Documentação para o Trabalho Prático de AEDS 2

Autor: Vítor Gabriel Reis Caitité

Prof. Leonardo Barbosa e Oliveira

1. Introdução:

O objetivo deste trabalho é implementar uma busca recursiva por um valor específico, nesse caso 0, em imagens digitais no formato PGM, o qual já foi descrito no TP0. Esse formato armazena uma imagem em tons de cinza.

Algoritmos para realizar uma busca por um pixel específico, ou seja, varrer uma matriz de uma imagem procurando tal pixel, têm algumas aplicações importantes na área de processamento de imagens, como por exemplo podem ser utilizados em processos de correção de pequenas imperfeições em imagens.

2. Implementação:

Estrutura de Dados

Para a implementação do trabalho foi criado um Tipo Abstrato de Dados PGM com os seguintes campos:

- int c - Número de colunas da imagem
- int l - Número de linhas da imagem
- unsigned char máximo - Valor máximo para cada pixel
- unsigned char **imagem - ponteiro de ponteiro para os pixels da imagem (matriz)

E também um Tipo Abstrato de Dados Pilha que contém o campo

- Celula *topo – Ponteiro para a célula no topo da pilha

A estrutura Celula, por sua vez, possui os campos:

- int l - Linha de um ponto do caminho até a primeira célula de valor 0
- int c - Coluna de um ponto do caminho até a primeira célula de valor 0
- Celula próximo - Ponteiro para a próxima célula da pilha

Funções e Procedimentos

O TAD desenvolvido possui as seguintes funções:

PGM*LerPGM(char*entrada): Essa função primeiramente aloca memória para o variável do tipo da estrutura de dados desenvolvida a qual foi dada o nome de img. Então ela lê de um arquivo (.pgm) alguns dados importantes como o números de linhas e de colunas da matriz de pixels e também o maior valor que

cada pixel pode possuir. A partir desses valores ela chama uma função que aloca memória para a matriz de pixels à qual existe na estrutura um ponteiro de ponteiro associado a ela.

O próximo passo realizado é chamar a função fscanf repetidamente para assim ler e armazenar em img->imagem os valores da matriz de pixels lida.

Por fim, essa função fecha o arquivo de onde se leu todos os dados e retorna um endereço onde foi alocado memória e armazenado os elementos.

unsigned char alocacao(int lines, int columns):** Essa função é responsável por alocar memória para a matriz de pixels (por isso a função é do tipo unsigned charr**) utilizada durante o funcionamento do programa. Ela recebe como parâmetros as linhas e colunas da matriz a ser alocada.

O processo realizado foi alocar um vetor de ponteiros, então percorrer esse vetor e alocar um vetor de unsigned char para cada posição dele. E no fim da função retorna-se o endereço para a matriz alocada.

void liberarPGM(PGM* img): Procedimento que libera a memória alocada para a variável do tipo da estrutura implementada e para todos os campos que ela possui (inclusive para a matriz de pixels).

Primeiramente libera-se todas as linhas dessa matriz, para assim desalocar também o vetor de ponteiro restante. Após isso bastou liberar a memória alocada para a variável do tipo da estrutura implementada.

void FPVazia(Pilha* pilha): Esse procedimento recebe como parâmetro um endereço alocado para uma variável do tipo Pilha e então cria uma pilha vazia, ou seja cria uma célula cabeça que passa a ser o topo da pilha e faz com que o campo próximo da célula cabeça aponte para NULL (exatamente porque a pilha está vazia). Após a execução dessa função tem-se uma pilha inicialmente vazia. Como a função é do tipo void, ela não retorna valor;

int TestePVazia(Pilha *pilha): Função que testa se uma pilha está vazia, e retorna 1 caso ela esteja e 0 caso contrário. Para verificar se a pilha está vazia basta observar se pilha->topo->próximo é igual a NULL, ou seja, verificar se a pilha só possui a célula cabeça.

A função deve receber como parâmetro um endereço de uma variável do tipo Pilha.

void Empilha(Pilha *pilha, Celula* celula): Esse procedimento recebe como parâmetros um endereço de uma variável do tipo Pilha e um endereço de uma

célula que contém as coordenadas de um ponto percorrido no caminho para encontrar o pixel 0.

Com isso, cria-se uma nova célula cabeça e adiciona-se as coordenadas, contidas na célula passada como parâmetro, na antiga célula cabeça.

A célula cujo o endereço foi passado como parâmetro ainda tem sua memória liberada ao fim da função empilha.

Celula* Desempilha(Pilha *pilha): Recebe como parâmetros um endereço de uma variável do tipo Pilha. Primeiramente ela chama a função “TestePVazia” que testa se a pilha está vazia, caso ela esteja então avisa que é impossível desempilhar já que a pilha está vazia, e sai.

Caso a pilha não esteja vazia desliga-se a célula cabeça da lista. A próxima célula, que contém o primeiro item, passa a ser a célula cabeça, por fim, é retornado um ponteiro para essa nova célula cabeça que contém dados das coordenadas de um ponto percorrido durante o percurso para encontrar o pixel 0 na matriz imagem.

void Busca(PGM *img, Pilha *pilha): Essa se trata de uma função recursiva que realiza a busca recursiva do pixel 0 na imagem PGM e armazena na pilha, cujo o endereço foi passado como parâmetro, o percurso percorrido até encontra-lo. Os critérios de caminhamento na imagem utilizados nesse procedimento foram:

- Se o valor da posição em avaliação não for 0, escolhe-se entre os vizinhos esquerdo, direito, superior e inferior (caso existam) aquele de menor valor.
- Foi criado um mecanismo para garantir que nenhum pixel será percorrido mais de uma vez (até porque caso isso ocorresse o programa entraria em loop infinito).
- Caso dois ou mais vizinhos tenham o menor valor, escolhe-se segundo a seguinte ordem de precedência: direito, inferior, esquerdo e superior.
- Foi utilizado uma pilha para armazenar as posições já percorridas e a ser percorrida.

Para executar todos esses comandos implementou-se a seguinte lógica: primeiramente foi criada a condição de parada, que é se as coordenadas lidas da pilha corresponderem a um valor zero na matriz imagem. Se essa condição de parada não for satisfeita, testa-se se é possível andar para direita e se a posição da direita ainda não foi percorrida. Caso essas condições forem respeitadas então copia-se o valor desse pixel da direita para uma variável “menor” e salva suas coordenadas em uma célula. Caso as condições para andar para direita não sejam respeitadas, atribui-se um valor maior do que o valor de todos os pixels da imagem para a variável menor, garantindo assim que o pixel escolhido para ser percorrido seja obrigatoriamente ou o de baixo, ou o da esquerda ou o da direita.

Após esse primeiro teste, a função checa também se é possível andar para baixo e o elemento de baixo ainda não foi percorrido. Satisfeitas essas questões, realiza-se um novo teste para saber se o elemento de baixo é menor que o valor salvo na variável menor. Caso afirmativo, a variável menor recebe esse novo valor e salva-se as coordenadas desse novo elemento; caso contrário, segue-se o código normalmente.

As mesmas operações são executadas também para o pixel logo a esquerda e para o pixel logo acima. Assim ao final tem-se guardado em uma célula as coordenadas do pixel de menor magnitude dentre os que foram testados. Então empilha-se essa célula na pilha passada como parâmetro e chama-se a função novamente, caracterizando uma recursividade.

int BuscaPilha(Pilha* pilha, int linha, int coluna): Verifica se existe, na pilha, uma célula com um determinado par de coordenadas passado como parâmetro. Essa função recebe um endereço de uma estrutura Pilha e variáveis, do tipo inteiro, que contêm as coordenadas que definem a posição de um pixel. Por fim, retorna um se a pilha contém uma célula com o par de coordenadas passado como parâmetros e retorna zero caso contrário. Para realizar tal operação, percorre-se toda a pilha, comparando os valores dos campos ‘l’ e ‘c’ de cada célula da pilha com os valores presentes nas variáveis linha e coluna. Nessa atividade de percorrer a pilha, não foi desempilhado nada, já que foi criado um ponteiro do tipo célula para percorrer cada célula da pilha, começando da primeira logo após a célula cabeça e terminando na primeira que foi empilhada.

void ImprimeCaminho(Pilha *pilha): Imprime na tela o caminho percorrido da origem, ou seja, pixel (0,0), até o pixel de módulo 0, respeitando as condições de percurso já citadas. Recebe como parâmetro o endereço de memória alocado para uma variável pilha do tipo Pilha. Ao final da sua execução tem-se impresso na tela das coordenadas dos pontos percorridos, iniciando pixel de posição (0,0).

Primeiramente, foi necessário criar uma pilha auxiliar para receber os elementos desempilhados da pilha original. Foi feito isso para assim ser possível imprimir as coordenadas percorridas na ordem correta, visto que se caso fosse impresso exatamente os elementos desalocados da pilha original, então seria visto o percurso ao inverso, ou seja, começando das coordenadas do pixel de módulo zero e terminando na origem. A partir, do momento em que todos os elementos desempilhados da pilha recebida como parâmetro foram empilhados na pilha auxiliar, pode-se então começar a desempilha-la e imprimir na tela as coordenadas presentes em cada célula. Após a impressão de cada coordenada foi-se liberando a célula que a continha.

Por fim, resta-se na pilha auxiliar apenas a célula cabeça, então chama-se a função “LiberaPilha” que libera a memória alocada para essa célula e também para a pilha.

void LiberaPilha(Pilha* pilha): Função que libera memória da variável do tipo pilha alocada, juntamente com a célula cabeça dessa pilha. A função deve ser chamada apenas quando a pilha já está vazia, ou seja, contém apenas a célula cabeça). Esse procedimento recebe o endereço de memória alocado para uma variável pilha do tipo Pilha e somente utiliza duas vezes a função “free” para assim liberar a memória alocada para a célula cabeça e para a pilha.

Programa principal: int main(int argc, char *argv[ ]): Primeiramente o programa principal, cria um ponteiro do tipo PGM (que inicialmente aponta para NULL) que irá apontar para os dados que serão lidos da imagem.pgm. Posteriormente, chama-se no “main” a função “LerPGM” que lê os dados do arquivo.pgm e também aloca a memória onde esses dados ficarão armazenada.

Ainda nessa fase de declarações, também foi declarado um ponteiro para a pilha que irá guardar o caminho percorrido até alcançar o pixel zero. Para isso foi necessário alocar dinamicamente a pilha.

Então foi chamada a função “FPVazia”, para verdadeiramente criar uma pilha inicialmente vazia (apenas com a célula cabeça). Nessa pilha, é empilhado a célula com as coordenadas da origem.

Tendo assim todos esses procedimentos iniciais realizados, a função principal chama a função “Busca”, passando como parâmetros o endereço da estrutura PGM contendo a imagem e endereço de uma estrutura Pilha, na qual será empilhado as células contendo as coordenadas de cada pixel percorrido (e já possui empilhada as coordenadas do pixel de origem).

Mapeado e salvo em uma pilha todo o percurso até o pixel de módulo zero é necessário imprimir esse percurso na tela, sendo assim a função principal apenas invoca a função “ImprimeCaminho”, que por sua vez, executa todas as operações necessárias para imprimir esse percurso.

E por fim, restou apenas desalocar as memórias alocadas dinamicamente, que foram a memória da pilha e a memória da estrutura que guarda os dados da imagem. Para isso foi invocado os procedimentos “LiberarPilha” e “LiberaPGM” respectivamente.

Obs.: Quando se chama a função “LiberarPilha”, a pilha já está vazia e com todas as suas células, exceto a cabeça, desalocadas.

Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está dividido em três arquivos principais: TP1.c e TP1.h implementam o Tipo Abstrato de Dados enquanto o arquivo main.c implementa o programa principal.

Além das funções exigidas resolvi implementar outras funções para facilitar a execução do projeto, essas funções foram:

- unsigned char** alocacao(int lines, int columns)
- void liberarPGM(PGM* img)
- void Empilha(Pilha *pilha, Celula* Nova)
- Celula* Desempilha(Pilha *pilha)
- void FPVazia(Pilha *pilha)
- int TestePVazia(Pilha *pilha)
- int BuscaPilha(Pilha* pilha,int linha, int coluna)
- void LiberaPilha(Pilha* pilha)

O compilador utilizado foi o gcc no sistema operacional Linux Ubuntu. Para executar o programa, basta compilar e então a partir da linha de comando abrir o programa e passar como argumento o nome do arquivo com a imagem original (imagem essa que deve estar no mesmo diretório do programa).

Exemplo: ./nome_do_programa teste1.pgm

3. Análise de Complexidade

Função LerPGM: Considerando a operação de leitura como a operação principal e ‘n’ como o número de elementos da matriz da imagem.pgm, então a função “LerPGM” executa a operação de leitura ‘n’ vezes, para cada uma das linha da matriz, logo ela é $O(n^2)$.

Função alocacao: Considerando a operação de alocação de memória como a operação principal, então a função “alocacao” executa essa operação uma vez, para cada uma das ‘n’ linhas da matriz da imagem.pgm, logo ela é O(n).

Função liberarPGM: Considerando a operação liberar memória como a operação principal e ‘n’ como o número de linhas da matriz imagem, então a função “liberarPGM” executa a operação de desalocar memória n+2 vezes, ou seja essa função é O(n).

Função FPVazia: Considerando a operação de alocação de memória como a operação principal, então a função “FPVazia” executa essa operação uma única vez, a cada vez que é chamada, logo ela é O(1).

Função TestePVazia: Considerando a operação de comparação como a operação principal, então a função “TestePVazia” executa essa operação uma única vez, a cada vez que é chamada, logo ela é O(1).

Função Empilha: Considerando a operação de atribuição como a operação principal, então a função “Empilha” executa essa operação quatro vezes, a cada vez que é chamada, logo ela é O(1).

Função Desempilha: Considerando a operação de atribuição como a operação principal, então a função “Desempilha” executa essa operação duas vezes, a cada vez que é chamada, logo ela é O(1).

Função Busca: Considerando a operação de comparação como a operação principal e ‘n’ como o número de elementos que estão no percurso entre a origem e o pixel 0, então a função “Busca” executa comparação um número conhecido de vezes, para cada um desses elementos, a cada vez que ela é chamada (pois se trata de uma função recursiva), ou seja, essa função é O(n).

Função BuscaPilha: Considerando a operação de comparação como a operação principal e ‘n’ como o número de elementos da pilha, exceto a célula cabeça, então a função “BuscaPilha” executa a operação de comparação duas vezes, para cada célula da pilha, ou seja a operação de comparação é executada 2n vezes, logo ela é O(n) para o pior e médio caso e é O(1) para o melhor caso.

Função ImprimeCaminho: Considerando a operação de Desempilhar como a operação principal e ‘n’ como o número de elementos da pilha, então a função “ImprimeCaminho” executa essa operação duas vezes, utilizando a função “Desempilha”, para cada célula da pilha, assim essa função é O(n). OBS: Se se considerasse a função principal como sendo a de impressão ainda sim se teria O(n).

Função LiberaPilha: Considerando a operação de liberar memória como a operação principal, então a função “LiberaPilha” executa essa operação apenas duas vezes, a cada vez que é chamada, logo ela é O(1).

Função Principal: o programa principal apenas chama uma vez cada uma das funções descritas acima e faz alguns comandos constantes. Dessa forma, a sua complexidade é regida pela função de maior custo computacional: $O(n^2) + O(1) + O(n) + O(n) + O(n) + (1) + O(n) = O(n^2)$.

4. Testes:

Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Utilizando-se a imagem teste1.pgm encontrou-se exatamente o resultado esperado, como mostrado abaixo:

- Resultado esperado pode ser visto na seguinte ilustração:

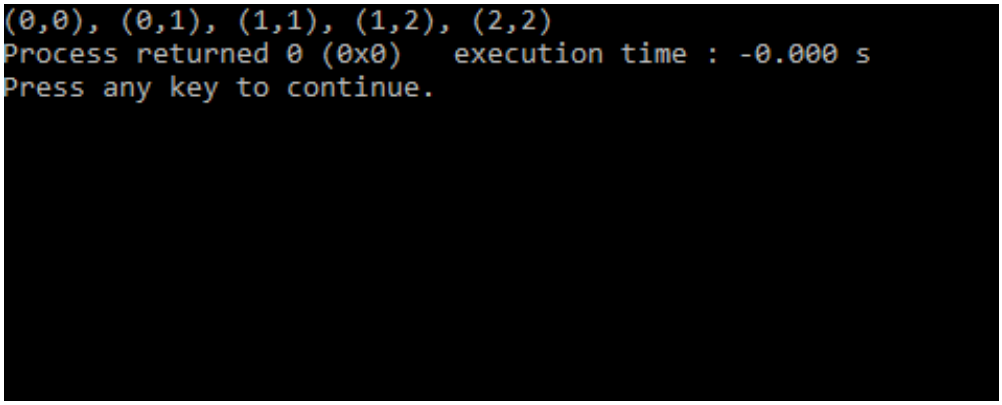
Entrada			
92	80	205	0
170	32	164	43
25	164	0	36
75	0	87	26

Caminho	
92	0,0
80	0,1
32	1,1
164	1,2
0	2,2

Pilha
2,2
1,2
1,1
0,1
0,0

Para esse exemplo, a saída esperada é (0,0), (0,1), (1,1), (1,2), (2,2).

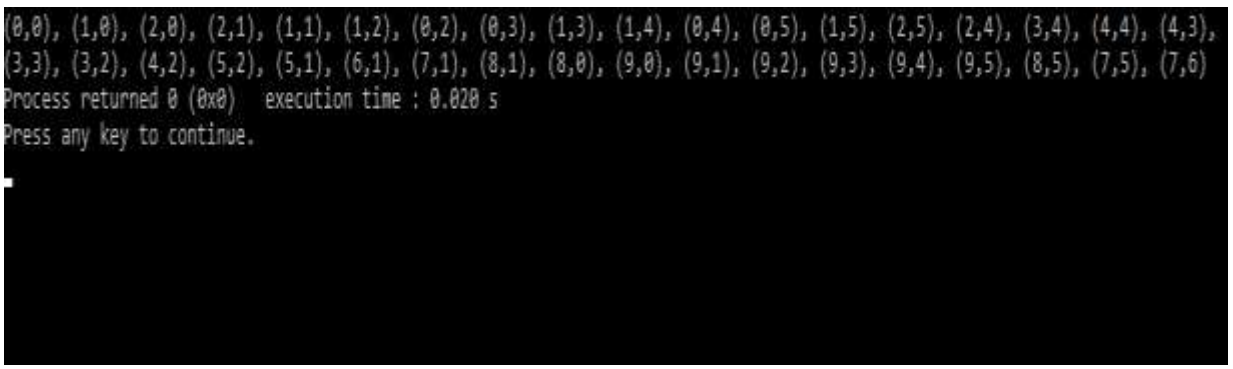
- Resultado encontrado:



Outro teste importante foi utilizando a imagem “teste2.pgm”, enviada também juntamente com a descrição do trabalho.

- Resultado esperado: (0,0), (1,0), (2,0), (2,1), (1,1), (1,2), (0,2), (0,3), (1,3), (1,4), (0,4), (0,5), (1,5), (2,5), (2,4), (3,4), (4,4), (4,3), (3,3), (3,2), (4,2), (5,2), (5,1), (6,1), (7,1), (8,1), (8,0), (9,0), (9,1), (9,2), (9,3), (9,4), (9,5), (8,5), (7,5), (7,6).

- Resultado encontrado:



5. Conclusão:

A partir de tudo mostrado acima, pode-se concluir que a implementação do trabalho transcorreu sem maiores problemas e os resultados ficaram dentro do esperado. A realização dessa atividade além de gerar maior conhecimento no tema, ainda foi importante para afixar ainda mais a parte da matéria relacionada a operações em pilhas, alocação dinâmica, manipulação de imagens.

A principal dificuldade se deu pelo fato de inicialmente eu estar implementando o código no sistema operacional Windows e durante o período de testes ter mudado para Linux, com isso foi necessário a realização de algumas adequações no código que causaram uma certa dificuldade, porém essa pôde ser superada com sucesso.

Referências

[1] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 2ª Edição,Editora Thomson, 2004.

[2] <https://docs.gimp.org/2.8/pt_BR/plug-in-convmatrix.html> Acesso em: 16/04/17

Anexos:

Listagem dos programas:

- TP1.h
- TP1.c
- main.c