Trabalho Estrutura de Dados

Flávio Lúcio Corrêa Júnior

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

- **Sumário**:

---

1 Introdução

O trabalho consiste em implementar variações do algoritmo QuickSort e fazer uma análise comparativa de seus respectivos melhores e piores casos. Para isso, analisaremos as seguintes implementações:

- **QuickSort Clássico:** seleção de pivô usando o elemento central.
- **QuickSort Primeiro Elemento:** seleção do pivô como sendo o primeiro elemento do subconjunto.
- **QuickSort Mediana de 3:** seleção do pivô usando a “mediana de três” elementos, em que o pivô é escolhido usando a mediana entre a chave mais à esquerda, a chave mais à direita e a chave central (como no algoritmo clássico).
- **QuickSort Inserção 1%:** o processo de partição é interrompido quando o subvetor tiver menos de k = 1% chaves. A partição então deve ser ordenada usando uma implementação especial do algoritmo de ordenação por inserção, preparada para ordenar um subvetor. Seleção de pivô é feita usando a “mediana de três” elementos, descrita acima.
- **QuickSort Inserção 5%:** mesmo que o anterior, com k = 5%
- **QuickSort Inserção 10%:** mesmo que o anterior, com k = 10%
- **QuickSort Não Recursivo:** implementação que não usa recursividade. Utiliza pilha para simular as chamadas de função recursivas e identificar os intervalos a serem ordenados a cada momento. A seleção do pivô deve ser feita assim como no Quicksort clássico.

2 Implementação:

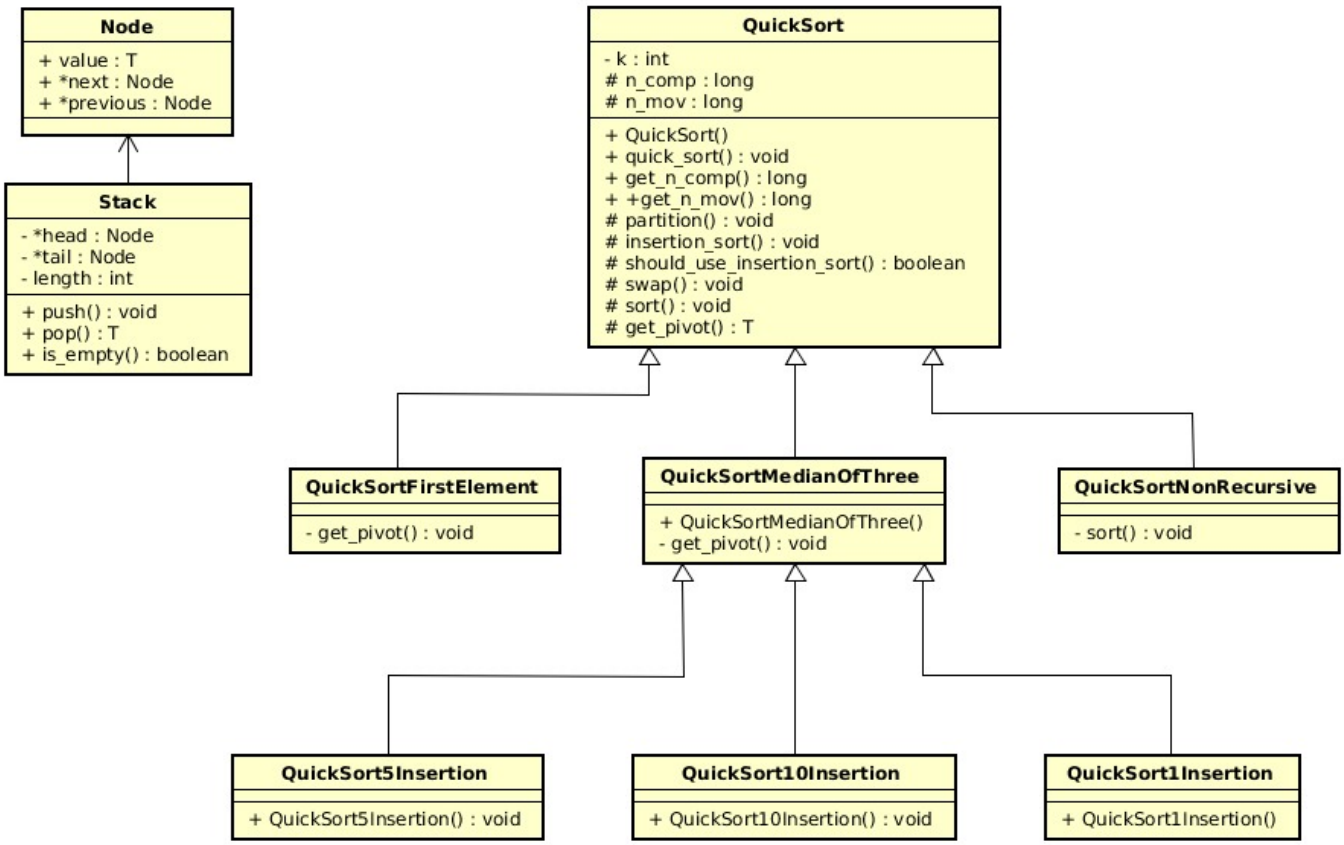
No geral, o programa desenvolvido aceita uma entrada do tipo `./nomedoprograma <variacao> <tipo> <tamanho> [-p]` e produz uma saída `<variacao> <tipo> <tamanho> <n_comp> <n_mov> <tempo>,` onde `variacao`, `tipo` e `tamanho` são os parâmetros recebidos na entrada, `n_comp` e `n_mov` se referem ao número médio de comparações de chaves e de movimentações de registros efetuadas e `tempo` ao tempo mediano de execução, em microssegundos.

2.1 Estruturas de dados:

- **Pilha:**
  - **Uso**: memória auxiliar para execução do Quicksort não recursivo.
  - **Motivo de escolha:** característica FILO (first in last out) que facilita a execução do algoritmo, operações push e pop com complexidade de tempo constantes.
- **Vetor:**
  - **Uso**: armazenar os elementos que serão ordenados pelo algoritmo.
  - **Motivo de escolha:** estrutura a ser ordenada, exigida pela descrição do trabalho

2.2 Classes:

- Diagrama de Classes UML



2.3 Principais Funções:

- `void quick_sort(T *array, int size)`: esta função simplismente inicializa as variáveis acumuladoras dos números de comparações e movimentações com elementos do vetor e faz a primeira chamada da função recursiva `sort()`.
- `virtual void sort(int ini, int end, T *array, int size)`: função recursiva executada para cada partição criada pela chamada do procedimento `partition()`, esta é declarada como um método virtual pois a sub-classe `QuickSortNonRecursive` utiliza sua implementação não recursiva;
- `void partition(int ini, int end, int &i, int &j, T *array)`: principal função do algoritmo, aqui é onde acontece a separação (partição) dos elementos do vetor que são maiores e menores do que um pivot especificamente escolhido. Tal procedimento é executado armazenando 2 indices (um que começa na posição inicial e outro na final do vetor) que são usados para iterar sobre os elementos e efetuar a troca de suas posições quando necessário.

- `void insertion_sort(int ini, int end, T *array)`: esta função é uma implementação do algoritmo Insertion Sort que é usado de maneira híbrida com o Quick Sort nas classes `QuickSortXInsertion` onde X pode assumir o valor de 1, 5 ou 10 como descrito acima.
- `virtual T get_pivot(int ini, int end, T *array)`: função virtual pois podemos ter 3 diferentes métodos de obter tal pivot, cada uma dessas 3 maneiras é implementada nas seguintes classes `QuickSort`, `QuickSortFirstElement`, `QuickSortMedianOfThree`.

2.4 Compilador:

O compilador usado foi o `g++` com a flag `-std=c++14`

3 Análise Experimental:

- Os gráficos, para uma melhor visualização dos dados analisados, podem ser encontrados no arquivo `chart.html`. A escolha por usar um arquivo html foi tomada pensando no fato de que os dados encontrados estão extremamente dispersos o que impedia uma visualização precisa de como estes se comportam. A demostração em html com a bibliotéca `chartJs` permite uma visualização interativa dos dados onde os com maior desvio padrão podem ser omitidos pelo usuário.

3.1 Benchmarks:

- Os testes para coletar as amostras de dados para a análise foram executados em uma máquina com as seguintes configurações de hardware/software:

Parâmetros	Valores
Memória	7,7 GiB
Processador	Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
Placa Gráfica	Intel® UHD Graphics 620 (Kabylake GT2)
Sistema Operacional	Ubuntu 18.04.2 LTS (64-bit)
Disco	KINGSTON SA1000M8240G

3.2 Tempo de Execução:

- **Maior tempo de execução**: Percebe-se que, nesta métrica, quando temos uma distribuição crescente ou decrescente dos elementos do vetor, a implementação com o maior tempo de execução é de quando a escolha do pivot é dada pelo primeiro elemento da partição, ou seja, a encontrada na classe QuickSortFirstElement, já quando a distribuição dos elementos dos vetores se dão de maneira aleatória, a implementação com o maior tempo é a QI10 (QuickSort10Insertion).

- **Menor tempo de execução**: Nesta categoria, quando analisamos os vetores com distribuilção aleatória e decrescente, destacam-se as implementações clássica (QC) e não recursiva (QNR), no entanto quando comparamos os valores na distribuição dos elementos de forma crescente, os algorimos com implementação híbrida entre QuickSort e InsertionSort (QI1, QI5, QI10) vêm à tona, obtendo os melhores tempos de execução.

3.3 Número de Comparações:

- **Maior tempo de execução**: Em ambos os vetóres com distribuições crescente e decrescente o QuickSortFirstElement (QPE) dispara na frente com o maior tempo de execução, isto é dado pois a escolha do pivot é dada pelo primeiro elemento do array, implicando em um número muito grande de comparações para o vetor ser ordenado. No entanto, quando temos elementos aleatóriamente distribuidos, o QPE se comporta de maneira razoavel abrindo espaço para o QuickSort10Insertion, que, quando aplicado em partições com 10% dos elementos do vetor, executa uma implementação do Insertion Sort elevando bastante o número de comparações.

- **Menor tempo de execução**: Aqui é onde obtemos a maior variedade de possíveis resultados. Em vetóres com elementos ordenados aleatóriamente, exceto pelas implementações híbridas entre Quick e Insertion sort, observamos resultados bem similares para as outras implementações, com o QuickSortMedianOfThree (QM3) um pouco à frente dos outros. Já no caso crescente, a situação inverte, pois como o vetor já está ordenado, as implementações com o Insertio Sort, alcançam baixíssimas quantidades de comparações comparada às outras. Por último, com vetóres decrescentes, o QuickSort Clássico e o Não-Recursivo obtêm os menores números de comparações.

3.4 Número de Movimentações:

- **Maior tempo de execução**: Nesta métrica, a situação é um pouco similar a de número de comparações, pois os QuickSort Insertion apresentam os maiores números nas categorias aleatórios e decrescentes, já que o algoritmo de ordenação por inserção demanda um alto número de movimentação, contudo, quando os vetores já estão ordenados (caso crescente), estes abrem espaço para o QM3 uma vez que não precisam de operar movimentações sobre os vetores, enquanto o QM3 terá um alto número de movimentações devido a sua escolha do elemento pivot.

- **Menor tempo de execução**: Aqui as implementações QPE, QC e QNR apresentam, de fato, os menores números de movimentação, exceto pela organização dos vetores com elementos de forma crescente, isso se dá devido ao método de escolha do pivot, que faz com que não existam muitas movimentações para definir os subvetores das partições quando o algoritmo é executado.

4 Conclusão:

- Acredito que o trabalho prático proposto foi de grande utilidade para exercitar a implementação das diversas variações do Quick Sort, bem como as estruturas de dados auxiliares usadas para a execução dos mesmos. Além disso, percebe-se uma parte teórica de grande importância para a conclusão da dinâmica, esta se deu, em sua grande maioria, pela a análise comportamental das implementações dos algoritmos quando expostos à diferentes modelos e tamanhos de dados, permitindo a simulação de um cenário real onde o desenvolvedor é colocado em uma posição em que a melhor decisão deve ser tomada a partir da análise empírica dos possíveis algoritmos disponíveis para a resolução do problema. Aplicando benchmarks e modelando os resultados para uma visualização precisa em que a decisão ótima seja alcançada.

5 Bibliografia:

- Ziviani, N. (2006). *Projetos de Algoritmos com Implementações em Java e C++:*
  - *Capítulo 4: Ordenação.* Editora Cengage.
- http://stackoverflow.com/
- https://www.wikipedia.org/

- https://github.com