Trabalho prático 2: Hashbin.

Algoritmos e Estruturas de dados II (2021/1)

1 Introdução

Neste trabalho você irá criar um algoritmo de pesquisa em memória primária mais eficiente. Para isso, você deverá implementar o $Hash$ com tratamento de colisão com a seguinte modificação: ao invés de usar listas encadeadas para tratar colisões em cada endereço da tabela, você deverá implementar uma árvore binária de busca. Por exemplo, suponha que você tenha uma tabela $Hash$ com número de entradas $M = 3$ e insera os seguintes números na tabela, na ordem que aparecem: [3,4,1,0,2,5,9,6,7,8,10]. Sua estrutura final deverá ficar igual a mostrada pela figura 1, com uma tabela $Hash$ com três endereços e, em cada endereço, existe uma árvore binária de busca para tratar colisões.

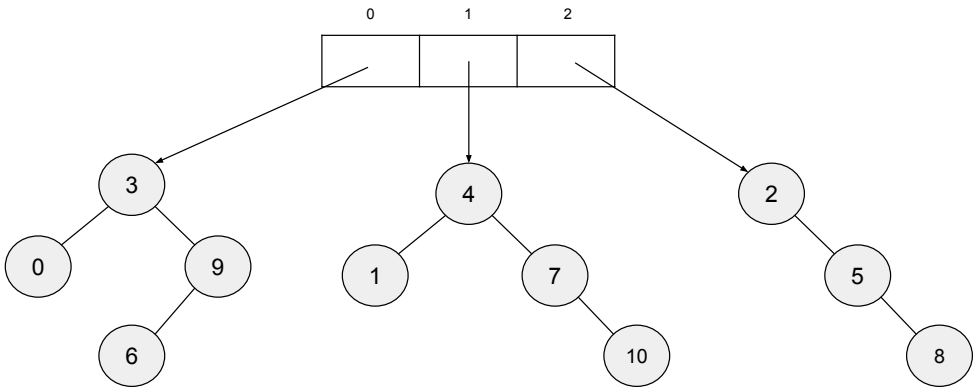

Figura 1: Exemplo de estrutura gerada para $[3, 4, 1, 0, 2, 5, 9, 6, 7, 8, 10]$ e $M = 3$.

2 Instruções sobre a Implementação

Seu código deverá ser escrito em C e deverá conter, pelo menos, os seguintes arquivos: *arvore_binaria_busca.c*, *hash.c* e *main.c*. Você pode criar arquivos de cabeçalho para os arquivos definidos (por exemplo, *quicksort.h*) para tornar o código mais modular. Porém isso não é necessário.

3 Formato de Entrada e Saída

O trabalho deve ser executado da seguinte forma, ou seja, sem argumentos:

```
./tp
```

Seu programa deverá ler a entrada da entrada padrão ($stdin$) e gravar a saída na saída padrão ($stdout$). A **entrada** possui múltiplos casos de teste. Cada caso de teste é organizado da seguinte forma. A primeira linha possui o tamanho da tabela hash M ($M < 10^3$) e o número de elementos N que serão inseridos na estrutura, nesta ordem e separados por um espaço. A segunda linha possui os N números inteiros positivos, separados por espaço. A terceira linha contém um inteiro X ($[0, M-1]$) informando a posição da tabela Hash que deverá ser impressa na saída. A impressão da árvore binária deverá ser feita usando o caminhamento **pré-ordem** com um espaço simples entre as chaves impressas.

Exemplo de Entrada:

```
5 20
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
3
3 11
3 4 1 0 2 5 9 6 7 8 10
0
0 0
```

Exemplo de Saída:

```
3 8 13 18
3 0 9 6
```

4 O que entregar

Você deve submeter uma documentação de até 10 páginas, em formato PDF, contendo uma descrição de como seu código foi implementado. **Sua documentação deverá ter uma análise de complexidade de tempo do novo método de pesquisa desenvolvido e testes experimentais que comparem o tempo de execução da técnica desenvolvida com a técnica de $Hash$ com tratamento de colisões por listas encadeadas** . Siga as diretrizes sobre como fazer uma documentação que foram disponibilizadas no Portal Didático.

Além da documentação, você deve submeter um arquivo compactado no formato *.tar.gz* ou *.gzip* contendo todos os arquivos de código (*.c* e *.h*) que foram implementados. Além dos arquivos de código, esse arquivo compactado deve incluir um *makefile*. Consulte os tutoriais disponibilizados no Portal Didático para descobrir como fazer um. Finalmente, lembre-se de não incluir pastas no arquivo compactado.

5 Avaliação

Para que seu trabalho seja elegível para correção, seu código deve acertar todos os casos de teste do exemplo desta especificação! Seu trabalho será avaliado quanto a documentação escrita e à implementação. Eis uma lista **não exaustiva** de critérios de avaliação utilizados.

Documentação

Introdução Inclua uma breve explicação do problema que está sendo resolvido no seu trabalho e um resumo da sua solução.

Solução do Problema Você deve descrever a solução do problema de maneira clara e precisa. Para tal, artificios como pseudo-códigos, exemplos ou figuras podem ser úteis. Note que documentar uma solução não é o mesmo que documentar seu código. **Não** é preciso incluir trechos de código em sua documentação nem mostrar detalhes de sua implementação, exceto quando os mesmos influenciem o seu algoritmo principal, o que se torna interessante. Por exemplo, é interessante discutir quais estruturas de dados foram utilizadas no *bucketsort* para mostrar como os dados foram organizados em memória.

Análise de Complexidade Inclua uma análise de complexidade de tempo dos principais algoritmos implementados e uma análise de complexidade de espaço das principais estruturas de dados de seu programa. Cada complexidade apresentada deverá ser devidamente **justificada** para que seja aceita. Apesar da complexidade dos algoritmos implementados serem conhecidos na teoria, a complexidade pode variar de acordo com a implementação. Um dos principais pontos a serem abordados na documentação é a complexidade de tempo do *bucketsort* e de espaço dos outros algoritmos. Busque entender e explicar com suas próprias palavras. Somente dessa forma você aprenderá um dos tópicos essenciais em computação.

Avaliação Experimental Sua documentação deve incluir os resultados de experimentos que avaliem o tempo de execução de seu código em função de características da entrada. Cabe a você gerar entradas para esses experimentos.

Limite de Tamanho Sua documentação deve ter no máximo 10 páginas. Todo o texto a partir da página 11, se existir, será desconsiderado.

Guia Há um guia e exemplos de documentação disponíveis no Portal Didático.

Implementação

Linguagem e Ambiente

- Implemente tudo na linguagem **C**. Você pode utilizar qualquer função da biblioteca padrão da linguagem em sua implementação, mas **não** deve utilizar outras bibliotecas e muito menos funções que realizem o trabalho para você como o *qsort*. **Trabalhos em outras linguagens de programação serão zerados. Trabalhos que utilizem outras bibliotecas também.**

- Os testes serão executados em **Linux**. Portanto, garanta que seu código compila e roda corretamente nesse sistema operacional. A melhor forma de garantir que seu trabalho rode em Linux é escrever e testar o código nele. Você pode fazer o download de uma variante de Linux como o **Ubuntu** (`http://www.ubuntu.com`) e instalá-lo em seu computador ou diretamente ou por meio de uma máquina virtual como o **VirtualBox** (`https://www.virtualbox.org`). Há vários tutoriais sobre como instalar Linux disponíveis na web.

Casos de teste Seu trabalho será executado em um conjunto de entradas de teste.

- Essas entradas *não* serão disponibilizadas para os alunos até que a correção tenha terminado. Faz parte do processo de implementação testar seu próprio código.
- Você perderá pontos se o seu trabalho produzir saídas incorretas para algumas das entradas ou não terminar de executar dentro de um tempo limite pré-estabelecido. Esse tempo limite é escolhido com alguma folga. Garanta que seu código roda a entrada de pior caso em no máximo alguns minutos e você não terá problemas.
- A correção será **automatizada**. Esteja atento ao formato de saída descrito nessa especificação e o siga precisamente. Por exemplo: se a saída esperada para uma certa entrada o número 10 seguido de uma quebra de linha, você deve imprimir apenas isso. Imprimir algo como "A resposta e: 10" contará como uma saída **errada**.
- Os exemplos mostrados nessa especificação são parte dos casos de teste.
- **Você deve entregar algum código e esse código deve compilar e executar corretamente para, pelo menos, o *exemplo* de entrada desta especificação. Se isso não ocorrer, a nota do trabalho prático será zerada.**

Alocação Dinâmica Você também deve liberar *tudo* o que for alocado utilizando `free()`. Verifique o tutorial disponibilizado no Portal Didático sobre como utilizar o Valgrind para verificar vazamentos de memória.

Makefile Inclua um makefile na submissão que permita compilar o trabalho.

Qualidade do Código Seu código deve ser bem escrito:

- Dê nomes a variáveis, funções e estruturas que façam sentido.
- Divida a implementação em módulos que tenham um significado bem definido.
- Acrescente comentários sempre que julgar apropriado. Não é necessário parafrasear o código, mas é interessante acrescentar descrições de alto nível que ajudem outras pessoas a entender como sua implementação funciona.
- Evite utilizar variáveis globais.
- Mantenha as funções concisas. Seres humanos não são muito bons em manter uma grande quantidade de informações na memória ao mesmo tempo. Funções muito grandes, portanto, são mais difíceis de entender.
- Lembre-se de indentar o código. Escolha uma forma de indentar (tabs ou espaços) e mantenha-se fiel a ela. Misturar duas formas de indentação pode fazer com que

o código fique ilegível quando você abri-lo em um editor de texto diferente do que foi utilizado originalmente.

- Evite linhas de código muito longas. Nem todo mundo tem um monitor tão grande quanto o seu. Uma convenção comum adotada em vários projetos é não passar de 80 caracteres de largura.
- **Tenha bom senso**.

6 Considerações Finais

- Essa especificação não é isenta de erros e ambiguidades. Portanto, se tiver problemas para entender o que está escrito aqui, pergunte ao professor da disciplina ou aos seus colegas de classe.
- Não existe política de atraso! Após a data de entrega, a nota é zero.
- Você **não** precisa de utilizar uma IDE como Netbeans, Eclipse, Code::Blocks ou VScode para implementar esse trabalho prático. No entanto, se o fizer, **não** inclua os arquivos que foram gerados por essa IDE em sua submissão.
- **Seja honesto**. Você não aprende nada copiando código de terceiros nem pedindo a outra pessoa que faça o trabalho por você. Se a cópia for detectada, sua nota será zerada e seu caso será comunicado ao colegiado do curso, conforme resolução da UFSJ.