Notas de Aula - AED1 – Algoritmos de Ordenação (parte 2)
Prof. Jefferson T. Oliva

Na aula anterior vimos dois algoritmos de ordenação por troca: bubblesort e quick sort

Hoje veremos dois algoritmos de ordenação por seleção: select sort e heap sort.

Ordenação por seleção: seleciona o maior ou o menor elemento do conjunto para cada iteração para colocá-lo em sua devida posição

Select sort

Ideia básica
1 - Selecionar o maior elemento do conjunto
2 - Trocá-lo com o último elemento
3 - Repetir os dois passos anteriores com os n − 1 elementos restantes, após com os n − 2 e assim por diante até sobrar o primeiro elemento que será o menor do conjunto

Implementação

```
void selectsort(int v[], int n){
        int i, j, p, aux;
        for (i = n - 1; i > 1;  i-){
                p = i;
                for (j = 0; j < i; j++)
                        if (v[j] > v[p])
                                p = j;
                aux = v[i];
                v[i] = v[p];
                v[p] = aux;
        }
}
```

Implementação 2:

```
void selectsort(int v[], int n){
        int i, j, p, aux;
        for (i = 0; i  < n - 1;  i--){
                p = i;
                for (j = 0; j < i; j++)
                        p = j;
                if (p != 0){
                        aux = v[i];
                        v[i] = v[p];
                        v[p] = aux;
                }
```

```
        }
}
```

Ver exemplo no slide 6.

Complexidade do método: O(n^2).

Esse método de ordenação é estável.

Simples implementação.

Um dos algoritmos mais rápidos para a ordenação de vetores pequenos.

Em vetores grandes, esse algoritmo é um dos mais lentos.

Ver slide 8

Heap sort

Baseado no princípio de ordenação por seleção em árvore binária.

O método consiste em duas fases distintas:
1 - Montagem da árvore binária (HEAP)
2 - Seleção dos elementos na ordem desejada

ver slide 10

Em uma heap
→ O sucessor à esquerda do elemento de índice i é o elemento de índice 2 * (i + 1) − 1, se 2 * (i + 1) − 1 < n, caso contrário não existe
→ O sucessor à direita do elemento de índice i é o elemento de índice 2 * (i + 1), se 2 * (i + 1) < n, caso contrário não existe

Código para rearranjar um vetor para que o mesmo atenda a condição para ser uma heap

```
static void gerarHeap(int v[], int n){
        int esq = n / 2;

        // Para o vetor ser rearranjado como heap, o processo começa
        // pelos "nós folhas" e a troca pode ocorrer até o "nó raiz"
        while (esq >= 0){
                refazer(v, esq, n - 1);
                esq--;
        }
}

static void refazer(int v[], int esq, int dir){
        int j = (esq + 1) * 2 - 1; // posição de um nó descendente (esquerda)
                                   // do nó localizado na posição esq
        int x = v[esq]; // representa o nó raiz, ou seja, o elemento a partir
                        // do qual será testada a condição de heap

        // Pa partir do nó na posição esq, o arranjo é percorrido até o
        // "nó folha"
        while (j <= dir){
                // É verificado se o "nó filho" esquerdo é menor que o direito
                if ((j < dir) && (v[j] < v[j + 1]))
                        j++;

                // Se o x for maior que o seu descendente, a condição de heap
                // não foi violada
                if (x >= v[j])
                        break;

                // Quando a condição de heap é violada, devem haver trocas
                // de posições entre os elementos
                v[esq] = v[j];
                esq = j; // posição de um dos descendentes do elemento localizado
                        // em esq
                j = (esq + 1) * 2 - 1; // descendente "esquerdo" de v[esq]
        }

        v[esq] = x;
}
```

O fazheap tem o propósito de rearranjar o vetor para que o mesmo atenda a condição para ser uma heap. Esse processo é iniciado na primeira metade do arranjo. Assim, como se fosse em uma árvore binária, a primeira troca poderá entre um nó folha e o seu respectivo pai. Esse processo segue até chegar ao “nó raiz”, que seria o primeiro elemento do vetor. Para a construção da heap, as movimentações são realizadas na função refazheap para impor a condição de heap no arranjo.

No refazheap, os parâmetros são o vetor, a posição de um nó da heap e a posição do último elemento do vetor. No método, primeiramente é definida a posição de um nó descendente (esquerda) do nó localizado na posição esq. Em seguida, na variável x é atribuído o valor do arranjo na posição esq, que será o "nó raiz" (da "árvore" (esq = 0) ou "sub-árvore" (esq > 0)). A partir desse "nó", são explorados os seus respectivos descendentes (no laço while). Dentro do laço while é verificado qual descendente é maior: v[j] ou v[j + 1]. Assim, na variável j será armazenada a posição do descendente com maior valor, o qual será explorado na próxima "rodada" de operações do laço while. Caso a "raiz" explorada seja maior que um descendente (v[j]), não fará sentido a continuação do laço while, pois o maior elemento está na "raiz". Caso contrário, v[esq] receberá v[j], ou seja, o maior valor será posicionado no "nó raiz" explorado. Em seguida, o próximo "nó raiz" passará a ser o elemento na posição j (esq passará a receber j) e j receberá o valor da posição de um dos descendentes no novo "nó raiz". Esse laço while é processado até um nó maior que os seus descendentes imediatos sejam encontrado ou alcançar um "nó folha". Por fim, v[esq] receberá o menor valor processado (x). Esse método parece não ser eficiente, mas ele é usado apenas durante a geração da heap e a aplicação da ordenação. Para a geração da heap, o processamento começa a partir do nível mais baixo da "árvore", ou seja, a partir dos nós folhas e os seus respectivos "nós raizes". O processo é repetido até chegar ao processamento do "nó raiz" (v[0]), no qual, sempre o maior valor estará posicionado.

Função principal

```
void heapsort(int v[], int n){
        int x;
        int dir = n - 1;

        gerarHeap(v, n);

        while (dir > 0){
                x = v[0];
                v[0] = v[dir];
                v[dir] = x;
                dir--;

                refazer(v, 0, dir);
        }
}
```

Na função heapsort, o primeiro passo é a geração da heap. Em seguida, no laço while é o elemento "raiz" do heap (v[0]) é posicionado em sua respectiva posição no arranjo (v[dir]). Após, a variável dir é decrementada para atualizar a posição onde a "raiz" da heap deverá ser colocada na próxima "rodada" do loop. Por fim, a heap é ajustada para manter o arranjo entre 0 e dir em condições de heap. O loop é executado até a ordenação completa da estrutura.
Ver exemplo slide 15.

A primeira vista, o algoritmo aparenta não ser eficiente por causa da remoção do maior elemento da heap, pois o arranjo deve ser manipulado para manter a condição de heap. Mas, o algoritmo é rápido.

Heap sort não é um algoritmo de ordenação estável.

Esse algoritmo não é recomendado para pequenos conjuntos de dados, pois neste caso, um algoritmo simples é mais que suficiente.

Por fim, o custo de tempo é logarítmico: O (n log n)

Ver slide 17.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Rosa, J. L. G. Métodos de Ordenação. SCE-181 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2018.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.