SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 6

Alunos	**NUSP**
Felipe Camargo Cerri	15451119
Gabriel Campanelli Iamato	15452920

Exercício 6 – Merge Sort e Heap Sort

Item a

Comentário

Foram implementadas três formas de ordenação de vetores, o Merge Sort e o Heap Sort (este com duas implementações distintas, mas que usa a mesma ideia) . Os algoritmos testados passaram em todos os testes do RunCodes, e serão apresentados a seguir

Merge Sort

O Merge Sort é um algoritmo de ordenação baseado na ideia de divisão e conquista. O processo do algoritmo baseia-se na ideia de:

1-) Divisão: separar o vetor sucessivamente em dois subarrays (separando o vetor original ao meio), em um processo recursivo, até que o tamanho de cada subarray seja 1 (é o critério de parada para essa recursão) . Esse processo naturalmente é O(log(n))[1].

2-) Intercalação: dados 2 subarrays, mescla-os intercalando os elementos de forma a ordenar sucessivamente o vetor, até que tenham 2 subarrays que , intercalados, resultem no array original. Esse processo é naturalmente O(log(n)) , pois percorre todos os elementos.

[1] Considere todos os “log” citados como base 2.

Portanto, combinando essas duas complexidades que se relacionam, a complexidade do algoritmo , tem relação a tempo, resulta em O(n *log(n)), com complexidade de espaço O(n), pois usa arrays auxiliares para o processo de intercalação.

Heap Sort

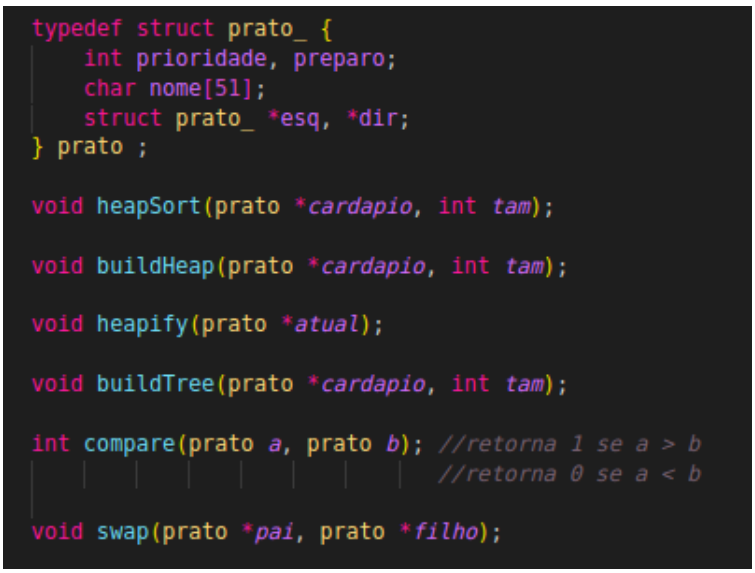
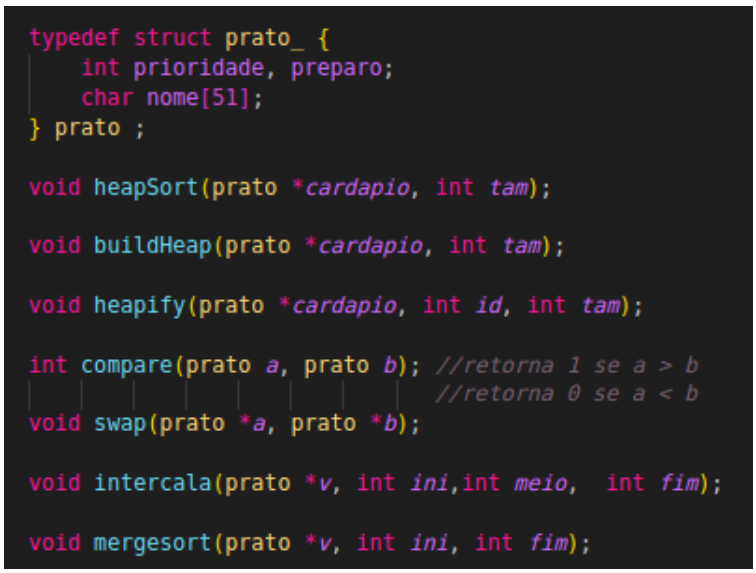
O algoritmo HeapSort é uma melhoria da ordenação por seleção (Selection Sort), utilizando uma estrutura de dados chamada Heap para localizar de forma eficiente o maior ou menor elemento de um vetor. A Heap é uma árvore binária em que cada nó pai tem um valor maior (em uma Max-Heap) ou menor (em uma Min-Heap) que seus filhos, de forma que o maior (ou menor) elemento esteja sempre na raiz da árvore. Além disso, a estrutura possui propriedades que permitem sua implementação diretamente em um vetor (evitando a estrutura de árvore), tornando possível acessar os dois filhos de cada pai de forma indexada e vice e versa. Para implementar o HeapSort, primeiramente transformamos o array em uma Max-Heap pela função “buildHeap”, e logo em seguida o maior elemento é trocado com o último elemento do array para garantir a ordenação crescente. Finalmente, ajustamos a Heap através do “Heapify” e continuamos o processo até a ordenação completa. A construção da Heap é realizada em O(n), enquanto as operações de remoção ou ajuste dos nós para manter a propriedade de Heap ocorrem em O(log n), de forma que a complexidade final para a ordenação fica O(n + nlog n) = O(nlog n).

Heap Sort (Bônus)

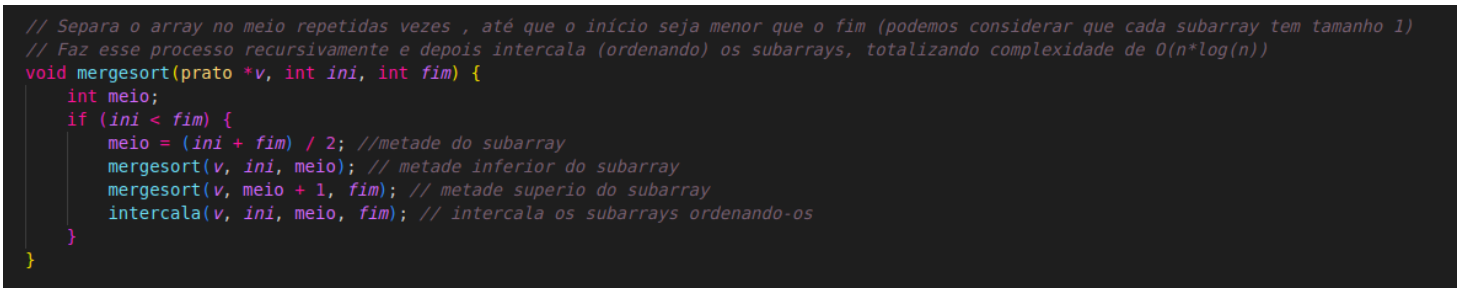
No item bônus, a implementação do Heap Sort foi adaptada para utilizar a Heap no formato de árvore binária com ponteiros, ao invés de um array. Assim, a estrutura foi representada explicitamente como uma árvore, onde cada nó contém ponteiros para seus filhos esquerdo e direito. Em termos da organização dos dados foi necessário alterar a struct principal para que contivesse os devidos ponteiros dos pais para os filhos. Na parte da implementação utilizamos uma função extra (buildTree) responsável por construir a árvore binária completa, enquanto as demais funções foram alteradas para trabalhar diretamente com os ponteiros de cada nó. Além disso, na troca de valores entre os nós, os ponteiros dos filhos também foram atualizados, ademais foi necessário isolar efetivamente cada nó retirado da árvore assegurando a consistência da estrutura.

🧑‍💻 Código

Informações Gerais

Heap Sort Bônus	Merge/Heap Sort Tradicional
	

Merge Sort



```c
void intercala(prato *v, int ini, int meio, int fim) {
    int i, j, k, n1, n2;

    // tamanhos dos dois subarranjos a serem intercalados
    n1 = meio - ini + 1;
    n2 = fim - meio;

    // Aloca-se dinamicamente espaço para os arrays
    prato *L = (prato *) malloc((n1 + 1) * sizeof(prato));
    prato *R = (prato *) malloc((n2 + 1) * sizeof(prato));

    // Caso haja problema na alocação
    if (L == NULL || R == NULL) {
        printf("Erro ao alocar memória!\n");
        exit(1);
    }

    // Inicia-se os dois subarranjos
    for (i = 0; i < n1; i++)
        L[i] = v[ini + i];
    L[n1].prioridade = INT_MAX;
    L[n1].preparo = 0;

    for (j = 0; j < n2; j++)
        R[j] = v[meio + j + 1];
    R[n2].prioridade = INT_MAX;
    R[n2].preparo = 0;

   // Intercala-se os dois arranjos
    i = j = 0;
    for (k = ini; k <= fim; k++) {
        // Verificando prioridades
        if (L[i].prioridade < R[j].prioridade) {
            v[k] = L[i];
            i++;
        }
        // Se prioridades iguais, compara-se tempo decrescentemente
        else if (L[i].prioridade == R[j].prioridade && L[i].preparo > R[j].preparo) {
            v[k] = L[i];
            i++;
        }
        else {
            v[k] = R[j];
            j++;
        }
    }

    // Libera-se memória alocada
    free(L);
    free(R);
}
```

Heap Sort

```c
void heapify(prato *cardapio, int id, int tam) {
    int esq = 2 * id + 1, //filho da esquerda
    dir = 2 * id + 2,  //fihlo da direita
    maior = id;

    if (esq < tam && compare(cardapio[esq], cardapio[id])) { //acha maior filho
        maior = esq;
    }
    if (dir < tam && compare(cardapio[dir], cardapio[maior]) ) {
        maior = dir;
    }
    if (maior == id) return; //caso o pai seja maior que os filhos

    swap(&cardapio[maior], &cardapio[id]); //troca o pai com o maior filho
    heapify(cardapio,maior, tam); //chama o arranjo para o filho que estava desordenado

}

void buildHeap(prato *cardapio, int tam) {
    for (int i = ((tam)/2 - 1) ; i>=0; i--) { //rearrajo a partir do primeiro que nao e fo
        heapify(cardapio, i, tam);
    }
}

void heapSort(prato *cardapio, int tam) {
    buildHeap(cardapio, tam);

    for (int i = tam - 1; i > 0; i--) {
        swap(&cardapio[0], &cardapio[i]); //maior vai para o final do array
        heapify(cardapio, 0, i);//estabelece max heap com raiz em idx 0
    }
}

int compare(prato a, prato b) {        //1 se a > b
    if (a.prioridade == b.prioridade) {//0 se a < b
        if (a.preparo < b.preparo)
            return 1;
        else
            return 0;
    }

    if (a.prioridade > b.prioridade)
        return 1;
    else
        return 0;
}

void swap(prato *a, prato *b) {
    prato temp = *a;
    *a = *b;
    *b = temp;
}
```

Heap Sort (Bônus)

```c
void heapify(prato *atual) {
    prato *maior = atual;
    int trocas = 0;

    if (atual->esq != NULL && compare(*(atual->esq), *atual)) { //acha maior filho
        maior = atual->esq;
        trocas++;
    }
    if (atual->dir != NULL && compare(*(atual->dir), *maior) ) {
        maior = atual->dir;
        trocas++;
    }
    if (trocas == 0) return; //caso o pai seja maior que os filhos

    swap(maior, atual); //troca o pai com o maior filho
    heapify(maior); //chama o arranjo para o filho que estava desordenado

}

void buildHeap(prato *cardapio, int tam) {
    for (int i = ((tam)/2 - 1) ; i>=0; i--) { //rearrajo a partir do primeiro que nao e fo
        heapify(&cardapio[i]);
    }
}

void heapSort(prato *cardapio, int tam) {
    buildTree(cardapio, tam);
    buildHeap(cardapio, tam);

    for (int i = tam - 1; i > 0; i--) {
        swap(&cardapio[0], &cardapio[i]); //maior vai para o final do array

        //isola o maior elemento da arvore
        if (i%2) { //filho é da forma 2k + 1 (impar), portanto é o filho da esquerda
            cardapio[(i-1)/2].esq = NULL; //acessa o caminho do pai pro filho isolado
        }
        else{
            cardapio[(i-1)/2].dir = NULL;
        }

        cardapio[i].dir = NULL;
        cardapio[i].esq = NULL;

        heapify(&cardapio[0]);//estabelece max heap com raiz em idx 0
    }
}
```

```c
void buildTree(prato *cardapio, int tam) {
    int esq, dir;
    for (int i =0; i<(tam/2); i++) { //constroi a arvore
        esq = 2*i +1;
        dir = 2*i +2;

        if (esq < tam)
            cardapio[i].esq = &cardapio[esq];
        if (dir < tam)
            cardapio[i].dir = &cardapio[dir];
    }

    for (int i = (tam/2); i<tam; i++) { //folhas apontam para nulo
        cardapio[i].esq = NULL;
        cardapio[i].dir = NULL;
    }
}

int compare(prato a, prato b) {         //1 se a > b
    if (a.prioridade == b.prioridade) {//0 se a < b
        if (a.preparo < b.preparo)
            return 1;
        else
            return 0;
    }

    if (a.prioridade > b.prioridade)
        return 1;
    else
        return 0;
}

void swap(prato *pai, prato *filho) {
    prato temp = *pai; //troca do pai com o filho
    *pai = *filho;
    *filho = temp;

    prato *filho_esq = pai->esq, //troca dos filhos
          *filho_dir = pai->dir;

    pai->esq = filho->esq;
    pai->dir = filho->dir;

    filho->esq = filho_esq;
    filho->dir = filho_dir;
}
```

□ Saída

Para comprovar a eficiência dos dois métodos de ordenação, foi utilizada a biblioteca <time.h> da linguagem C para medir o tempo de execução dos algoritmos. Como os resultados foram testados no RunCodes, serão apresentados apenas os tempos de execução de cada algoritmo, com casos de teste do RunCodes:

Tempos de Execução

	497 casos	**1193 casos**	**38286 casos**
Merge Sort	0.000182	0.000462	0.019712
Heap Sort Tradicional	0.000311	0.000575	0.029486
Heap Sort Bônus	0.000293	0.000837	0.030410

Algoritmos mais rápidos

497 casos	**1193 casos**	**38286 casos**
Merge Sort	Merge Sort	Merge Sort

A partir dos testes, percebe-se que Merge Sort é um algoritmo mais eficiente do que o Heap Sort (nas duas implementações testadas) no que tange tempo de execução, e apresentou eficiência mais rápida nos casos testados. A diferença de processamento ficou mais clara ao passo que o tamanho dos casos teste aumentava. No caso das comparações do Heap Sort, ambos apresentaram tempos de execução muito próximos, não tendo diferença *significativa* no tempo de execução. Vale ressaltar que apesar da ligeira superioridade em eficiência temporal do Merge Sort em relação ao Heap Sort, o algoritmo baseado na divisão e conquista apresenta como desvantagem o uso de espaço extra O(n), dado que o Merge Sort requer armazenamento adicional para realizar as operações de mesclagem, ao contrário do Heap Sort que opera diretamente sobre o array.