Projeto de Programação I
Algoritmos de Busca

Objetivo

O objetivo deste trabalho é confirmar ou corrigir a hipótese de que são significativas as diferenças entre os tempos de execução das funções de busca sequencial, binária e ternária; e que as versões recursivas são mais rápidas do que as iterativas para todos os casos. Para isso, serão implementados os três modelos de busca em ambas as versões iterativa e recursiva. As implementações deverão ser testadas para diferentes cargas (*workloads*) a fim de permitir a análise dos resultados e, com isso, concluir sobre a hipótese inicialmente dada.

Algoritmos de busca

O problema da *busca em um vetor sequencial* pode ser definido como:

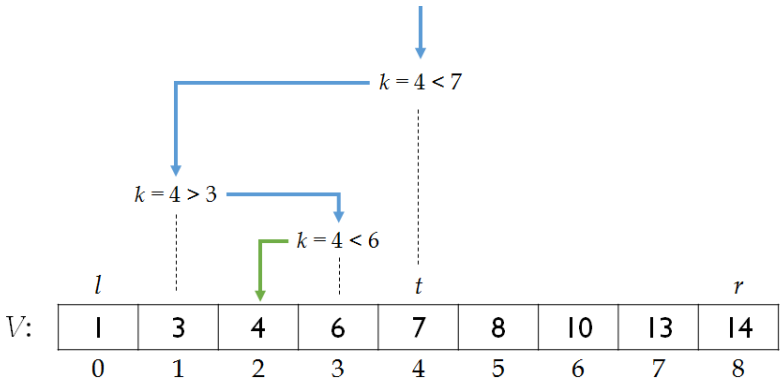
> *Dado um conjunto de valores previamente armazenados em um vetor V nas posições V*[*l*], *V*[*l*+1], ..., *V*[*r*], *sendo* $0 \leq l \leq r \in \mathbb{N}^0$, *verificar se um valor (chave) k está entre esse conjunto de valores. Em caso positivo, indicar qual o índice da localização de k em V ou retornar* -1 *caso k não seja encontrado.*

Para solucionar esse problema, existem diversos algoritmos com características e implementações peculiares. Conhecer tais algoritmos e suas características orienta qual algoritmo melhor se adequa a uma determinada situação na qual precisa-se realizar o procedimento de busca. Dentre esses algoritmos estão os de busca *sequencial* (ou *linear*), *binária* e *ternária*.

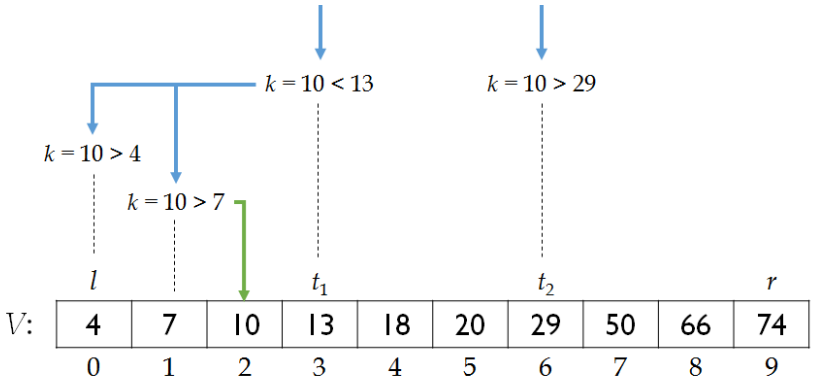
O algoritmo de busca sequencial percorre todo o vetor *V* e comparando elemento por elemento até que a chave buscada seja encontrada ou que tenha se chegado ao final do vetor. Apesar da sua implementação extremamente simples, esse algoritmo não é o mais eficiente em termos de tempo principalmente quando se trata de um vetor com um grande número de elementos. No pior caso, a chave a ser buscada encontra-se na última posição do vetor e, até encontra-la, todo o vetor precisará ser percorrido. Por essa razão, a busca sequencial não é muito aplicada na prática em comparação a outros procedimentos de busca existentes.

A busca binária aplica uma estratégia conhecida como "dividir para conquistar", ou *divide-and-conquer*, em Inglês. Partindo do pressuposto que o vetor está ordenado, a ideia é dividi-lo sucessivamente em duas partes de mesmo tamanho (aproximado) e determinar em qual dessas partes a chave a buscar se encontraria, para então concentrar a busca nessa parte. Mais especificamente, considerando que *l* define o índice do limite esquerdo de um vetor *V*, *r* o índice do

limite direito de V e $t = \left\lfloor \frac{l + r}{2} \right\rfloor$ é o índice que divide V em duas partes, o algoritmo verifica se a chave procurada k está em $V[t]$. Se não estiver, o algoritmo analisa o valor de k em relação a $V[t]$ para decidir se irá realizar a busca sobre $[V[l]; V[t])$, se $k < V[t]$, ou sobre $(V[t]; V[r]]$, se $k > V[t]$. Por exemplo, considere o exemplo abaixo no qual busca-se pela chave $k = 4$ em um vetor V com $n = 9$ elementos. Ao dividir o vetor em duas partes, obtém-se o limite $t = 4$ e, após analisar o valor de k em relação a $V[t]$, a busca é realizada sobre a primeira metade do vetor, ou seja, $[V[l = 0]; V[t = 4])$. Essa parte do vetor é novamente dividida em duas partes para continuar o procedimento de busca.



Por fim, a busca ternária segue um princípio similar ao da busca binária com a diferença que o vetor V é dividido em três partes de mesmo tamanho (aproximado), ao invés de apenas duas partes. Considerando que l define o índice do limite esquerdo de V, r o índice do limite direito de V, t_1 o índice do limite do primeiro terço de V e t_2 o índice do limite do segundo terço de V, o algoritmo verifica se a chave procurada k está em $V[t_1]$ ou $V[t_2]$. Se não estiver, o algoritmo analisa o valor de k em relação a $V[t_1]$ e/ou $V[t_2]$ para decidir sobre qual das três partes $[V[l]; V[t_1])$, $(V[t_1]; V[t_2])$ ou $(V[t_2]; V[t_r]]$ a busca ternária será realizada. Por exemplo, considere o exemplo abaixo no qual busca-se pela chave $k = 10$ em um vetor V com $n = 10$ elementos. Ao dividir o vetor em três partes, obtém-se os limites $t_1 = 3$ e $t_2 = 6$. Após analisar o valor de k em relação a $V[t_1]$, o algoritmo procede a busca no primeiro terço de V, ou seja, $[V[l = 0]; V[t_1 = 3])$, dado que $k < V[t_1]$. De maneira similar, se $k = 66$, o algoritmo realiza a busca sobre a última parte do vetor, ou seja, $[V[t_2 = 6]; V[r = 9])$, uma vez que $k > V[t_2]$.



Tarefas

As tarefas a serem realizados neste exercício de programação consistem em três partes: (1) *implementação*, na qual você deverá projetar e implementar os algoritmos de busca anteriormente descritos; (2) *experimentação*, na qual você deverá realizar sucessivas execuções dos algoritmos e observar o seu comportamento, a fim de fazer uma análise comparativa entre eles, e; (3) *relato*, na qual você deverá elaborar um relatório descrevendo as atividades que foram realizadas, os resultados observados e as conclusões obtidas.

Implementação

Para resolver o problema da busca em um vetor sequencial, você deverá projetar e implementar os três modos de busca (sequencial, binária e ternária) tanto em sua forma recursiva quanto em sua forma iterativa, perfazendo um total de seis algoritmos a serem analisados. Cada algoritmo deverá ser implementado na forma de uma função que recebe três parâmetros: (1) um valor inteiro representando a chave a ser buscada no vetor; (2) um vetor de inteiros sobre o qual será realizada a busca, e; (3) um valor inteiro representando o tamanho do vetor. Dessa forma, os seguintes protótipos deverão ser obrigatoriamente seguidos:

```
int busca_sequencial_ite(int chave, int* vetor, int tamanho); // Sequencial iterativa
int busca_sequencial_rec(int chave, int* vetor, int tamanho); // Sequencial recursiva
int busca_binaria_ite(int chave, int* vetor, int tamanho);    // Binaria iterativa
int busca_binaria_rec(int chave, int* vetor, int tamanho);    // Binaria recursiva
int busca_ternaria_ite(int chave, int* vetor, int tamanho);   // Ternaria iterativa
int busca_ternaria_rec(int chave, int* vetor, int tamanho);   // Ternaria recursiva
```

O código deverá ser devidamente comentado e anotado para dar suporte à geração automática de documentação no formato de páginas Web (HTML) utilizando a ferramenta Doxygen. Para maiores informações, você poderá acessar a página do Doxygen na Internet (http://www.doxygen.org/) bem como consultar o documento extra disponibilizado na Turma Virtual do SIGAA com algumas instruções acerca do padrão de documentação e uso da ferramenta. Um exemplo de boa documentação para a função que implementa a busca sequencial de forma iterativa seria:

```
/**
 * @brief     Funcao que implementa uma busca sequencial de forma iterativa
 * @details   A busca sequencial iterativa percorre o vetor comparando elemento por
 *            elemento ate que a chave buscada seja encontrada ou que se tenha chegado
 *            ao final do vetor. Este procedimento nao requer que o vetor esteja ordenado.
 *
 * @param     chave Chave de busca
 * @param     vetor Vetor de inteiros sobre o qual a busca sera realizada
 * @param     tamanho Tamanho do vetor
 * @return    Indice para a chave, caso o elemento exista no vetor, ou -1 caso contrario
 */
int busca_sequencial_ite(int chave, int* vetor, int tamanho) {
    // Corpo da funcao
}
```

O programa principal deverá ser executado via linha de comando da seguinte forma:

```
$ ./busca <total_elementos> <chave> <algbusca>
```

sendo:

- `<total_elementos>` um valor inteiro definindo a quantidade total de elementos do vetor de busca V a ser alocado dinamicamente, de modo que $V \rightarrow V[0] \ldots V$[`<total_elementos>`-1];
- `<chave>` um valor inteiro a ser buscado no vetor V, e;
- `<algbusca>` uma *string* identificando o algoritmo de busca a ser executado, podendo assumir um dos seguintes seis valores: `BSI` – *Busca Sequencial Iterativa*; `BSR` – *Busca Sequencial Recursiva*; `BBI` – *Busca Binária Iterativa*; `BBR` – *Busca Binária Recursiva*; `BSR` – *Busca Sequencial Recursiva*; `BTI` – *Busca Ternária Iterativa*, e; `BTR` – *Busca Ternária Recursiva*.

O programa principal deverá ler os parâmetros, passados através da linha de comando, e executar o algoritmo de busca indicado pelo parâmetro `<algbusca>`. Como resultado, o programa deverá indicar se a chave a ser buscada foi (neste caso sendo indicada qual a posição da chave no vetor) ou não encontrada no vetor e o tempo gasto pela função de busca. Para medir o tempo de execução do programa, você poderá fazer uso da *Chrono Time Library*, uma biblioteca presente no padrão C++11. Documentação sobre essa biblioteca pode ser consultada na Internet através dos *links* http://www.cplusplus.com/reference/chrono/ e http://en.cppreference.com/w/cpp/chrono. Um bom exemplo do uso dessa biblioteca também pode ser encontrado na Internet, no endereço https://www.guyrutenberg.com/2013/01/27/using-stdchronohigh_resolution_clock-example/. É importante lembrar que, para que seja possível utilizar essa biblioteca, você deverá adicionar a diretiva de compilação `-std=c++11` ao compilar o seu programa.

Internamente, a definição do algoritmo de busca a ser executado deverá ser implementada por meio de *ponteiros para função*, através de uma função com o seguinte protótipo

```
int seleciona_busca(const char* opcao, <ponteiro_funcao>);
```

em que `opcao` refere-se ao tipo de busca (por exemplo, "`BTI`") e `<ponteiro_funcao>` é um ponteiro para a função de busca a ser executada.

A geração do *workload* deverá ser dada pelo trecho de código:

```
for (int i = 0; i < N; i++) {
    v[i] = i*2;
}
```

onde `N` é o valor de `<total de elementos>`. Com isso, será gerado um *workload* de valores pares, permitindo que você faça buscas do tipo *hit* (acerto) e *miss* (falha) no vetor de busca, pois sabe-se que a eficiência dos métodos de busca pode variar entre um *hit* e um *miss*.

Experimentação

A fim de permitir uma avaliação consistente, você deverá executar seu experimento pelo menos cinco vezes com casos do tipo *hit* e outras cinco vezes com casos do tipo *miss*, para cada algoritmo implementado e para cada *workload*. Os *workloads* deverão variar, sendo inicialmente de 2 milhões de inteiros e reduzidas pela metade até 1 (ou seja, 2 milhões, 1 milhão, 500 mil, 250 mil, ..., 1).

Seguem abaixo dois exemplos de execução do programa com valores hipotéticos, a primeira para uma busca com acerto (*hit*) e a segunda para uma busca com falha (*miss*):

```
$ ./busca 1000000 48 BTI
A chave 48 foi encontrada na posicao 14 do vetor.
Algoritmo usado: Busca Ternaria Iterativa
Tempo: 0.300 s.

$ ./busca 500000 17 BSR
A chave 17 nao foi encontrada no vetor.
Algoritmo usado: Busca Sequencial Recursiva
Tempo: 179.5 s.
```

Confirme a precisão dos resultados obtidos com o uso do *profile* (*gprof*). Descreva em seu relatório se houveram diferenças significativas, indicando inclusive qual a causa provável.

Relato

Uma vez realizadas as tarefas de implementação e experimentação, você deverá elaborar um relatório técnico contendo, no mínimo, as seguintes seções:

1) **Introdução –** Explicar o propósito do relatório.
2) **Detalhes de Implementação –** Descrever como foi feita a sua implementação em termos de arquivos, funções, etc. e como o programa funciona de uma maneira geral.
3) **Metodologia –** Indicar o método adotado para realizar os experimentos com os algoritmos e analisar os resultados obtidos. Por exemplo, você deve apresentar a caracterização técnica do computador utilizado (processador, sistema operacional, quantidade de memória RAM), a linguagem de programação e a versão do compilador empregados, os cenários considerados, entre outras informações. Você também deverá descrever qual o procedimento adotado para gerar os resultados, como a comparação entre os algoritmos foi feita, etc.
4) **Resultados –** Apresentar os resultados obtidos na forma de gráficos de linha e tabelas. Para cada algoritmo deverão ser apresentados uma tabela contendo os tempos mínimo, médio e máximo obtidos para cada *workload*, nos casos de acerto e de falha. Adicionalmente, deverão ser apresentados dois gráficos de linha mostrando apenas os tempos médios obtidos nas execuções de cada algoritmo em cada *workload*, sendo um gráfico para os casos de acerto (*hit*) e outro para os casos de falha (*miss*). Para a produção dos gráficos, você pode utilizar uma ferramenta de sua escolha, a exemplo do Microsoft Office Excel, LibreOffice Calc, *gnuplot*, R, Matlab, etc.
5) **Discussão –** Discutir os resultados, ou seja, o que foi possível concluir através dos resultados obtidos através dos experimentos.

Orientações gerais

Você deverá atentar para as seguintes observações gerais no desenvolvimento deste trabalho:

1) Apesar da completa compatibilidade entre as linguagens de programação C e C++, seu código fonte não deverá conter recursos da linguagem C nem ser resultante de mescla entre as duas linguagens, o que é uma má prática de programação. Dessa forma, deverão ser utilizados estritamente recursos da linguagem C++.

2) Durante a compilação do seu código fonte, você deverá habilitar a exibição de mensagens de aviso (*warnings*), pois elas podem dar indícios de que o programa potencialmente possui problemas em sua implementação que podem se manifestar durante a sua execução.

3) Aplique boas práticas de programação. Codifique o programa de maneira legível e documente-o adequadamente na forma de comentários. Como anteriormente instruído, o código fonte deverá ser anotado para dar suporte à geração automática de documentação utilizando o Doxygen.

4) Busque desenvolver o seu programa com qualidade, garantindo que ele funcione de forma correta e eficiente. Pense também nas possíveis entradas que poderão ser utilizadas para testar apropriadamente o seu programa e trate adequadamente possíveis entradas consideradas inválidas.

5) Para melhor organização do seu código, implemente diferentes funções e faça a separação da implementação do programa entre arquivos cabeçalho (`.h`) e corpo (`.cpp`).

Autoria e política de colaboração

O trabalho poderá ser feito individualmente ou em equipe composta por no máximo dois estudantes, sendo que, neste último caso, é importante, dentro do possível, dividir as tarefas igualmente entre os integrantes da equipe. A fim de estimular o desenvolvimento colaborativo, uma sugestão de divisão de tarefas é que cada membro da equipe esteja responsável pela implementação dos arquivos cabeçalho e corpo de pelo menos três dos seis algoritmos de busca, sendo os códigos fonte feitos individualmente em um repositório Git local e, posteriormente, unificados em um repositório Git remoto. Cada equipe terá um repositório individual no Gitlab do IMD-UFRN (http://projetos.imd.ufrn.br/), no qual deverão ser disponibilizados todos os arquivos referentes ao programa implementado. O endereço do repositório será informado pelo professor a cada equipe, devendo todos os integrantes terem feito previamente seu cadastro no Gitlab.

A atividade de cada integrante da equipe deverá ser registrada através de *commits* sobre o repositório Git, que será examinado pelo professor durante a avaliação do trabalho. Além disso, a critério do professor, qualquer equipe pode ser convocada para uma entrevista cujo objetivo é confirmar a autoria do trabalho desenvolvido e determinar a contribuição real de cada integrante. Durante a entrevista, cada membro da equipe deverá ser capaz de explicar, com desenvoltura, qualquer parte do trabalho, mesmo que esta tenha sido desenvolvida por outro membro da equipe. Portanto, é possível que ocorra, após a entrevista e/ou exame das atividades registradas, redução da nota geral do trabalho ou ajustes nas notas individuais com vistas a refletir a verdadeira contribuição de cada membro da equipe.

O trabalho em cooperação entre estudantes da turma é estimulado, sendo aceitável a discussão de ideias e estratégias. Contudo, tal interação não deve ser entendida como permissão para utilização de (parte de) código fonte de outras equipes, o que pode caracterizar situação de plágio. Trabalhos copiados em todo ou em parte de outras equipes ou da Internet serão sumariamente rejeitados e receberão nota zero.

Entrega

Todos os códigos fonte referentes à implementação do trabalho deverão ser disponibilizados sem erros de compilação e devidamente testados e documentados através do repositório Git da equipe no Gitlab do IMD-UFRN (http://projetos.imd.ufrn.br/). Além disso, você deverá submeter, até o horário da aula do dia 13 de setembro de 2016, um único arquivo compactado através da opção *Tarefas* na Turma Virtual do SIGAA contendo: (1) os mesmos arquivos de código fonte disponíveis no repositório Git; (2) a documentação do projeto na forma de páginas HTML, geradas automaticamente com a ferramenta Doxygen, e; (3) o relatório escrito, preferencialmente em formato PDF. É importante destacar que serão avaliados única e exclusivamente os arquivos submetidos via SIGAA.

Avaliação

A avaliação deste trabalho será feita principalmente sobre os seguintes critérios: (1) utilização correta dos conteúdos vistos nas aulas presenciais da disciplina; (2) a corretude da execução do programa implementado, que deve apresentar saída em conformidade com a especificação e as entradas de dados fornecidas; (3) a aplicação correta de boas práticas de programação, incluindo legibilidade, organização e documentação de código fonte, e; (4) qualidade do relatório técnico produzido. O trabalho possuirá nota máxima de 5,00 (cinco) pontos, distribuídos de acordo com a seguinte composição:

Item avaliado	**Nota máxima**
Modularização adequada	0,50
Recursividade e iteratividade	
- Implementação de versão recursiva da busca linear	0,10
- Implementação de versão iterativa da busca linear	0,10
- Implementação de versão recursiva da busca binária	0,15
- Implementação de versão iterativa da busca binária	0,15
- Implementação de versão recursiva da busca ternária	0,25
- Implementação de versão iterativa da busca ternária	0,25
Uso adequado de ferramentas de compilação (com *Makefile*), depuração, verificação de memória e *profiling*	0,75
Uso correto de controle de versão com Git	0,50
Uso consistente de alocação dinâmica de memória	0,50
Aplicação adequada de sobrecarga de funções e ponteiros para funções	0,75
Qualidade do relatório escrito	1,00
Total	**5,00**

Por sua vez, o não cumprimento de algum dos critérios anteriormente especificados poderá resultar nos seguintes decréscimos, calculados sobre a nota total obtida até então:

Falta	**Decréscimo**
Programa apresenta erros de compilação, não executa ou apresenta saída incorreta	–70%
Falta de comentários no código fonte e/ou de documentação gerada com Doxygen	–10%
Implementação na linguagem C ou resultante de mistura entre as linguagens C e C++	–30%
Programa compila com mensagens de aviso (*warnings*)	–50%
Plagiarismo	–100%