Departamento de Informática e Matemática Aplicada – DIMAp – UFRN
Algoritmos e Estrutura de Dados I – DIM0110.0
Prática de Algoritmos e Estrutura de Dados I – DIM0110.1
Selan R. dos Santos – 16/10/2009

◁ Projeto de Programação ▷

Simulador de armazenamento de arquivos

Apresentação

O objetivo deste exercício de programação é apresentar um problema de simulação que requer a utilização da estrutura de dados do tipo *lista encadeada* em sua implementação. Um dos pré-requisitos deste exercício é que esta estrutura de dados já esteja implementada na forma de classes genéricas em C++.

A aplicação da TAD lista encadeada será na elaboração de um programa para simular o gerenciamento de armazenamento de arquivos em um disco virtual, denominado `alienados`[1]. O comportamento simulado é muito similar ao que acontece, por exemplo, no sistema de arquivos de um sistema operacional.

Espera-se que com esta aplicação seja possível perceber a importância das listas encadeadas como estrutura de dados na solução de problemas práticos.

1 Introdução

Um dos tópicos estudados na disciplina de Sistemas Operacionais refere-se a organização utilizada para armazenar informações (arquivos) em disco. Dependendo do sistema operacional, existem diversas estratégias para armazenamento e manipulação de arquivos, cada um com suas particularidades, vantagens e desvantagens. Por exemplo, *alocação contígua*, *arquivos encadeados*, *arquivos indexados* e *arquivos indexados multinível*.

Destas estratégias, a abordagem por *arquivos encadeados* requer a utilização de listas encadeadas tanto para manter a lista de setores ocupados por um dado arquivo quanto para representar a lista de setores livres em disco.

Mas antes de entrar em detalhes sobre o que estas listas devem armazenar, como devem ser criadas e manipuladas, precisamos introduzir alguns elementos básicos sobre a estrutura física do disco (virtual) que vamos simular.

1.1 Estrutura Física de um Disco de Dados

De uma maneira geral um *disco rígido* ou simplesmente *disco*, possui um certo número de *lados*, cada um com uma superfície magnética capaz de armazenar uma informação no formato binário.

Cada lado é divido em um certo número de *trilhas* concêntricas, onde a informação é de fato armazenada. Cada trilha, por sua vez, é dividida em um número de *setores*, cada um com capacidade para armazenar, por exemplo, 512 bytes. Um grupo de setores que é tratado como uma unidade de armazenamento é denominado de *cluster*[2]. No nosso caso, algumas simplificações se fazem necessárias para facilitar o processo de simulação.

[1] **A**plicação de **LI**stas **EN**cadeadas para **A**rmazenamento de **A**rquivos em um **D**isc**O** **S**imulado.
[2] Agrupamento.

Um byte, no nosso caso, será equivalente a um caractere (`char`). Um setor deverá armazenar `sizeOfSector` bytes, que deverá ser um número pequeno de forma a facilitar o acompanhamento da simulação. Por exemplo, podemos começar assumindo que um setor terá a capacidade de armazenamento de apenas 3 bytes (ou caracteres), assim `sizeOfSector` = 3. Contudo, este valor deverá ser um dos parâmetros de simulação, devendo estar na faixa $[2; 512]$.

O disco, denominado de `disk`, será representado como um arranjo unidimensional (vetor) de `numOfSectors` setores, cujo valor deverá também ser um dos parâmetros da simulação. Por exemplo, se `numOfSectors` = 30 e `sizeOfSector` = 3 teremos um disco com capacidade total de armazenamento de `numOfSectors*sizeOfSector` $= 30 \times 3 = 90$ bytes.

1.2 Estrutura Lógica de um Disco de Dados

A parte lógica da organização do disco requer a criação de um *sistema de arquivos*. O sistema de arquivos permite a realização de operações de criação e remoção de arquivos, bem como a localização e recuperação dos seus conteúdos. Outras operações de gerência envolvem a *formatação* (preparação do disco para receber os dados) e a *desfragmentação* (reorganização da listas de setores ocupados por um arquivo, com o objetivo de melhorar o desempenho da operação de recuperação de conteúdo de arquivos).

O nosso sistema de arquivos simplificado deve manter uma *área de sistema* e uma *área de dados*. A área de sistema armazena apenas uma FAT (*File Allocation Table*) correspondente ao registro de gerenciamento de espaço livre (*free storage management record*). Isto nada mais é do que uma lista de setores do disco que estão disponíveis, doravante denominada de `pool`.

Já a área de dados do nosso sistema de arquivos deve manter um registro dos arquivos atualmente armazenados em disco, o que seria equivalente a uma listagem de diretório (ou pasta). Este registro, denominada de `dataFiles`, corresponde a uma lista encadeada de entidades do tipo `file`.

A entidade `file` deve armazenar, pelo menos, informações como nome do arquivo, tamanho em bytes, uma lista encadeada de *clusters* de setores ocupados pelo arquivo e um apontador para o próximo `file` da área de dados (`dataFile`). Contudo, a entidade `file` **não poderá armazenar o conteúdo do arquivo em sua representação** — o conteúdo deve residir **apenas** no `disk`.

1.3 Exemplo

Um exemplo deve ilustrar os conceitos introduzidos nas seções anteriores. Supondo que já foram realizadas várias operações de armazenamento e remoções de arquivos, o disco atualmente armazena dois arquivos, `Arquivo 1` e `Arquivo 2`. O conteúdo do primeiro arquivo é de 23 '1's (23 bytes ou caracteres), enquanto que o conteúdo do segundo arquivo é de 13 '2's (13 bytes ou caracteres). Um "instantâneo" da memória correspondente ao disco do nosso simulador é apresentado na Figura 1.

Note que os setores 18 e 28, apesar de reservados, não estão sendo totalmente utilizados. Esta situação constitui um certo desperdício de espaço, uma vez que os bytes livres de tais setores **não** podem ser atribuídos a nenhum outro arquivo que venha a ser armazenado no disco.

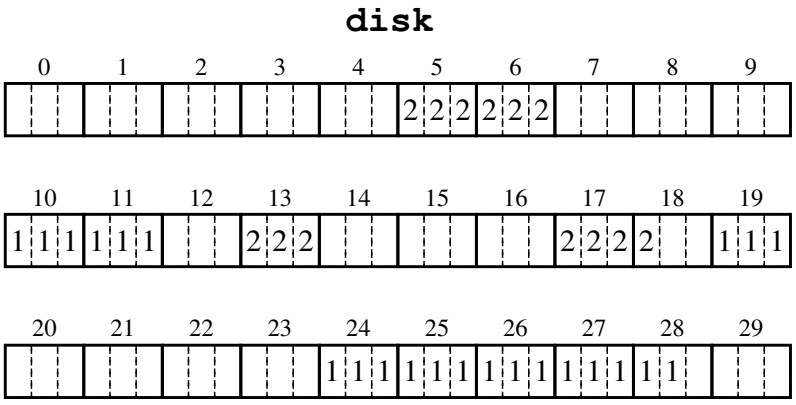

Figura 1: Representação da ocupação do disco, organizado em 30 setores. Cada setor armazena 3 bytes. O disco contém 2 arquivos, cujos conteúdos estão fragmentados em vários setores não-contíguos.

Para esta mesma configuração, o `pool` de setores livre seria representado como na Figura 2. Já a área de dados, `dataFiles`, pode ser visualizado na Figura 3.

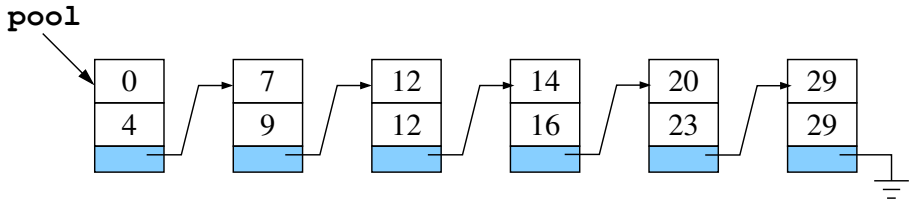

Figura 2: Representação da `pool` de setores livres no disco.

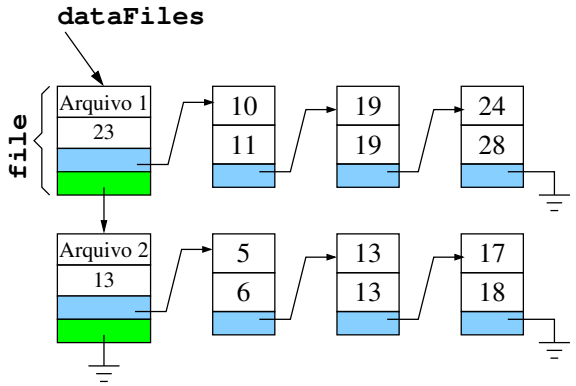

Figura 3: Representação da área de dados `dataFiles`, indicando a presença de 2 arquivos no disco. Note que os conteúdos dos arquivos **não** estão armazenados diretamente na estrutura `file`.

Se a operação de desfragmentação fosse realizada sobre este disco, o resultado seria uma reorganização dos setores ocupados pelos arquivos, conforme demonstrado na Figura 4. Note, na parte (a) da figura, o "lixo" de memória presente do setor 13 em diante. Apesar de ainda permanecer no disco, as informações do setor 13 pra frente não são mais acessíveis

diretamente. É por este motivo que as perícias policiais são capazes de recuperar (muitas vezes apenas parcialmente) dados que foram logicamente "deletados" em um disco — na verdade apenas a lista encadeada foi modificada, alguns setores ainda permanecem com seus conteúdos originais.

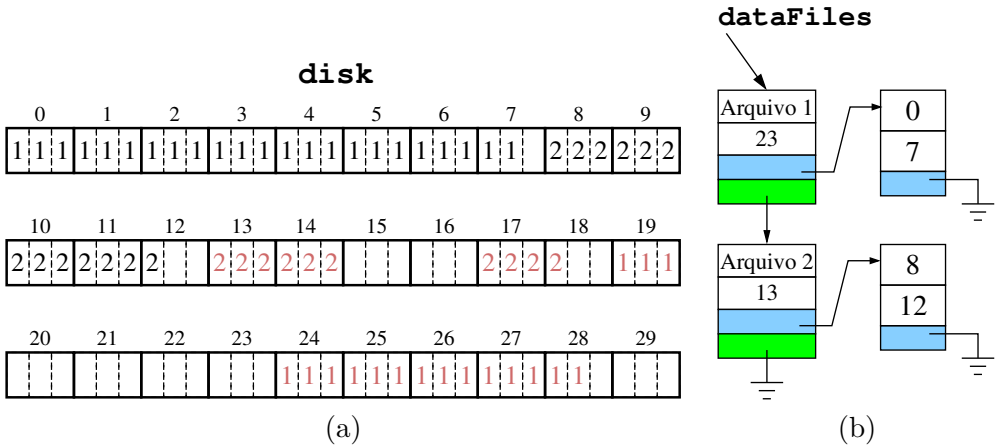

Figura 4: Representação do (a) disco desfragmentado e do correspondente (b) `dataFiles`.

1.4 Tarefa

Sua tarefa consiste em desenvolver o programa `alienados.cpp` para simular o gerenciamento de disco descrito nas seções anteriores. O gerenciador simulado deve ser capaz de suportar as seguintes operações:

- **Armazenamento** de arquivo: Esta operação recebe como entrada uma entidade 'arquivo' com informações sobre seu nome e conteúdo, na forma de uma string. A partir do conteúdo, é possível deduzir a quantidade necessárias de bytes (i.e. caracteres) necessários para armazenar tal arquivo em disco.

  O gerenciador então deve solicitar um número suficiente de setores livres do `pool`, se disponível. Estes setores, obviamente, podem não ser contíguos, portanto a lista encadeada correspondente a este arquivo pode conter vários nós. Cada nó armazena os índices inicial e final de uma faixa de setores contíguos reservado para o arquivo.

  Se não existir um número suficientes de setores livres para armazenar o arquivo em questão, o simulador deve exibir uma mensagem indicando tal fato na saída do programa[3].

  Após satisfeita a reserva de setores, o gerenciador deve escrever o conteúdo do arquivo nos setores reservados (i.e. posições do vetor `disk`), em seqüência, a partir do primeiro setor livre retornado.

- **Remoção** de arquivo: A remoção implica em remover a lista de setores da entrada da tabela `dataFiles` correspondente ao arquivo em questão, retornando-os para o `pool`. Note, contudo, que esta devolução não implica em simplesmente "mover nós" de uma lista para outra, realizando uma união de listas. É necessário fazer ajustes

[3] O formato da mensagem está definido mais adiante na Seção 2.2.

no `pool`, caso existam dois (ou mais) nós que, na verdade, representam um único bloco contíguo de memória.

Por exemplo, se o `Arquivo 2` da Figura 3 fosse removido (antes da desfragmentação), o `pool` deveria receber os setores pertencentes ao `Arquivo 2`, conforme indicado na Figura 5-(a). No entanto, faz-se necessário ajustar os intervalos de setores livres, conforme ilustrado na Figura 5-(b), que representa o `pool` em sua configuração final após o ajuste associado à remoção.

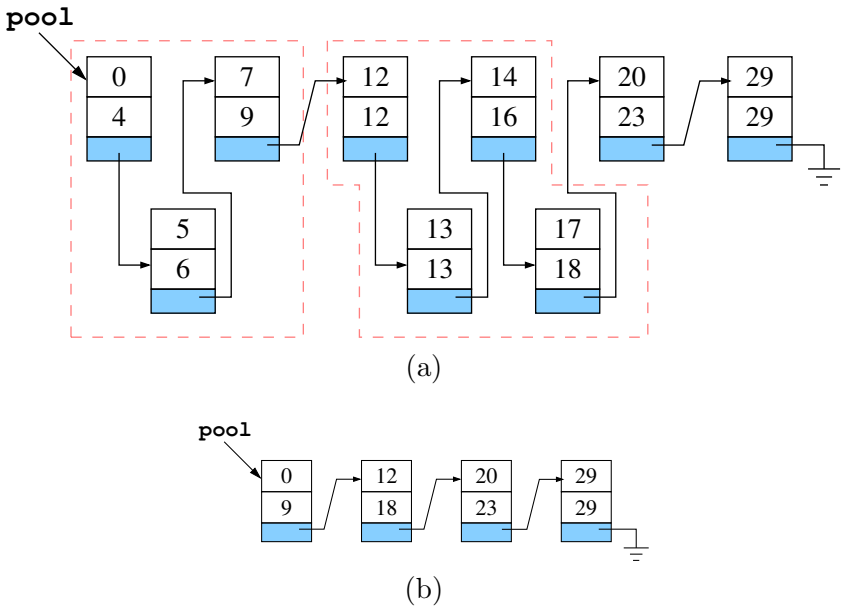

Figura 5: Representação (a) intermediária do `pool` após a remoção do `Arquivo 2` da Figura 3; e (b) representação final do `pool` após ajuste de continuidade de setores.

- ⋆ **Listagem** de arquivos em disco: Esta operação simplesmente requer a iteração através da `dataFiles`. A cada entrada encontrada, deve-se imprimir o nome do arquivo, seu tamanho em bytes e seu conteúdo (cadeia de caracteres). Para acessar o conteúdo de um arquivo devemos percorrer sua lista de setores, utilizando os índices armazenados em tal lista para recuperar as informações do vetor `disk`.
- ⋆ **Formatação** de disco: Esta ação implica na remoção de todas as entradas da `dataFiles` e liberação dos setores para o `pool`. Após a formatação, o `disk` deve conter apenas '0' (zeros), o `pool` deve conter apenas um nó, indicando uma única faixa de memória contígua livre, e o `dataFiles` deve ser uma lista vazia.
- ⋆ **Desfragmentação** de disco: A fragmentação de um arquivo em disco retarda o processo de recuperação dos dados, pois é necessário percorrer uma longa lista encadeada para poder ter acesso aos dados do arquivo. Em uma situação ideal, apenas um único *cluster* de setores deve ser atribuído a um arquivo, indicando que o arquivo ocupa um único bloco contíguo de memória. Portanto, o objetivo da operação de desfragmentação é realizar o agrupamento de setores, ou seja, transferir os arquivos para setores contíguos, criando a situação ilustrada na Figura 4. Os arquivos `Arquivo 1` e `Arquivo 2`, anteriormente fragmentados, agora ocupam, cada um, um único *cluster* de setores.

Contudo, deve-se tomar cuidado para que, no decorrer do processo de desfragmentação, um setor ocupado por um arquivo não seja acidentalmente sobrescrito por partes do arquivo atualmente sendo desfragmentado. Por exemplo, o `Arquivo 1`, requer oito setores; cinco setores estão livres no início do `pool`, mas os setores 5 e 6 estão ocupados pelo `Arquivo 2`. Portanto, o arquivo f que está ocupando tais setores deve ser localizado, através de uma investigação da listas de arquivos `dataFiles`. O conteúdo destes setores devem ser, então, transferidos para posições desocupadas, o que requer a atualização da lista encadeada de setores pertencentes a f; apenas então, os setores 5 e 6 poderiam ser de fato liberados.

Uma maneira de realizar esta operação é copiar partes do arquivo para áreas do disco livres e grandes o suficiente para conter o maior número de setores contíguos. No exemplo da Figura 1, as partes iniciais do `Arquivo 1` foram copiadas para o setores de 0 à 4; então a cópia é temporariamente suspensa, pois o setor 5 está ocupado. Assim, o conteúdo dos setores 5 e 6 foram movidos para os setores 12 e 14; só então o restante do `Arquivo 1` pode ser copiado.

Por fim, se a desfragmentação for solicitada e não houver espaço livre suficiente para movimentar os setores, o simulador deve exibir uma mensagem indicado que a operação não pode ser executada por falta de espaço livre em disco.

2 Formato de Entrada e Saída de Dados

O programa `alienados` deverá receber, através de argumentos de linha de comando, o nome de um arquivo texto (ASCII) com os parâmetros da simulação.

```
$./alienados comandos.txt
```

Após o processamento da entrada para a simulação, o programa deve gerar dois arquivos de saída. O primeiro arquivo, denominado de `alienados_sim.txt`, deve conter o resultado da simulação, que também deve ser impresso na tela. Por resultado da simulação entende-se uma representação textual do `disk`, `pool` e `dataFiles`. O segundo arquivo, denominado de `alienados_trash.txt`, deve conter os arquivos que foram removidos durante a simulação (i.e. nome + conteúdo).

2.1 Entrada de Dados

A primeira linha do arquivo de entrada deve indicar o tamanho de um setor, `sizeOfSector`, e a quantidade de setores do disco a ser simulado, `numberOfSectors`. Com estes dois valores é possível determinar a capacidade total de armazenamento em bytes do disco virtual. Seu programa deve, então, alocar dinamicamente memória para `disk` de acordo com esta informação.

Da segunda linha válida[4] em diante, o arquivo organiza-se em grupos de, no máximo, 3 linhas. A primeira linha indica o comando a ser simulado e as linhas seguintes, se houver, constituem o argumento do comando.

Os comandos `Formatar`, `Desfragmentar` e `Listar` não possuem argumentos e, portanto, são representados com apenas 1 linha no arquivo de entrada. O comando `Remover` requer apenas o nome do arquivo a ser removido (total de 2 linhas). Já o comando `Armazenar` deve receber o nome do arquivo em uma linha e seu conteúdo na próxima linha

[4]Linhas em branco devem ser ignoradas.

(total de 3 linhas). Lembre-se que ao tratar um comando, o programa deve ignorar se o mesmo esta em caixa alta ou baixa. A Figura 6 apresenta um exemplo de um arquivo de entrada válido.

```
3 30

Formatar

Armazenar
Teste
Este eh o conteudo deste arquivo

Armazenar
Arquivo 1
111111111111111111111

Armazenar
Arquivo 2
22222222222222

Listar

Remover
Teste

Desfragmentar
```

Figura 6: Exemplo de arquivo de entrada válido com 7 comandos. As linhas em branco foram introduzidas apenas para facilitar a leitura e compreensão do conteúdo do arquivo de entrada, sendo ignoradas pelo programa.

2.2 Saída de Dados

A saída do programa deverá seguir rigorosamente o proposto nesta seção. O arquivo `alienados_sim.txt` deve conter uma saída formatada para cada comando na entrada. Uma saída formatada deve conter um *mapa da memória*, ou seja, uma representação do `disk`, `pool` e `dataFiles`, organizados da seguinte maneira:

```
disk:[000][000][000][222][000][222][200][000][111][110]
pool:[0-2]->[4-4]->[7-7]->null
dataFiles:
{Arquivo 1,5 bytes,[8-9]->null}
{Arquivo 2,7 bytes,[3-3]->[5-6]->null}
```

No exemplo acima, temos `sizeOfSector` $= 3$ e `numberOfSectors` $= 10$, e dois arquivos, `Arquivo 1` e `Arquivo 2`, estão armazenados. Note que `disk` e `pool` devem ocupar apenas 1 linha cada. Já o `dataFiles` deve ocupar $f + 1$ linhas, onde f é o número de arquivos atualmente armazenados no disco virtual.

Um arquivo é representado no formato '`{<nome arquivo>,<tamanho em bytes>,<lista encadeada de setores ocupados>}`', sem espaços em branco entre os três componentes. Uma lista encadeada é representada no formato '`[`s_i`-`s_f`]->...->null`', onde `[`s_i`-`s_f`]` indica a ocupação do setor inicial, s_i, até o setor final, s_f; `->` indica encadeamento; e `null` indica fim de lista. Uma lista vazia deve ser representada apenas com `null`.

No caso do comando `Listar`, deve-se listar o nome, tamanho em bytes e o conteúdo de cada arquivo (recuperados de `disk`), não sendo necessário gerar um mapa da memória. Para o exemplo acima, a saída correspondente ao comando `Listar` seria:

```
Arquivo 1, 5 bytes
11111
Arquivo 2, 7 bytes
2222222
```

A Figura 7 apresenta um exemplo de entrada de dados, enquanto que a Figura 8 apresenta o arquivo `alienados_sim.txt` correspondente e a Figura 9 apresenta o arquivo `alienados_trash.txt`, ambos gerados ao final da simulação.

```
3 10
Formatar
Armazenar
Teste
conteudo deste arquivo
Armazenar
Arquivo 1
11
Armazenar
Arquivo 2
22222222222222
Listar
Remover
Teste
Armazenar
Arquivo 3
XXXXXXXXXXXXX
Armazenar
Read.me
YYYYYYYYYY
Desfragmentar
Remover
Arquivo 3
Desfragmentar
```

Figura 7: Exemplo de arquivo de entrada válido com 11 comandos.

3 Implementação

A implementação do trabalho deve ser feita através do uso de classes. Não serão aceitas soluções que utilizem qualquer classe de estruturas de dados do STL. Portanto, você deve usar a classe *List* do exercício anterior ou desenvolver uma nova classe de lista encadeada, se achar necessário.

4 Avaliação do Programa

O programa completo deverá ser entregue sem erros de compilação, testado e totalmente documentado. O programa `alienados` será avaliado sob os seguintes critérios:

- ⋆ Armazenar funcionando (15 %)

- ⋆ Listar funcionando (15 %)
- ⋆ Remover funcionando (20 %)
- ⋆ Formatar funcionando (5 %)
- ⋆ Desfragmentar funcionando (25 %)
- ⋆ Leitura correta de arquivo (10 %)
- ⋆ Gravação correta de arquivo (10 %)
- ◦ Presença de erros de compilação e/ou execução (até −20%)
- ◦ Falta de documentação do programa com Doxygen (até −10%)
- ◦ Vazamento de memória identificado com o valgrind (até −10%)
- ◦ Falta ou incompletude do arquivo README.TXT (até −10%)

A pontuação acima não é definitiva e imutável. Ela serve apenas como um guia de como o trabalho será avaliado em linhas gerais. Desta forma, os professores têm total liberdade para realizar ajustes nas pontuações indicadas visando adequar a pontuação ao nível de dificuldade dos itens solicitados.

5 Tema de Pesquisa

Além do desenvolvimento do projeto de programação especificado, a equipe de trabalho deve realizar uma pesquisa teórica. Como retultado, deve ser elaborado um relatório descritivo para o trabalho e realizada a apresentação oral correspondente.

Vale a pena ressaltar que o relatório gerado deve apresentar as seguintes seções:

- ⋆ **Introdução**: descrição do problema tratado, fornecendo o contexto e deixando claro qual o tema de pesquisa/investigação;
- ⋆ **Revisão ou *Background***: nesta seção você deve citar fontes científicas relacionadas ao seu trabalho (referências bibliográficas) ou fornecer conceitos necessários ao entendimento da teoria que será apresentada mais adiante;
- ⋆ **Metodologia**: esta é a seção principal do relatório, onde deve ser descrito o que foi pesquisado e qual a metodologia utilizada.
- ⋆ **Resultados**: nesta seção deve-se apresentar os resultado obtidos;
- ⋆ **Conclusão**: nesta seção você deve sumarizar o propósito da pesquisa, destacando os principais resultados alcançados; por favor, evite comentários óbvios do tipo "(...) foi muito bom estudar esta matéria, pois consegui aprender muito!"

Para este trabalho sua pesquisa deve investigar estratégias usada para realizar *desfragmentação*. Procure direcionar seu trabalho de maneira a definir o que esta área estudada, além dos métodos tradicionamente utilizados. É necessário também apresentar exemplos de algoritmos utilizados em aplicações reais, como por exemplo em sistemas operacionais.

6 Entrega

O trabalho deve ser desenvolvido em triplas, tentando, dentro do possível, dividir as tarefas igualmente entre os componentes. Porém os componentes devem ser capazes de explicar qualquer trecho de código do programa, mesmo que o código tenha sido desenvolvido pelo outro membro da equipe.

Para a solução deste projeto **é obrigatório** a utilização das classes pilha, fila e lista sequencial que foram desenvolvidas em trabalhos anteriores. Não serão aceitas soluções que utilizem as estruturas de dados da biblioteca STL (e.g. `list`, `stack`, `vector`, etc.).

Você deve submeter dois componentes: o relatório de pesquisa e todo o código fonte correspondente ao projeto em uma pasta. Na pasta do programa deve existir um arquivo denominado de `README.TXT`, no qual a equipe deve indicar: (1) Nome e matrícula da dupla de desenvolvedores; (2) Instruções de como compilar o programa (ou então um `Makefile`); (3) Instruções de como executar o programa; e (4) Indicações de eventuais limitações ou incompletudes do programa.

Eventualmente, algumas duplas poderão ser convocadas para uma entrevista. O objetivo de tal entrevista é comprovar a verdadeira autoria do código entregue. Assim, qualquer um dos componentes da dupla deve ser capaz de explicar qualquer trecho de código do projeto. Trabalhos plagiados receberão nota **zero** automaticamente.

A entrega deve ser feita através da opção Tarefas da turma Virtual do Sigaa, em data divulgada no sistema. Entregas atrasadas terão sua nota final diminuída de um valor proporcional aos dias de atraso. **Somente serão aceitas entregas feita através do Sigaa.**

$\sim$ FIM $\sim$

```
disk:[000][000][000][000][000][000][000][000][000][000]
pool:[0-9]->null
dataFiles:

disk:[con][teu][do ][des][te ][arq][uiv][o00][000][000]
pool:[8-9]->null
dataFiles:
{Teste,22 bytes,[0-7]->null}

disk:[con][teu][do ][des][te ][arq][uiv][o00][110][000]
pool:[9-9]->null
dataFiles:
{Teste,22 bytes,[0-7]->null}
{Arquivo 1,2 bytes,[8-8]->null}

Not enough disk space to store requested file ``Arquivo 2''

Teste, 22 bytes
conteudo deste arquivo
Arquivo 1, 2 bytes
11

disk:[con][teu][do ][des][te ][arq][uiv][o00][110][000]
pool:[0-7]->[9-9]->null
dataFiles:
{Arquivo 1,2 bytes,[8-8]->null}

disk:[XXX][XXX][XXX][XXX][Xe ][arq][uiv][o00][110][000]
pool:[5-7]->[9-9]->null
dataFiles:
{Arquivo 1,2 bytes,[8-8]->null}
{Arquivo 3,13 bytes,[0-4]->null}

disk:[XXX][XXX][XXX][XXX][Xe ][YYY][YYY][YYY][110][YY0]
pool:null
dataFiles:
{Arquivo 1,2 bytes,[8-8]->null}
{Arquivo 3,13 bytes,[0-4]->null}
{Read.me,11 bytes,[5-7]->[9-9]->null}

Not enough free disk space to perform defragmentation

disk:[XXX][XXX][XXX][XXX][Xe ][YYY][YYY][YYY][110][YY0]
pool:[0-4]->null
dataFiles:
{Arquivo 1,2 bytes,[8-8]->null}
{Read.me,11 bytes,[5-7]->[9-9]->null}

disk:[11X][YYY][YYY][YYY][YY ][YYY][YYY][YYY][110][YY0]
pool:[5-9]->null
dataFiles:
{Arquivo 1,2 bytes,[0-0]->null}
{Read.me,11 bytes,[1-4]->null}
```

Figura 8: Arquivo `alienados_sim.txt` gerado pela simulação indicada na Figura 7. Repare que no total foram feitas 11 operações e o resultado de cada uma está separado por 1 linha em branco. Note que somente a operação *Listar* não precisou exibir o mapa de memória. Perceba também a indicação de duas mensagens de erro: uma gerada quando tentou-se armazenar o `Arquivo 2` e outra quando foi solicitado uma desfragmentação em um disco cheio.

```
Teste
conteudo deste arquivo

Arquivo 1
11
```

Figura 9: Arquivo `alienados_trash.txt` gerado pela simulação indicada na Figura 7. Dois arquivos foram removidos durante a simulação.