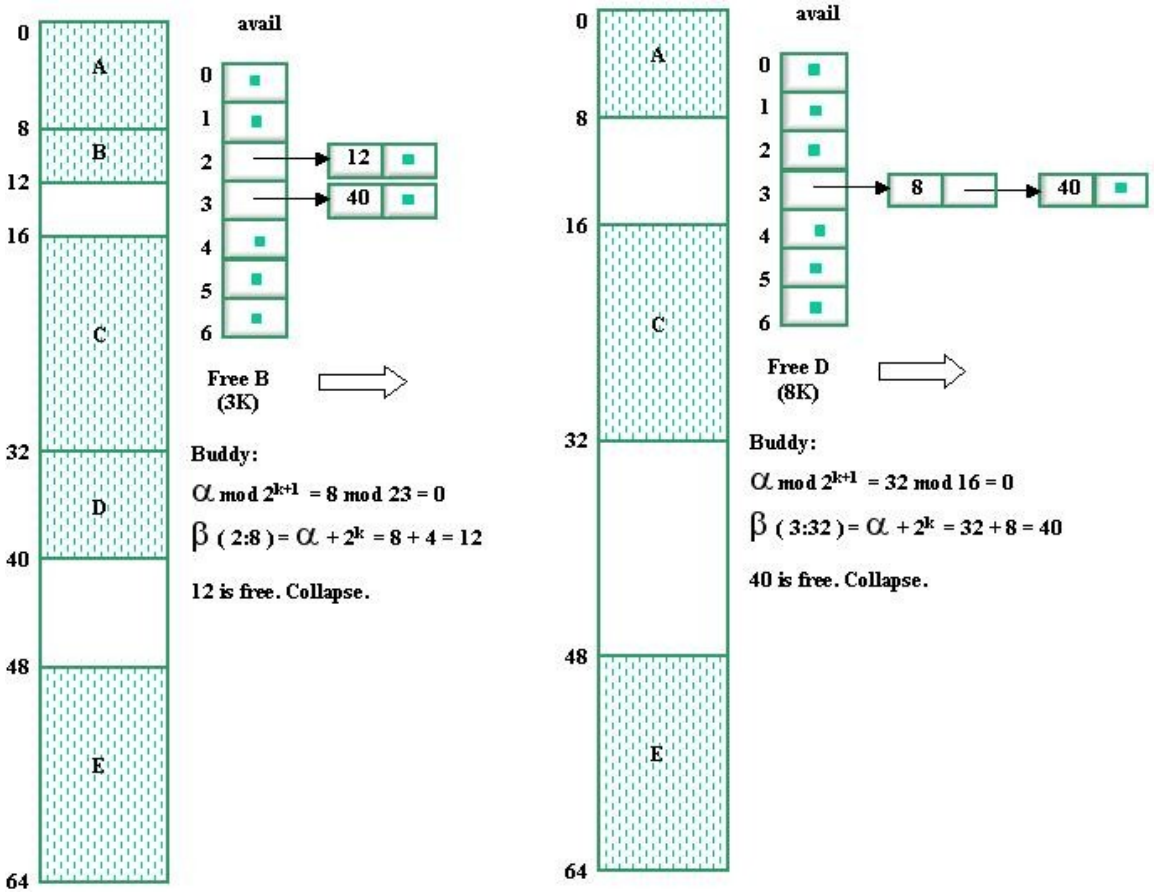
tamanho alocável. A seguir temos o algoritmo para a **reserva** usando o método *binary buddy system*:

1. Se um pedido para n *words* é feito e a lista disponível para blocos do tamanho 2^k, onde $k = \lceil \log_2 n \rceil$, não está vazio, então temos um bloco da lista disponível. De outra forma, vá para 2

2. Obtenha um bloco da lista disponível de tamanho 2^p onde p é o menor inteiro maior que k para que a lista não seja vazia

3. **Dividir** o bloco p-k vezes, inserindo blocos não usados em suas respectivas listas disponíveis

Usando a alocação anterior como nosso exemplo, liberar os blocos com reserva B (3K), D (8K), e A (7K) nesta ordem.



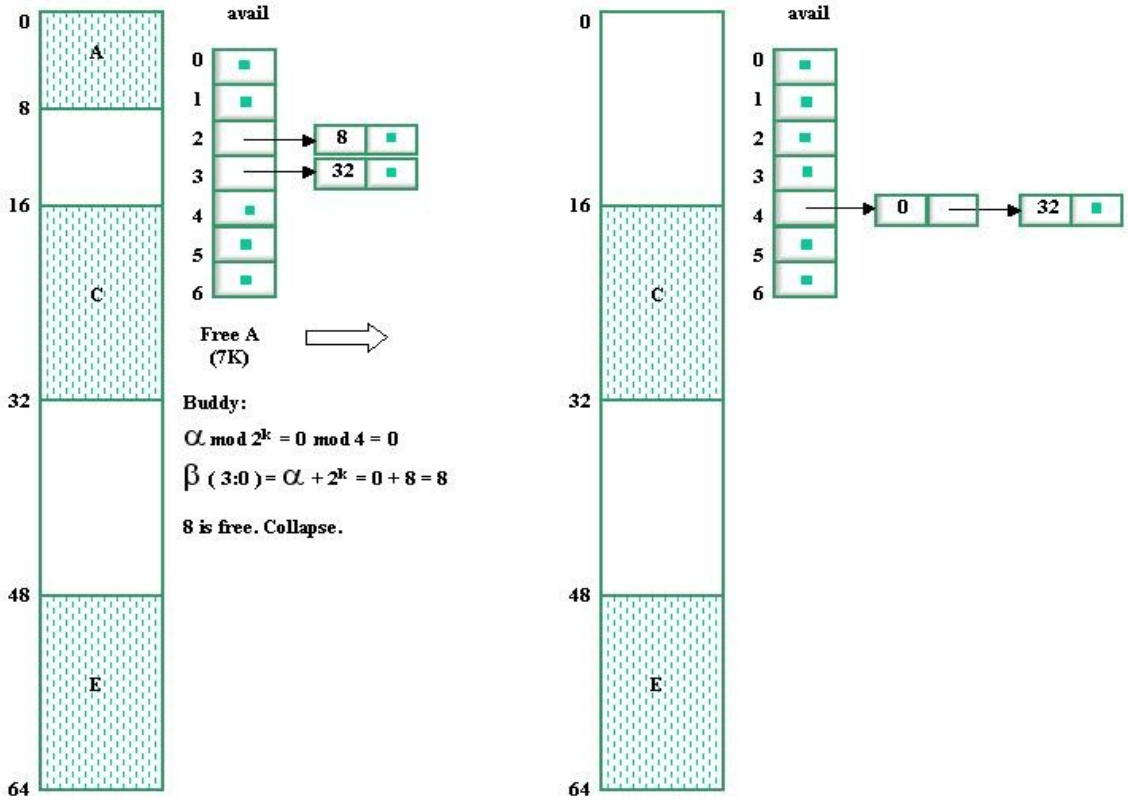


Figura 30. Método Binary Buddy System Exemplo de Liberação

Na implementação, a seguinte estrutura, uma lista *doubly-linked*, será usada para as listas disponíveis:

LLINK	TAG	KVAL	RLINK

TAG = 0 se o bloco estiver livre
1 se o bloco estiver liberado
KVAL = k se o bloco tem tamanho 2^k

Para inicializar o *memory pool* para o método *buddy-system*, assumimos que seu tamanho é $\mathbf{2^m}$. É necessário manter **m+1** listas. Os ponteiros **avail(0:m)** para as listas são armazenados em um *array* de tamanho m+1.

5.5. Fragmentação interna e externa na DMA

Após as séries de **reserva** e **divisão**, blocos de tamanho muito pequeno para satisfazer qualquer pedido irão sobrar na **lista disponível**. Sendo muito pequenos, eles terão pouca chance de serem reservados, e, eventualmente serão dispersados na **lista disponível**. Isto irá resultar em buscas muito longas. Além disso, mesmo que a soma dos tamanhos resulte em um valor que possa satisfazer um pedido, eles não poderão ser utilizados se estiverem dispersos no ***memory pool***. Isto é o que chamamos de ***external fragmentation***. Resolvemos este problema com métodos ***sequential-fit*** usando *minsize*, onde um *bloco* inteiro é reservado se o que sobrar na alocação for menor que o valor especificado. Durante a **liberação**, os *blocos* são dirigidos à uma mescla livre com blocos adjacentes.

A abordagem de usar *minsize* em método ***sequential-fit*** ou arredondar os pedidos para $2^{\lceil \log 2n \rceil}$ no método ***binary buddy-system***, ligações para '*overallocation*' de espaço. Esta alocação de espaço maior que a necessidade para tarefas é o que chamamos de ***internal fragmentation***.

6. Exercícios

1. Mostre a representação de ligação circular do polinômio

   $P(x,y,z) = 2x^2y^2z^2 + x^5z^4 + 10x^4y^2z^3 + 12x^4y^5z^6 + 5y^2z + 3y + 21z^2$

2. Mostre a representação da lista dos seguintes polinômios e dê o resultado quando Polynomial.add(P, Q) é executado:

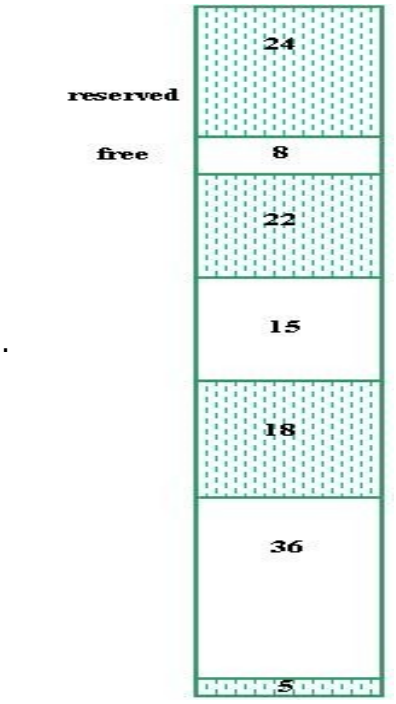
   $P(x, y, z) = 5xyz^2 - 12X^2y^3 + 7xy$
   $Q(x, y, z) = 13xy - 10xyz^2 + 9y^33z^2$

3. Usando métodos *first fit*, *best-fit* e *worst-fit* aloque 15k, 20k, 8k, 5k, e 10k no *memory pool* ilustrado abaixo, tendo minsize = 2:

4. Usando método *binary buddy-system* e dado um *memory pool* vazio de tamanho 128K, reserve espaço para os seguintes pedidos:

Tarefa	Pedido
A	30K
B	21K
C	13K
D	7K
E	14K

Mostre o estado do ***memory pool*** após cada *allocation*. Não é necessário mostrar as **listas disponíveis**. Libere C, E e D nesta ordem. Execute a mescla se necessário. Mostre como os *buddies* são obtidos



6.1. Exercícios para Programar

1. Crie uma interface Java contendo as operações *insert*, *delete*, *isEmpty*, *size*, *update*, *append two lists* e *search*.

2. Escreva uma definição de classe Java que implemente a interface criada no exercício 1 utilizando uma lista linear *doubly-linked*.

3. Escreva uma interface Java que faça uso de um *node* para criar uma lista generalizada.

4. Pela implementação da interface criada no exercício 1, crie uma classe para:

    a) uma lista *singly-linked*

    b) uma *circular-list*

    c) uma lista *doubly-linked*

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.