Gerenciamento de Memória

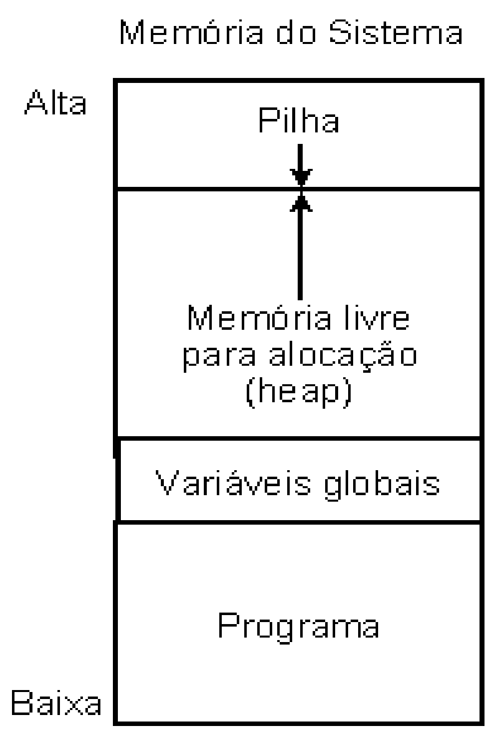
Referências:

A.V.Aho, J.E.Hopcroft, J.D.Ullman, Data Structures and Algorithms, Cap. 12.

H.Schildt, C Completo e Total, Cap. 16.

- Em linguagens como Fortran, Cobol e Basic, o compilador determina quanta memória é necessária para operar os programas.
- Diversas linguagens, como C e Pascal, permitem a *alocação dinâmica* de memória.

  Extremamente útil quando não se sabe quanta memória será necessária (p. ex. planilhas eletrônicas).
- *Heap:* é a região da memória principal a partir da qual porções de memória são dinamicamente alocadas sob solicitações de um programa.



- Quando uma função de alocação é chamada, um bloco de bytes consecutivos (com o tamanho determinado na chamada) é selecionado, e retorna-se o ponteiro para o primeiro byte daquele bloco.
- Alocação/liberação de memória Na linguagem C: <stdlib.h>

  void *calloc(size_t num, size_t size);

  void *malloc(size_t size);

  void free(void *ptr);

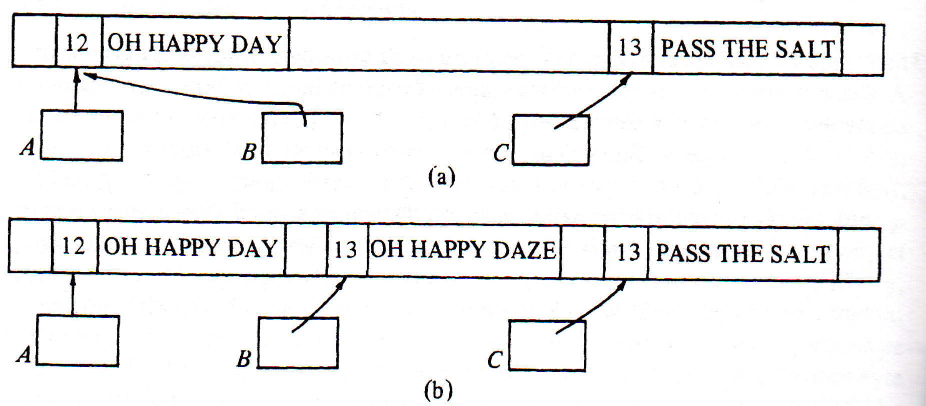
  Exemplo:

```
#include <stdlib.h>

int main() {
  long *matriculas;
  int quant_alunos;

  printf("Quantidade de alunos matriculados: \n");
  scanf("%d", &quant_alunos);

  printf("Alocando espaco... \n");
  matriculas = malloc(quant_alunos*sizeof(long));
  // alternativa:
  // matriculas = calloc (quant_alunos, sizeof(long));

  printf("Liberando espaco... \n");
  free(matriculas);
}
```

- Problema: gerenciar bytes livres da memória.

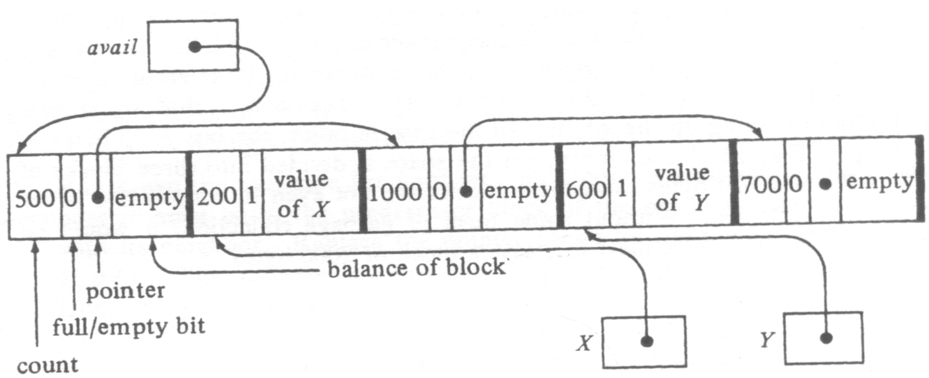
Gerenciamento de Blocos de Tamanho Variável

- Consideraremos o gerenciamento de um *heap*, no qual temos uma coleção de ponteiros para blocos alocados. Os blocos armazenam dados de algum tipo.



- Podemos imaginar que as regiões vazias são ligadas entre si como sugerido na figura abaixo.

  Ali vemos um heap de 3000 palavras dividido em cinco blocos. Dois blocos de 200 e 600 palavras armazenam os valores de X e Y. Os três blocos remanescentes estão vazios, e são ligados por uma cadeia partindo de *avail*, o cabeçalho para o espaço disponível.



- Para que se possa encontrar blocos vazios para armazenamento de novos dados, e para a liberação no heap de blocos que não serão mais utilizados, as seguintes premissas são assumidas:

  1. Cada bloco é suficientemente grande para armazenar:

     a) Um campo referente ao tamanho do bloco (em bytes ou palavras, de acordo com o que for mais apropriado para o sistema);

     b) Um ponteiro para ligar o bloco ao espaço disponível;

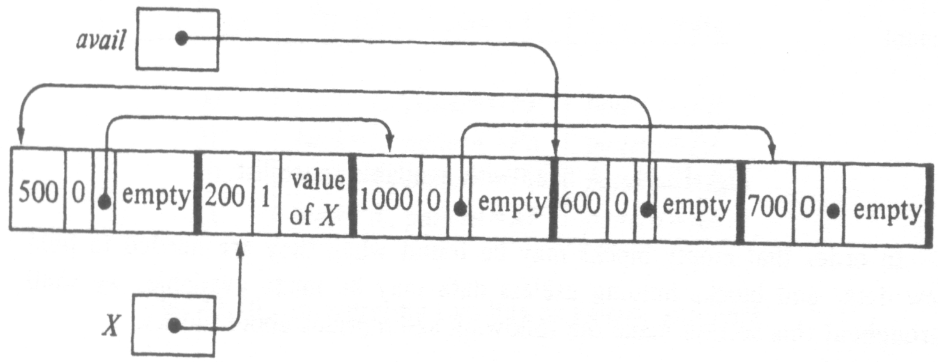
     c) Um bit *cheio/vazio* para indicar se um bloco está em uso ou vazio. Ao longo do texto, os termos "bloco usado" e "bloco cheio" serão adotados como sinônimos.

  2. Um bloco vazio tem, à esquerda (extremidade de endereço mais baixo) um campo para seu tamanho, um bit *cheio/vazio* com valor 0 (indicando que o bloco está vazio), um ponteiro para o próximo bloco disponível, e o espaço vazio para uso futuro.

  3. Um bloco contendo dados tem, à esquerda, um campo para o tamanho, um bit *cheio/vazio* com valor 1 (indicando que o bloco está em uso), e os dados propriamente ditos.

Fragmentação e compactação de blocos livres

- Suponha que a variável Y do exemplo anterior seja liberada, e assim o bloco apontado por Y necessita ser devolvido para o espaço disponível. A forma mais simples é inserir o novo bloco liberado no início da lista $avail$, como sugerido na figura abaixo.



- *Fragmentação:* tendência de grandes áreas vazias serem representados na lista de espaço disponível por "fragmentos", ou seja, diversos blocos pequenos constituindo o espaço todo.

  No exemplo acima, temos 2300 bytes livres, porém divididos em blocos de 1000, 600 e 700 bytes (não consecutivos)

  Sem uma forma de *garbage collection* (coleta de lixo), seria impossível alocar um bloco de 2000 bytes, por exemplo.

- No momento de retornar um bloco para o espaço disponível, seria desejável verificar os blocos imediatamente à esquerda e à direita do bloco liberado, combinando-os entre si se estiverem vazios.

- **Combinando o bloco liberado com o da direita:**

  Se o bloco sendo liberado começa na posição p e tem contagem c, o bloco à direita começa na posição $p + c$. Se o bloco à direita estiver vazio, os blocos começando na posição p e $p+c$ podem ser combinados.

  Para manter a lista de blocos disponíveis após combinar os blocos, deve-se lembrar que o segundo bloco ainda estará ligado na lista, e precisa ser removido. Fazer isso requer encontrar o ponteiro para aquele bloco a partir de seu predecessor na lista de blocos disponíveis. Três estratégias são possíveis:

1. Percorra a lista de disponíveis até encontrar um ponteiro com valor $p + c$. Este ponteiro estará no bloco predecessor do bloco recém-combinado. Substitua o ponteiro encontrado pelo ponteiro localizado no bloco em $p + c$.

   Em média, será necessário percorrer metade da lista de blocos disponíveis para encontrar o ponteiro $\Rightarrow$ tempo proporcional ao tamanho da lista.

2. Use uma lista duplamente ligada de espaço disponível.

   Esta estratégia leva tempo constante, mas não serve para se encontrar um bloco vizinho à esquerda do bloco sendo liberado.

3. Mantenha a lista de espaço disponível ordenada por posição. Com esta estratégia, após a inserção do bloco com endereço p na lista, saberemos que este é o predecessor do bloco em $p + c$, e a manipulação de ponteiros necessária para eliminar o segundo bloco pode ser feita em tempo constante.

   Em média, será necessário percorrer metade da lista de blocos disponíveis para cada inserção $\Rightarrow$ tempo proporcional ao tamanho da lista.

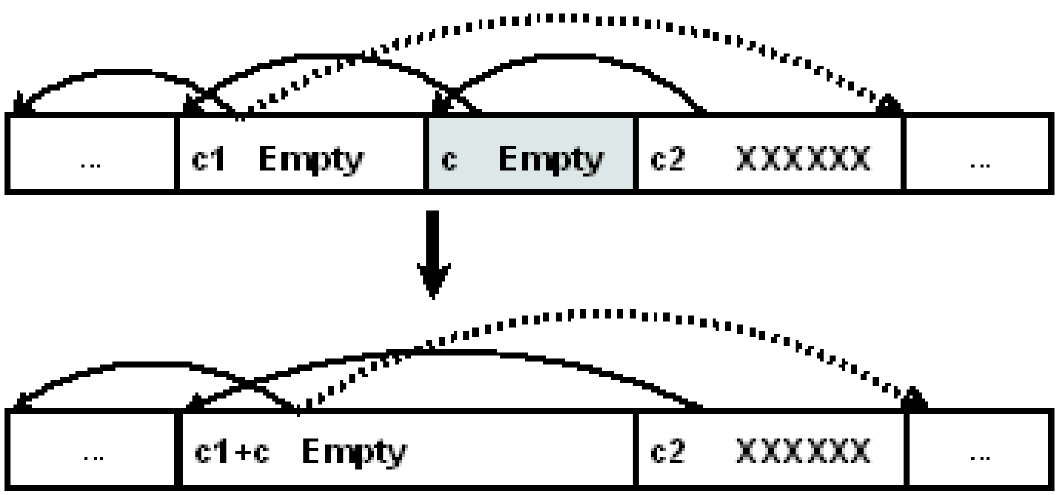
- **Combinando o bloco liberado com o da esquerda:**

  Precisamos encontrar um bloco vazio que começa em uma posição p_1, e tem uma contagem c_1 tal que $p_1 + c_1 = p$. Três estratégias são possíveis:

  1. Procure na lista de disponíveis por um bloco na posição p_1 e com contagem c_1 tal que $p_1 + c_1 = p$. Esta operação leva tempo proporcional ao tamanho da lista de disponíveis.

  2. Mantenha um ponteiro em cada bloco (usado ou ocioso) indicando a posição do bloco à esquerda. Este ponteiro é facilmente con-

figurado no momento de alocar o bloco pela primeira vez (basta manter-se um ponteiro para o último bloco de memória alocado).
Esta estratégia permite encontrar o bloco à esquerda em tempo constante; podemos verificar se ele está vazio e, em caso afirmativo, podemos combiná-lo com o bloco que está sendo liberado. Podemos encontrar o bloco na posição $p + c$ e fazê-lo apontar para o início do novo bloco, mantendo seu ponteiro à esquerda corretamente atualizado.
Desvantagem: Necessidade de espaço extra para o ponteiro à esquerda, independentemente do bloco estar ou não vazio. Quanto maior o tamanho médio do bloco, menos significante o custo relativo desse ponteiro.



3. Mantenha a lista de espaço disponível ordenada por posição. Então o bloco à esquerda é encontrado quando inserimos o novo bloco na lista, e temos apenas que verificar, usando a posição e contagem do bloco vazio anterior, se não existem blocos não-vazios entre os dois.
Em média, será necessário percorrer metade da lista de blocos disponíveis para cada inserção $\Rightarrow$ tempo proporcional ao tamanho da lista.

- **Estratégias gerais para tratar a fragmentação:** Para sumarizar as implicações das idéias apresentadas acima para tratar a questão de como se pode combinar blocos recém liberados com os vizinhos vazios, são apresentadas abaixo três estratégias para tratar a fragmentação.

  1. Use uma dentre várias abordagens, tais como manter a lista de disponíveis ordenada, o que requer tempo proporcional ao tamanho da lista de disponíveis cada vez que um bloco se torna ocioso, mas nos permite encontrar e combinar vizinhos vazios.
  2. Use uma lista de disponíveis duplamente ligada, e também use um ponteiro para o vizinho esquerdo em todos os blocos (disponíveis ou não), para combinar blocos vizinhos vazios em tempo constante.
  3. Não faça nada explicitamente para combinar vizinhos vazios. Quando não puder encontrar um bloco grande o suficiente para armazenar um novo item de dados, percorra os blocos da esquerda para a direita, combinando vizinhos vazios e então criando uma nova lista de disponíveis. Vide programa abaixo.

- À medida em que o número total de blocos e o número de blocos disponíveis tendem a não ser muito diferentes (especialmente em programas com muitas alocações/liberações de memória), e a freqüência com que um bloco vazio suficientemente grande não possa ser encontrado tende a ser baixa, Aho et al consideram o método 3 melhor do que o método 1 em situações práticas. O método 2 é um competidor possível, mas considerando o espaço necessário e o tempo adicional gasto a cada inserção ou remoção da lista de disponíveis, é considerado menos eficiente do que o método 3.

```
tipo bloco:
  vazio: booleano indicando se o bloco está vazio (V) ou cheio (F)
  tam: tamanho do bloco

constante ENDERECO_MAX;   // endereco maximo no heap

procedimento merge:
variaveis:
   p, q: ponteiros para blocos
   // p aponta para o endereço inicial de um bloco vazio sendo acumulado.
   // q aponta para blocos à direita de p

{
  p = endereço inicial do heap;
  inicialize a lista de blocos disponíveis;
  enquanto p < ENDERECO_MAX {
    se p->vazio == F {   // pule blocos cheios
      p = p + p->tam;
    } senão {      // blocos adjacentes vazios: devem ser fundidos
      q = p + p->tam;  // inicialize q como sendo o proximo bloco
      enquanto q < ENDERECO_MAX & q->vazio == V {
        p->tam = p->tam + q->tam;
        q = q + q->tam;
      }
      insira bloco apontado por p na lista de disponiveis;
      p = q;  // atualiza ponteiro p
    }
  }
}
```

Seleção de blocos disponíveis

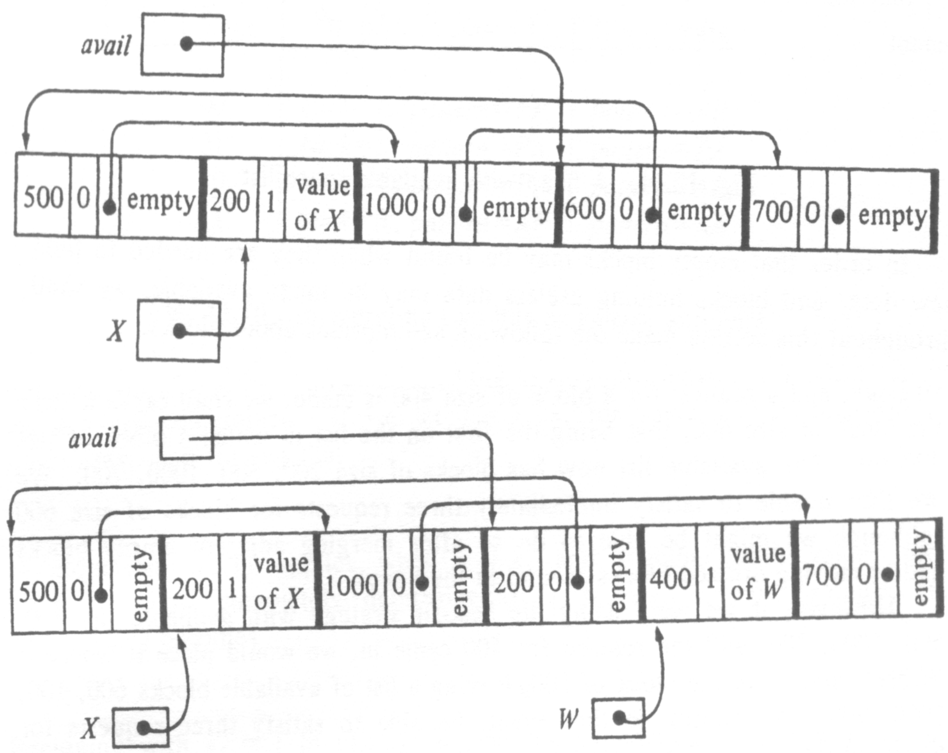
- Quando se necessita fornecer um bloco para armazenar novos dados, há duas questões a serem tratadas:

- Qual bloco vazio deve ser selecionado?

- Se tivermos que usar apenas uma parte do bloco selecionado, qual parte usaremos?

- A segunda questão tem resposta simples: se iremos usar um bloco com contagem c, e precisamos de $d < c$ bytes daquele bloco, escolheremos os últimos d bytes. Dessa forma, precisamos apenas substituir a contagem c por $c - d$, e o bloco vazio restante pode permanecer como está na lista de disponíveis.

  Exemplo:



- Para a escolha do bloco onde armazenar os novos dados, existem objetivos conflitantes a serem considerados:

  - Velocidade para escolha do bloco a ser usado;

  - Minimização da fragmentação.

- Suponha que precisamos de um novo bloco de tamanho d para os novos dados. Duas estratégias extremas para a escolha do bloco são:

  First-fit: Percorre-se a lista de disponíveis do início até encontrar um bloco de tamanho $c \geq d$.

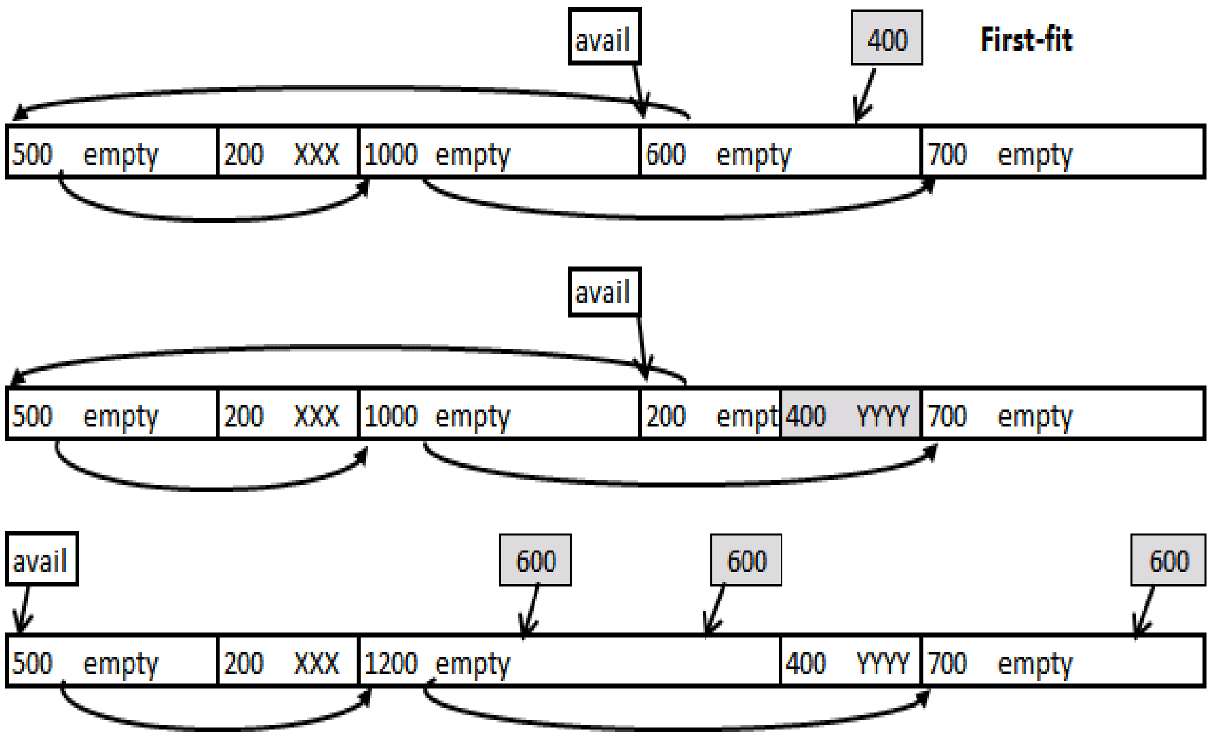
  Best-fit: Examina-se a lista de disponíveis inteira para encontrar o bloco de tamanho $c \geq d$, tal que $c - d$ seja o menor possível.

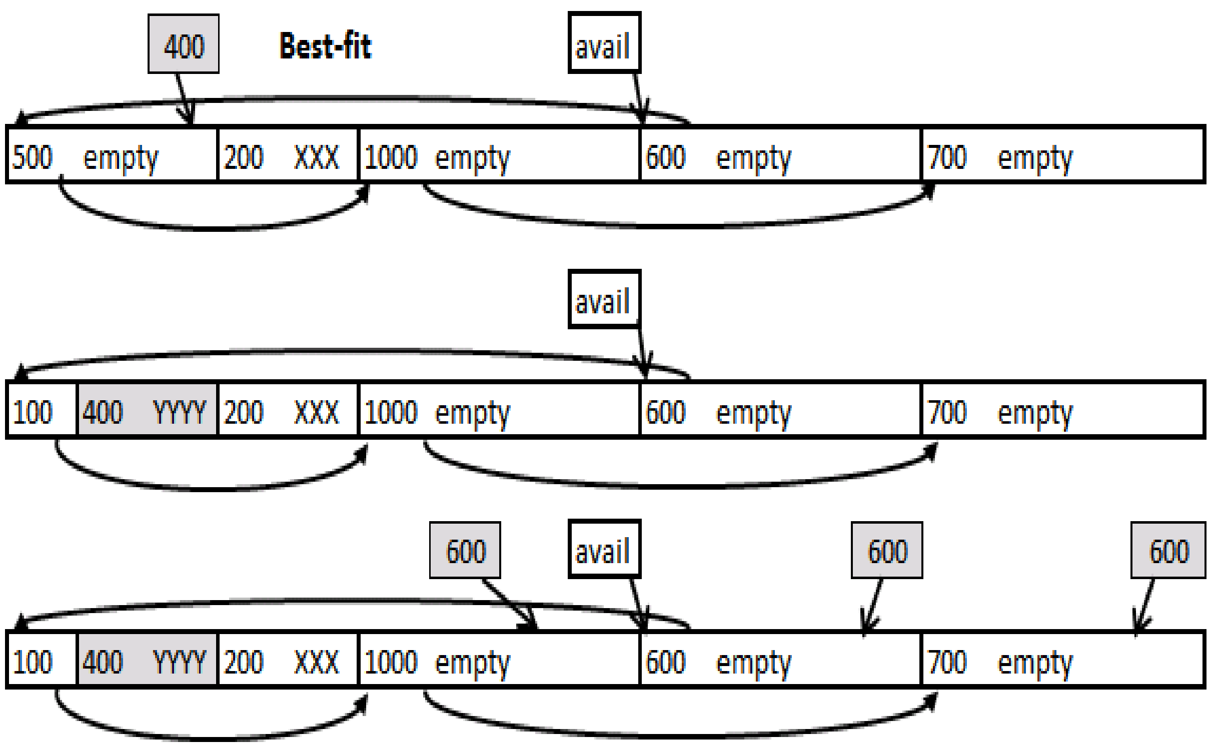
- Em ambas as estratégias, utiliza-se as d últimas palavras do bloco escolhido, conforme descrito anteriormente.

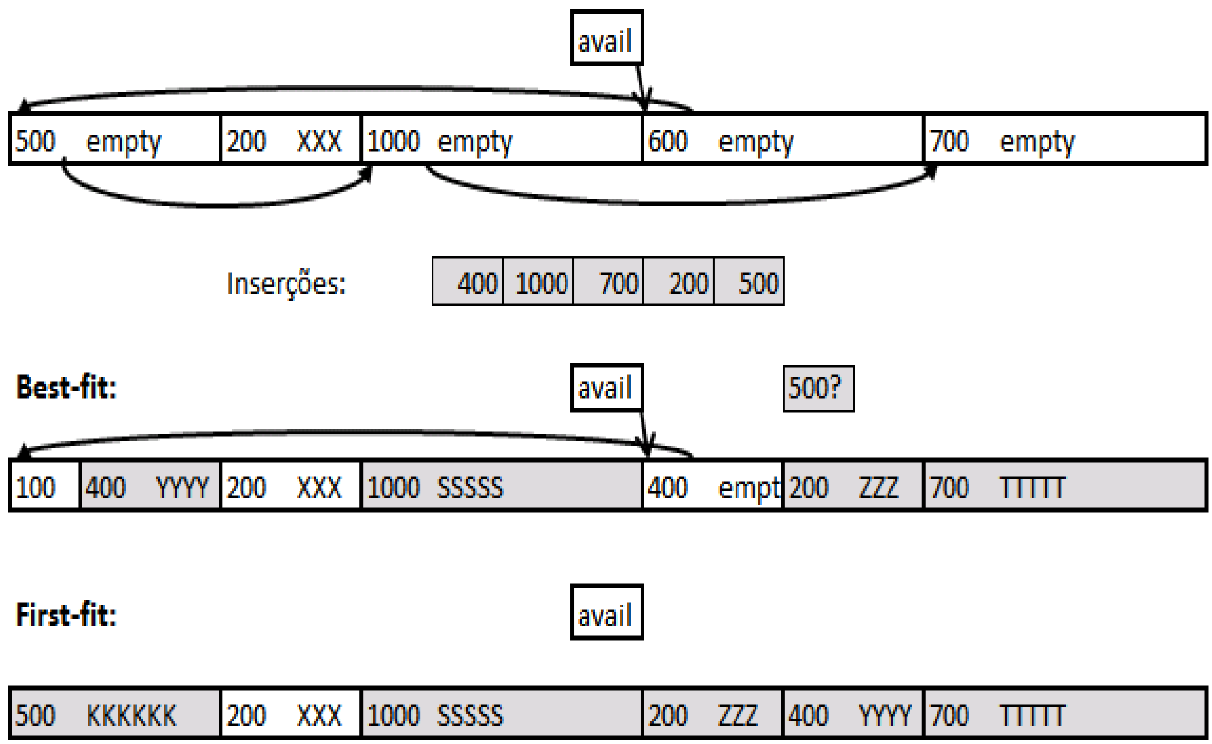
- A estratégia *best-fit* pode ter seu desempenho melhorado se mantivermos listas de blocos disponíveis separadas em várias faixas de tamanhos (por exemplo, blocos de 1–16 bytes; 17–32; 33–64, e assim por diante.

  Na estratégia *first-fit*, manter listas separadas não melhora significativamente o tempo de busca.

- Na estratégia *best-fit*, os blocos livres tendem a ser fragmentos de tamanhos muito pequenos, ou serão blocos retornados para o espaço disponível. Como conseqüência, há seqüências de requisições que a o *first-fit* pode satisfazer mas o *best-fit* não, e vice-versa.


avail
400
First-fit
500 empty
200 XXX
1000 empty
600 empty
700 empty
avail
500 empty
200 XXX
1000 empty
200 empt
400 YYYY
700 empty
avail
600
600
600
500 empty
200 XXX
1200 empty
400 YYYY
700 empty


400
Best-fit
avail
500 empty
200 XXX
1000 empty
600 empty
700 empty
avail
100
400 YYYY
200 XXX
1000 empty
600 empty
700 empty
600
avail
600
600
100
400 YYYY
200 XXX
1000 empty
600 empty
700 empty



Sistemas *Buddy*

- Existe uma família de estratégias para manutenção do heap que evitam parcialmente os problemas de fragmentação e a distribuição indesejável de tamanhos de blocos vazios.

  Essas estratégias, denominadas *sistemas buddy* ("companheiros"), consomem pouco tempo combinando blocos vazios adjacentes.

- Desvantagem: blocos são organizados em um sortimento finito de tamanhos, de forma que desperdiça-se algum espaço colocando um item de dados em um bloco maior do que o necessário.

- A idéia central por trás dos sistemas buddy é que os blocos vêm apenas em certos tamanhos: digamos que $s_1 < s_2 < s_3 < \ldots < s_n$ são todos os tamanhos em que os blocos podem ser encontrados.

Escolhas comuns para a seqüência $s_1, s_2, \ldots, s_n$ são

- $1, 2, 4, 8, \ldots$ - sistema exponencial, onde $s_{i+1} = 2s_i$; e

- $1, 2, 3, 5, 8, 13, \ldots$ - sistema de Fibonacci, onde $s_{i+1} = s_i + s_{i-1}$.

Todos os blocos vazios de tamanho s_i são ligados em uma lista, e há um arranjo de *headers* de listas de disponíveis, um para cada tamanho s_i permitido.

- Se necessitamos de um bloco de tamanho d para um novo dado, escolhemos um bloco disponível do tamanho s_i tal que $s_{i-1} < d \leq s_i$, ou seja, o menor tamanho permitido onde cabe o novo dado.

- Quando não existem blocos vazios do tamanho desejado s_i, encontramos um bloco vazio de tamanho s_{i+1} e o dividimos em dois, um de tamanho s_i e outro de tamanho $s_{i+1} - s_i$.

  (Se não houver blocos vazios de tamanho s_{i+1}, procuramos blocos de tamanhos s_{i+2}, s_{i+3}, etc.)

- O sistema buddy estabelece a restrição de que $s_{i+1} - s_i$ seja algum s_j, para $j \leq i$. Veremos a seguir a maneira em que as escolhas de valores para os s_i's são restringidos. Se permitimos $j = i - k$, para algum $k \geq 0$, então uma vez que $s_{i+1} - s_i = s_{i-k}$, segue que

$$s_{i+1} = s_i + s_{i-k}.$$

- A equação acima aplica-se quando $i > k$, e juntamente com os valores de $s_1, s_2, \ldots, s_k$, determina completamente $s_{i+1}, s_{i+2}, \ldots$.

- Exemplos:

  - Para $k = 0$, temos $s_{i+1} = 2s_i$. Começando com $s_i = 1$, temos a seqüência exponencial $1, 2, 4, 8, \ldots$.

- Para $k = 1, s_1 = 1, s_2 = 1$, a equação acima fica $s_{i+1} = s_i + s_{i-1}$, e temos a seqüência de Fibonacci: $1, 2, 3, 5, 8, 13, \ldots$.

- Para qualquer valor de k escolhido, tem-se um *sistema buddy de k-ésima ordem*. Para todo k, a seqüência de tamanhos permitidos cresce exponencialmente, ou seja, a razão s_{i+1}/s_i se aproxima de alguma constante maior que 1.

  Exemplos:

  - $k = 0 \Rightarrow s_{i+1}/s_i = 2$;

  - $k = 1 \Rightarrow s_{i+1}/s_i \approx (\sqrt{5} + 1)/2 = 1.618$

Distribuição dos blocos

- No sistema buddy de $k-$ésima ordem, cada bloco de tamanho s_{i+1} pode ser visto como consistindo de um bloco de tamanho s_i e um bloco de tamanho s_{i-k}. Por especificidade, suponha que o bloco de tamanho s_i está à esquerda (em posições mais baixas) do bloco de tamanho s_{i-k}. Se vermos o heap como um único bloco de tamanho s_n, para algum n grande, então as posições de início dos blocos de tamanho s_i são completamente determinadas.

- Caso mais simples: sistema exponencial (i.e de $0-$ésima ordem).

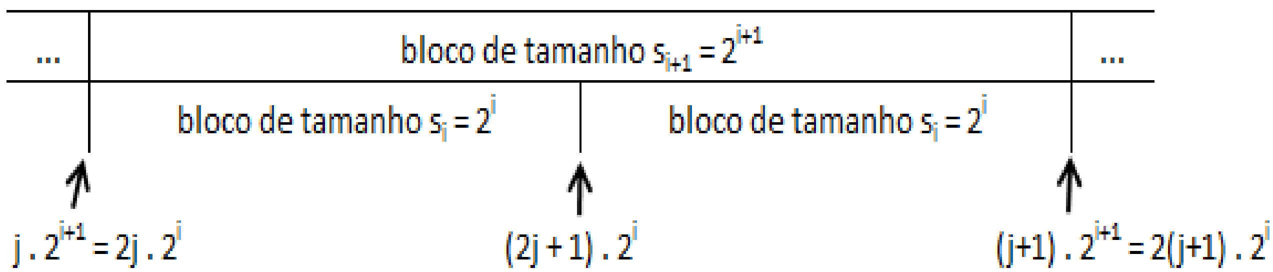
  Assumindo que as posições no heap são enumeradas a partir de 0, um bloco de tamanho s_i começa em qualquer posição começando com um múltiplo de 2^i, ou seja, $0, 2^i, 2.2^i, \ldots$.

  Além disso, cada bloco de tamanho 2^{i+1}, começando em, digamos, $j2^{i+1}$, é composto de dois "companheiros" de tamanho 2^i, os quais começam nas posições $j2^{i+1} = (2j)2^i$, e $j2^{i+1} + 2^i = (2j + 1)2^i$, respectivamente.

Assim, é fácil encontrar o companheiro de um bloco de tamanho 2^i:

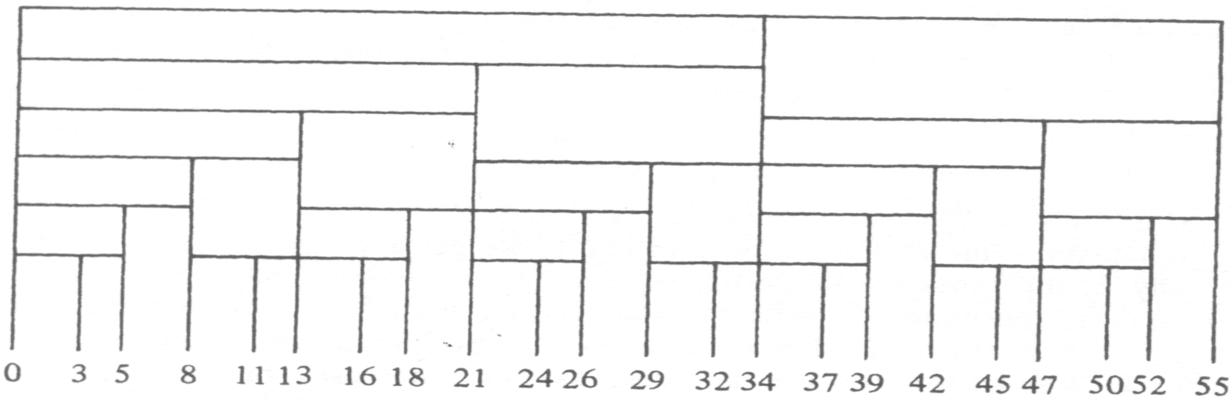
- Se ele começa em algum múltiplo *par* de 2^i, digamos $(2j)2^i$, seu companheiro está à direita, na posição $(2j+1)2^i$.

- Se ele começa em algum múltiplo *ímpar* de 2^i, digamos $(2j+1)2^i$, seu companheiro está à esquerda, na posição $(2j)2^i$.



- Para sistemas buddy de ordem maior do que 0 (p.ex. sistema de Fibonacci), a distribuição dos blocos é um pouco mais complexa.
- A figura abaixo ilustra o sistema de Fibonacci usado em um heap de tamanho 55, com blocos de tamanhos $s_1, s_2, \ldots, s_8 = 2, 3, 5, 8, 13, 21, 34, 55$.

  Por exemplo, o bloco de tamanho 3 começando em 26 é companheiro do bloco de tamanho 5 começando em 21; juntos eles formam o bloco de tamanho 8 começando em 21, que por sua vez é companheiro do bloco de tamanho 5 começando em 29. Juntos, eles formam o bloco de tamanho 13 começando em 21, e assim por diante.



Alocação de blocos nos sistemas buddy

- Quando se necessita de um bloco de tamanho n, escolhe-se qualquer bloco da lista de blocos disponíveis de tamanho s_i tal que $s_i \geq n$ e ou $i = 1$ ou $s_{i-1} < n$. Ou seja, escolhemos o bloco que melhor se ajusta ao tamanho desejado.

- Em um sistema buddy de k-ésima ordem, se não houver blocos de tamanho s_i disponíveis, podemos escolher um bloco de tamanho s_{i+1} ou s_{i+k+1} para dividir, pois em qualquer caso um dos blocos resultantes será de tamanho s_i. Se não houver blocos disponíveis de tamanho s_{i+1} ou s_{i+k+1}, pode-se usar recursivamente esta estratégia de partição para se obter um bloco de tamanho s_{i+1}.

- Observação: em um sistema de k-ésima ordem, pode não ser possível particionar blocos de tamanho $s_1, s_2, \ldots, s_k$, uma vez que sua partição poderia resultar em um bloco de tamanho menor do que s_1. Nesse caso, será necessário utilizar utilizar o bloco inteiro, se não houver blocos de tamanho menor disponíveis.

  - No sistema buddy exponencial, isto não ocorre.

Liberação de blocos para memória disponível

- Quando um bloco se torna disponível para reúso, pode-se reduzir a fragmentação combinando o bloco recentemente liberado com seu parceiro, se este também estiver disponível.

  Se ambos forem combinados, pode-se combinar o bloco resultante com seu vizinho (se esse estiver vazio), e assim por diante.

- A combinação de blocos companheiros tem custo de tempo constante.

- O sistema buddy exponencial torna a localização dos companheiros simples: se o bloco de tamanho 2^i liberado estiver na posição $p2^i$, seu vizinho estará na posição $(p+1)2^i$, se p for par, ou na posição $(p-1)2^i$, se p for ímpar.

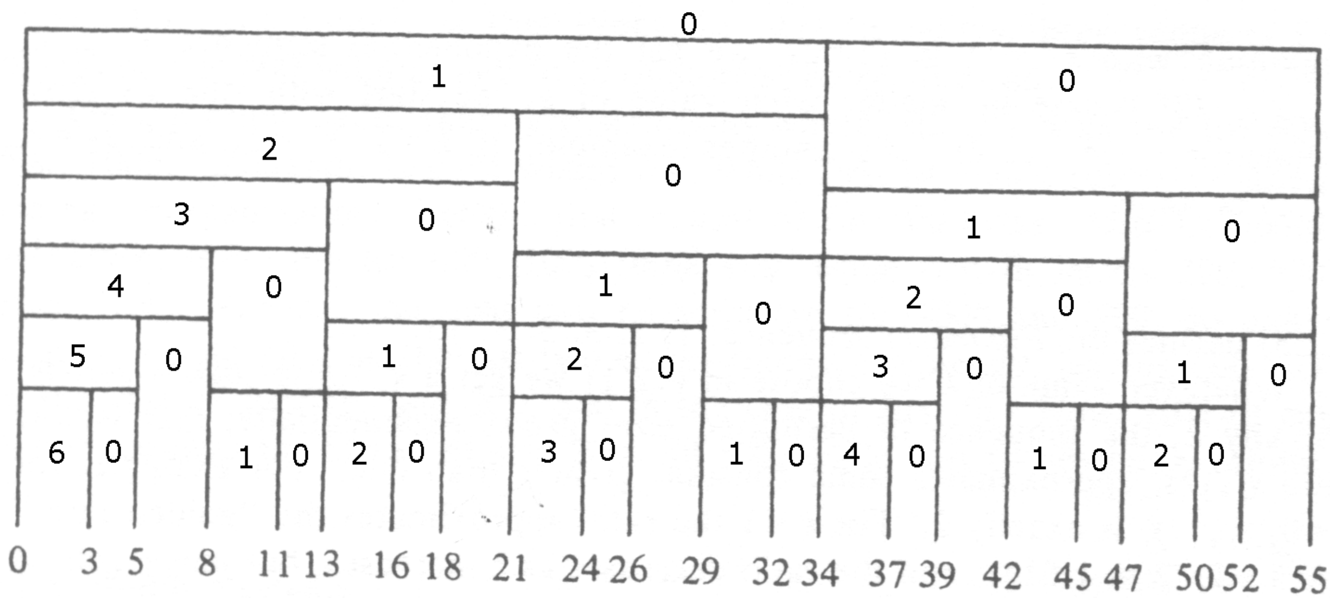
- Em um sistema buddy de ordem $k \geq 1$, para encontrar a busca pelos blocos companheiros, devem ser armazenadas as seguintes informações em cada bloco:

  1. Um bit cheio/vazio;
  2. O *índice de tamanho*, correspondente ao inteiro i tal que o tamanho do bloco é s_i;
  3. A contagem de companheiro esquerdo, descrita abaixo.

- Intuitivamente, a contagem de companheiro esquerdo de um bloco indica quantas vezes consecutivas ele é um companheiro esquerdo ou parte de um companheiros esquerdo.

  Formalmente, o heap inteiro, tratado como um bloco de tamanho s_n, tem uma contagem de companheiro esquerdo igual a 0. Quando dividimos qualquer bloco de tamanho s_{i+1}, com contagem de companheiro esquerdo b, em blocos de tamanho s_i e s_{i-k} - que são os companheiros esquerdo e direito, respectivamente - o companheiro esquerdo recebe uma contagem de companheiro esquerdo igual a $b+1$, enquanto o direito recebe uma contagem de companheiro esquerdo igual a 0, independente de b.

A figura abaixo ilustra o sistema de Fibonacci usado em um heap de tamanho 55, com blocos de tamanhos $s_1, s_2, \ldots, s_8 = 2, 3, 5, 8, 13, 21, 34, 55$. No interior de cada bloco, é mostrada sua contagem de companheiro esquerdo.



- Adicionalmente às informações acima, os blocos vazios possuem ponteiros para o sucessor e antecessor na lista de blocos disponíveis correspondente ao tamanho do bloco.

  Os ponteiros bidirecionais tornam a combinação de companheiros (que requer a deleção desses blocos nas correspondentes listas de disponíveis) mais fácil.

- Seja k a ordem do sistema buddy. Considere um bloco começando na posição p e com índice de tamanho i (ou seja, o bloco tem tamanho s_i).

- Se a contagem de companheiro esquerdo do bloco é 0, então ele é o companheiro direito de um bloco de tamanho s_{i+k} que começa na posição $p - s_{i+k}$.

- Se a contagem de companheiro esquerdo do bloco é maior do que 0, então ele é o companheiro esquerdo de um bloco de tamanho s_{i-k}, que inicia na posição $p + s_i$.

- Se combinamos um companheiro esquerdo de tamanho s_i, contendo uma contagem de companheiro esquerdo de b, com um companheiro direito de tamanho s_{i-k}, o bloco resultante tem índice de tamanho $i + 1$, começa na mesma posição do bloco de tamanho s_i, e tem uma contagem de companheiro esquerdo igual a $b - 1$.

  Dessa forma, toda a informação necessária pode ser facilmente mantida quando combinamos dois companheiros vazios.

  (Note que a informação pode ser atualizada quando dividimos um bloco vazio de tamanho s_{i+1} em dois blocos vazios de tamanhos s_i e s_{i-k}.)

- Se mantivermos toda essa informação, e ligarmos as listas de disponíveis em ambas as direções, cada divisão de um bloco em dois companheiros e cada união de dois blocos companheiros em um bloco terá custo de tempo constante.

  Uma vez que o número de uniões entre blocos nunca pode exceder o número de divisões, o custo total é proporcional ao número de divisões.

- Em geral, a maioria das requisições por um bloco não deve necessitar de nenhuma divisão, uma vez que um bloco do tamanho correto deverá estar disponível.

Pior caso: quando repetidamente requisitamos um bloco do menor tamanho possível, e em seguida o liberamos. Se houver n tamanhos diferentes, precisaremos de pelo menos $n/(k+1)$ divisões em um sistema buddy de ordem k, que serão seguidas por $n/(k+1)$ uniões quando o bloco for retornado.

Compactação da Memória

- Existem situações em que, mesmo depois de combinar todos os blocos adjacentes vazios, não é possível atender uma requisição por um novo bloco:

  a) Espaço disponível no heap é menor do que o espaço requisitado.

  b) Situação mais comum: a soma dos espaços disponíveis nos blocos livres é maior do que o espaço requisitado, mas não há nenhum bloco livre com tamanho suficiente. (O espaço disponível está dividido entre vários blocos não-contíguos.)

- Há duas abordagens gerais para o problema b:

  1. Faça com que o espaço disponível para um dado possa ser composto em diversos blocos vazios. Se isso for possível, pode-se também exigir que todos os blocos sejam de mesmo tamanho e contenham espaço para um ponteiro e espaço para os dados. Em cada bloco usado, o ponteiro indica o próximo bloco contendo o dado.

  2. Quando a combinação de blocos vizinhos adjacentes não é capaz de fornecer um bloco suficientementemente grande, mova os dados do heap de forma que blocos usados estejam na extremidade

esquerda (baixas posições), e que haja um grande bloco disponível à direita.

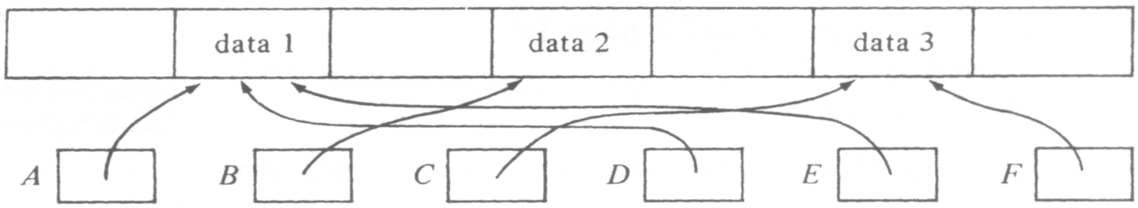
- O método 1 tende a desperdiçar espaço. A situação em que este tipo de estratégia deve ser preferida é quando o item de dados típico é grande.

  Por exemplo, muitos sistemas de arquivos trabalham desta forma, dividindo a memória secundária (tipicamente uma unidade de disco) em blocos de mesmo tamanho, de $2^9 - 2^{14}$ bytes, dependendo do sistema. Como muitos arquivos são bem maiores do que esses números, o desperdício de espaço é relativamente baixo.

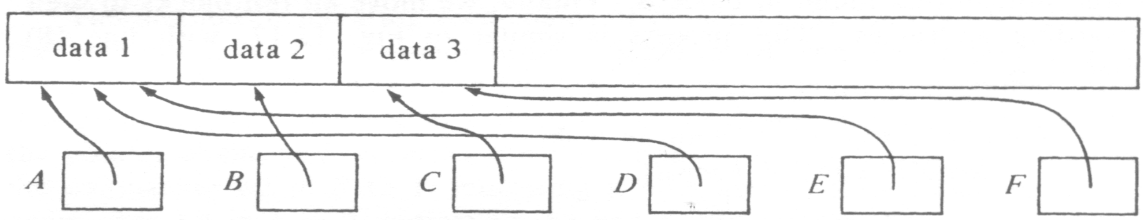
O problema da compactação

- Um problema típico é tomar uma coleção de blocos em uso, cada qual podendo ser de um tamanho diferente dos demais e podendo ser apontado por mais de um ponteiro, e deslocá-los à esquerda até que todo o espaço disponível esteja na extremidade direita do heap.

  Naturalmente, os ponteiros de dados precisam ser também atualizados.



(a) Before compaction



(b) After compaction

- Um esquema simples de compactação é primeiramente percorrer todos os blocos a partir da esquerda - usados ou vazios - e calcular um *forwarding address* (endereço de avanço) para cada bloco cheio.

  O endereço de avanço de um bloco é sua posição atual menos a soma de todo o espaço vazio à sua esquerda, ou seja, a posição para a qual o bloco deveria eventualmente ser removido.

- Para calcular os endereços de avanço: à medida em que varremos os blocos a partir da esquerda, acumulamos o montante de espaço vazio observado e subtraímos este montante da posição de cada bloco percorrido. Ver algoritmo abaixo.

```
(1)   var
          p: integer; { the position of the current block }
          gap: integer; { the total amount of empty space seen so far }
      begin
(2)       p := left end of heap;
(4)       gap := 0;
(5)       while p ≤ right end of heap do begin
              { let p point to block B }
(6)           if B is empty then
(7)               gap := gap + count in block B
              else { B is full }
(8)               forwarding address of B := p−gap;
(9)           p := p + count in block B
          end
      end;
```

- Tendo calculado os endereços de avanço, percorremos todos os ponteiros para o heap (referentes às variáveis).

  Seguimos cada ponteiro para algum bloco B e substituímos o ponteiro pelo endereço de avanço encontrado no bloco B.

  Finalmente, movemos todos os blocos cheios para seus endereços de avanço.

  A transferência dos blocos cheios para o início do heap, que requer tempo proporcional ao espaço ocupado por esses blocos, irá provavelmente dominar os demais custos da compactação.

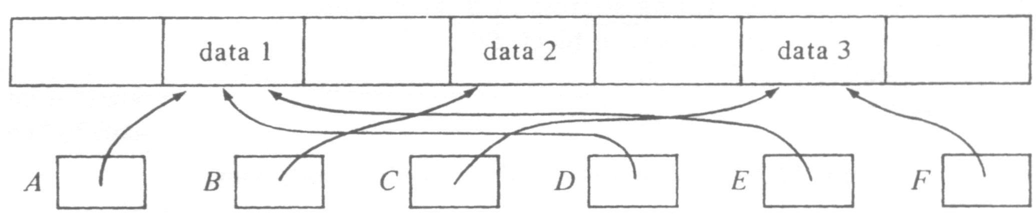
Algoritmo de Morris

- O algoritmo de Morris é um método de compactação de heap que dispensa o uso de endereços de avanço.

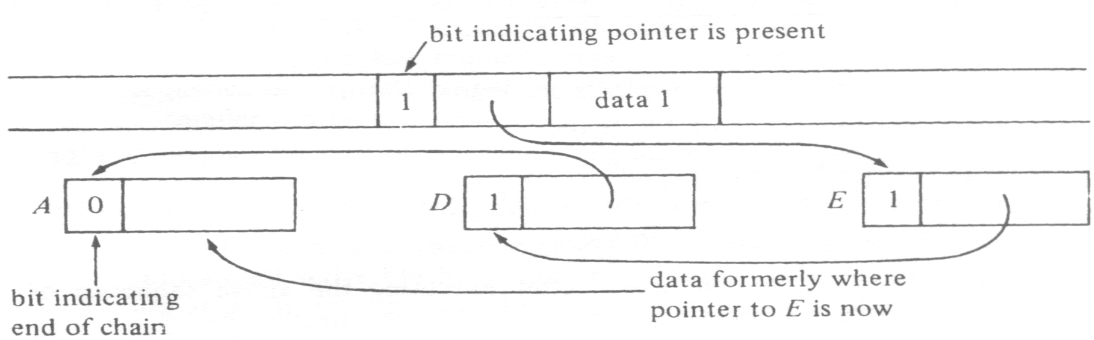
  Ele requer um bit associado com cada ponteiro e com cada bloco para indicar o fim de uma cadeia de ponteiros.

- A idéia básica é criar uma cadeia de ponteiros partindo de uma posição fixa em cada bloco cheio e ligando todos os ponteiros para aquele bloco.

  No exemplo da figura abaixo, há três ponteiros - A, D e E - apontando para o bloco cheio mais à esquerda.



A figura abaixo apresenta a cadeia de ponteiros desejada. Uma porção dos dados de tamanho igual ao de um ponteiro é removido do bloco e colocado no final da cadeia, onde o ponteiro A estava.



- O método para criar tais cadeias de ponteiros é como segue.
- Percorremos todos os ponteiros em qualquer ordem conveniente.

  Suponha que temos um ponteiro p para o bloco B.

  Se o bit marcador de B é 0, então p é o primeiro ponteiro encontrado que aponta para B. Colocamos em p o conteúdo daquelas posições de B usadas para a cadeia de ponteiros, e fazemos essas posições de B apontarem para p. Então atualizamos o bit marcador em B para 1, indicando que ele agora tem um ponteiro, e atualizamos o bit marcador em p para 0, indicando o fim da cadeia de ponteiros e a presença dos dados transferidos.

  Suponha agora que, quando olhamos o ponteiro p para B, o valor dor bit em B seja 1. Então B já contém a cabeça para uma cadeia de ponteiros. Copiamos o ponteiro em B para p, fazemos B apontar para p, e atribuímos o valor 1 para o bit marcador em p. Assim, efetivamente inserimos p na cabeça da lista.
- Uma vez que tenhamos todos os ponteiros para cada bloco ligados em uma cadeia partindo de cada bloco, podemos mover os blocos cheios tão à esquerda quanto possível, usando os métodos já apresentados.
- Finalmente, visitamos cada bloco em sua nova posição e percorremos sua cadeia de ponteiros. Cada ponteiro encontrado é atualizado para apontar para o bloco em sua nova posição. Quando encontramos o fim da cadeia, recuperamos os dados de B, guardado no último ponteiro, para seu lugar correto no bloco B, e atualizamos o bit marcador daquele bloco para 0.