Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

DCC004 - ALGORITMOS E ESTRUTURAS DE DADOS II - TURMA TW

Trabalho Prático 1 – Skip List

Guilherme Saulo Alves

Professoras - Raquel Minardi e Raquel Prates

1. Introdução

O proposito do Trabalho consiste em implementar estruturas em um tipo abstrato de dados para representar uma *Skip List.*

Basicamente é uma variação da lista encadeada ordenada simples onde a cada célula são acrescentados vários ponteiros para os elementos à frente, de modo que a pesquisa possa rapidamente pular partes da lista.

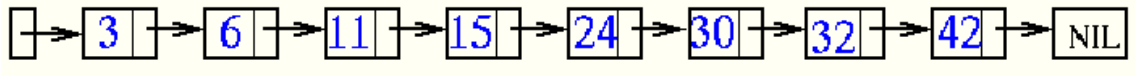


Fig. 1. Diagrama ilustrativo de uma lista encadeada ordenada simples

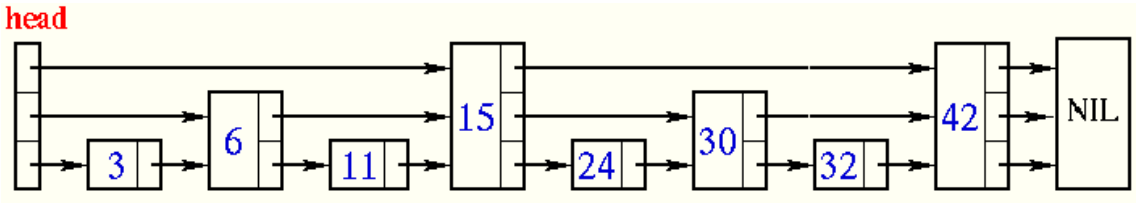


Fig.2. Diagrama ilustrativo de uma *Skip List.*

2. Implementação

Uma das primeiras implementações foi criar um tipo para representar a célula, contendo sua chave e os demais atributos e ponteiros necessários.

Cada função e estrutura serão explicadas detalhadamente ao longo da documentação.

2.1. Considerações Iniciais

O ambiente de desenvolvimento do código foi no CodeBlocks IDE 10.05 no Sistema Operacional Ubuntu Linux 12.04. Foi usado o compilador GNU GCC Compiler via linha de comando.

O código está dividido em três arquivos principais: *main.c* com o programa principal, skip.*c* e skip.*h.* com a implementação do TAD *Skip List.*

2.2. Estruturas e Funções

As estruturas e funções que formam o TAD serão apresentadas a seguir:

2.2.1 Estrutura *TipoChave*

A estrutura é formada por uma variável do tipo int chamada TipoChave, que representa o valor da chave que sera inserido em uma célula.

2.2.2 Estrutura *TipoItem*

A estrutura é formada por uma variável do TipoChave chamada Chave que armazena a chave a ser inserida na célula.

2.2.3 Estrutura *TipoCelula *Apontador*

A estrutura é formada por um apontador para uma célula que será criada posteriormente. Obs.: Nem todos os compiladores aceitam esse procedimento e, no caso, precisaria de algumas implementações.

2.2.4 Estrutura *TipoCelula Celula*

A estrutura é formada por uma variável do TipoItem Item e por um vetor de tamanho 5 do tipo Apontador, que possui o endereço da próxima célula a cada camada.

2.2.5 Estrutura *TipoLista*

A estrutura possui um Apontador para a primeira célula da lista e para a ultima.

2.2.6 void FLVazia(TipoLista *Lista);

A função FLVazia inicia a lista criando sua estrutura básica de funcionamento.

2.2.7 int Vazia(TipoLista Lista);

Verifica se a lista está vazia, retornando 0, ou se contém algum elemento, retornando 1. A lista está vazia quando todos os ponteiros do próximo elemento da cabeça apontam para NULL.

2.2.8 int Busca(TipoChave chave, TipoLista *L,

Apontador *ListaH, int imprimir);

A busca será feita pela chave passada por parâmetro, retornando 1 caso o elemento seja encontrado e 0 caso contrário. Em caso de sucesso, uma lista de ponteiros (ListaH) contendo o ponteiro de cada nível deve ser retornado. ListaH[h] guarda o ponteiro anterior ao elemento chave buscado ou a posição anterior a possível inserção caso ele não exista. A função busca serve como base para as funções insere e remove, pois ela fornece os ponteiros das células anteriores em cada nível hierárquico à posição de inserção e remoção de uma célula. Caso o parâmetro imprimir seja 1, é necessário imprimir o caminha da busca até o elemento a ser encontrado.

2.2.9 int Insere(TipoItem x, int h, TipoLista *Lista);

Esta função deve inserir um novo elemento x na *Skip List* mantendo a ordem da lista. É viável chamar a função Busca, pois está retornará um vetor com todos os ponteiros (de todas as camadas) para a posição que a chave deve ser inserida. A inserção com sucesso deve retornar 1, ou 0 caso contrário.

2.2.10 int Remove(TipoChave chave, TipoLista *L);

Esta função deve remover o elemento indicado com a chave. Dentro da função é chamada a Busca, pois confirma a existência ou não do elemento indicado pela chave e fornece todos os ponteiros anteriores que apontam para o elemento a ser excluído. A remoção com sucesso deve retornar 1, ou 0 caso contrário.

2.2.11 void Imprime(TipoLista Lista, int h);

A função lista todos os elementos na camada h, contendo o par chave/hierarquia (c h).

2.2.12 void ImprimeTodos(TipoLista Lista);

A função lista todos os elementos por nível hierárquico, começando pelo nivel mais superior até a camada 0.

2.3 Programa Principal

O programa principal lê da entrada padrão, os arquivos de entrada para vários testes. A saída é direcionada para o dispositivo de saída padrão (stdout).

O primeiro passo depois da declaração das variáveis é ler o carácter que corresponde ao teste realizado e comparamos se os nomes são iguais aos definidos do switch. Tais correspondem à I-(Inserir) R-(Remover) B-(Busca) P-(Imprime Nível h) A-(Imprime Todos Níveis).

Em seguida é feito a leitura dos parâmetros que correspondem a cada teste lido (item e nível para 'I', item.Chave para 'R' , item.Chave para 'B' e nível para 'P') e chamamos as funções para realização dos testes.

3. Análise de Complexidade

A análise de complexidade em tempo de execução das funções implementadas será feita em função da variável n que representa o número de vezes que a função é chamada (notação O).

Também é apresentado um estudo de complexidade mais teórico para busca, inserção e remoção.

Função FLVazia: A função executa alguns comandos O(1) e depois entra em um loop que é executado n vezes. Portando **O(n)**.

Função Vazia: Essa função é bem simples e não envolve nenhum laço de repetição. Portanto, é executado 1 vez e sua complexidade é constante dada por **O(1).**

Função Imprime: A função executa alguns comandos O(1) e depois entra em um loop que é executado n vezes. Dentro do loop, são executados 2 comandos O(1). A complexidade até aqui é O(1)+O(n+2) = **O(n).**

Função ImprimeTodos: A função chama a função Imprime dentro de um 'for'. Como o função 'Imprime' é executada n vezes, sua complexidade será $O((n)^2)$ = **$O(n^2)$**.

Função Busca: Para buscar uma dada chave começamos pelo topo da lista e seguimos ao longo de cada lista encadeada até encontrar o elemento o qual é menor ou igual à chave. Portanto, fica fácil perceber que o tempo de execução deste algoritmo é proporcional ao numero de células distintas visitadas somadas ao numero de níveis descidos.

(1) Em uma *Skip List* com n elementos, o numero esperado de níveis descidos é O(log n).

(2) Na busca, o número esperado de células distintas visitadas por nível é O(1) e em todos niveis O(log n).

Por fim, somando o numero esperado de níveis descidos (1) com o numero esperado de células visitadas (2), uma busca em uma Skip List com n elementos gasta tempo esperado **O(log n).**

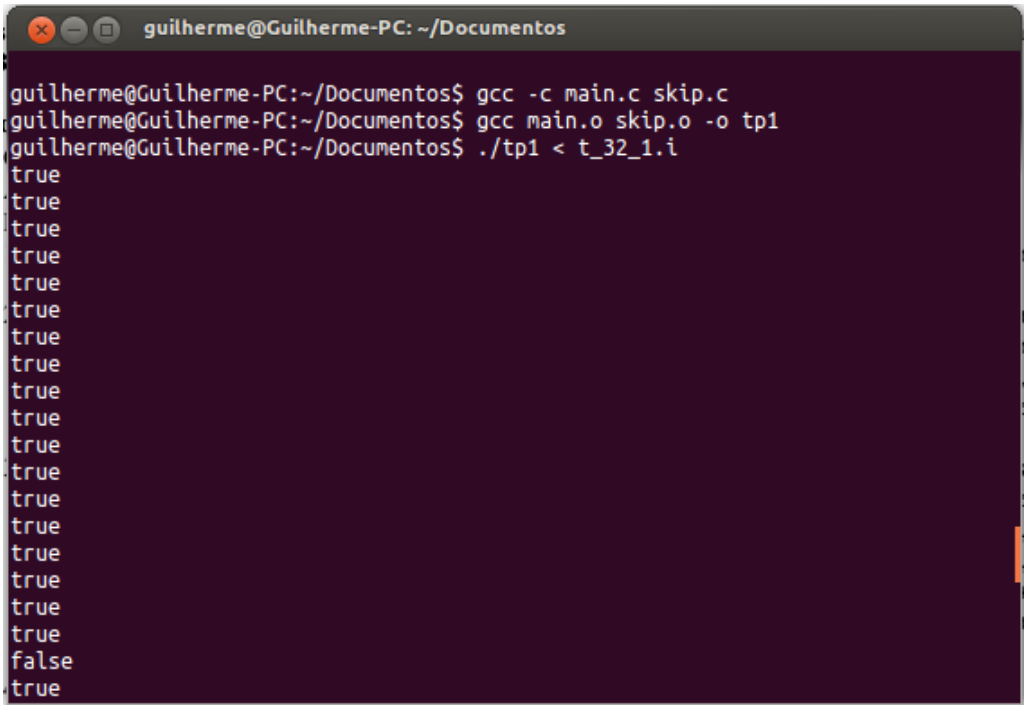
Função Inserir: A busca inicial tem tempo esperado O(log n). O tempo consumido nos laços da função inserir é proporcional à quantidade de níveis da *Skip List*. O tempo consumido nas outros comandos é O(1). Portanto, o algoritmo da função inserir gasta tempo esperado **O(log n).**

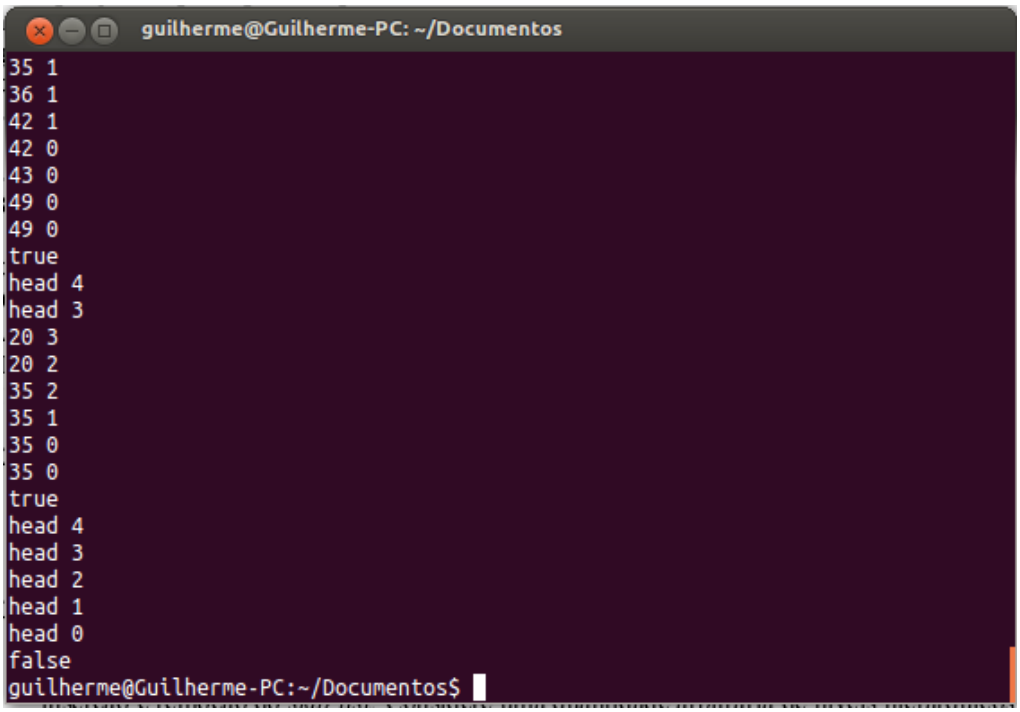
Função Remover: A analise é similar ao caso anterior, mas devemos considerar os laços da função remover. Da mesma forma, o tempo consumido nesses laços é proporcional à quantidade de níveis da *Skip List*. Portanto, o algoritmo remover gasta tempo **O(log n).**

Programa principal: O programa principal executa alguns comandos O(1) antes de aparecer o loop para lê cada teste. É executado um teste a cada laço do loop principal. A ordem de complexidade de tempo dentro do '*while*' é o teste de maior ordem de complexidade existente O(n). Portanto, como o loop é executado n vezes, a ordem de complexidade do programa principal como um todo é **$O(n^2)$.**

4. Testes

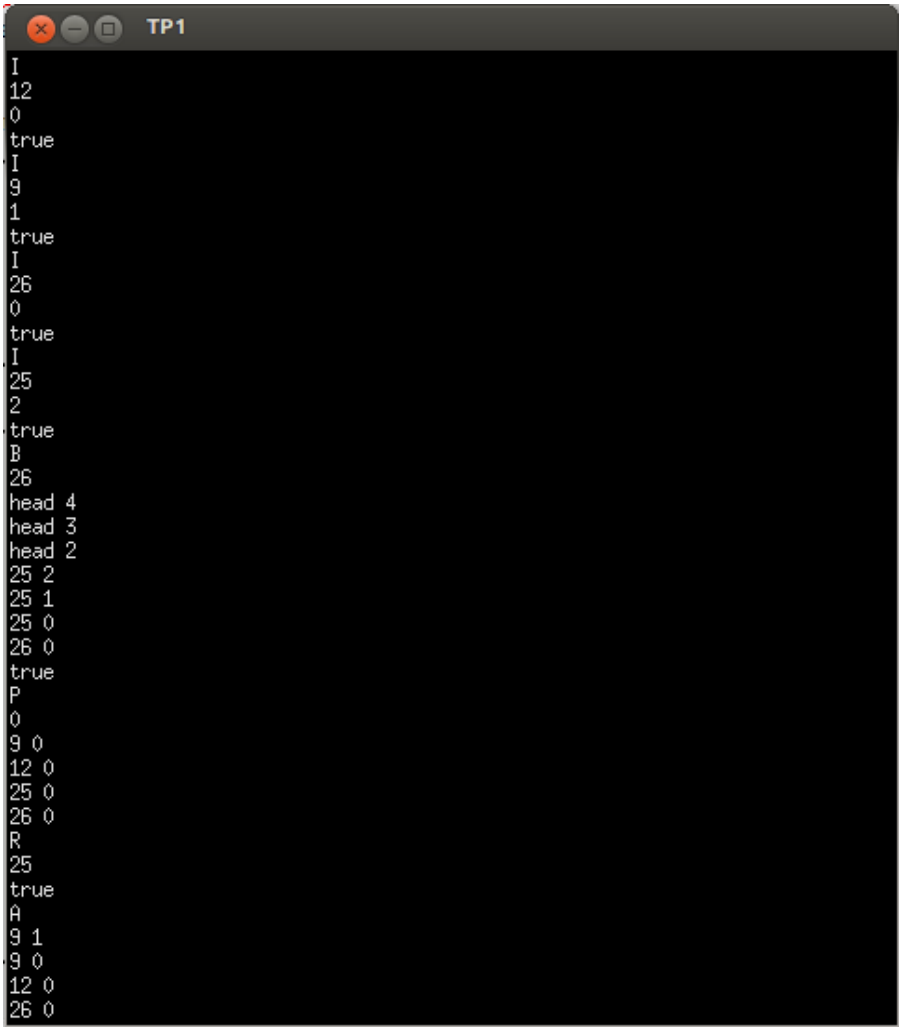
Vários testes foram realizados com o programa de forma a verificar o seu funcionamento. Os testes foram realizados em um Atom CPU D525, com 2 Gb de memória. A figura abaixo mostra a saída de uma execução típica quando executado na linha de comando.





Também foram realizados testes no *CodeBlocks* durante a elaboração do trabalho com '*printf*' para verificar os valores de entrada e saída de algumas variáveis. O exemplo de entrada usado foi o do enunciado do TP:

Entrada	Saída
I 12 0	true
I 9 1	true
I 26 0	true
I 25 2	true
B 26	head 4
P 0	head 3
R 25	head 2
A	25 2
	25 1
	25 0
	26 0
	true
	9 0
	12 0
	25 0
	26 0
	true
	9 1
	9 0
	12 0
	26 0



5. Conclusão

Com elaboração desse trabalho foi possível relembrar as técnicas de programação aprendidas na disciplina de AEDS 2 e também fixar os novos conceito de lista encadeadas e alocação dinâmica.

A principal dificuldade encontrada foi na implementação dos *'printfs'* dentro da função busca, pois em muitos casos quando era preciso acessar memoria, o compilador acusava ***segmentation fault*** e no caso era porque estava acessando um endereço que não existia.

Por fim, foi visto que existe um algoritmo mais eficiente para implementação de uma lista encadeada ordenada, como a *Skip List.*

6. Referências

[1] McGill University - School of Computer Science - DATA STRUCTURES AND ALGORITHMS - Project #25: SKIP LISTS - http://www.sable.mcgill.ca/~dbelan2/cs251/skip_lists.html - Acessado em 18 de Outubro de 2013.