Heap Sort

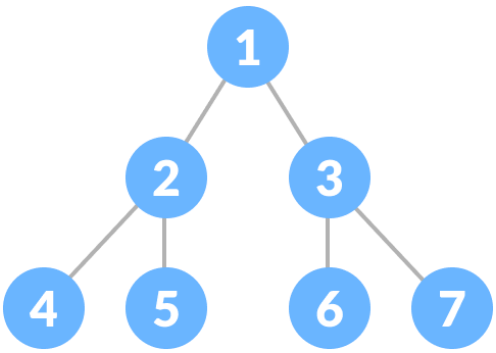
Para entender como o Heap Sort primeiro tem que se fazer um estudo de uma Árvore Binária Completa.

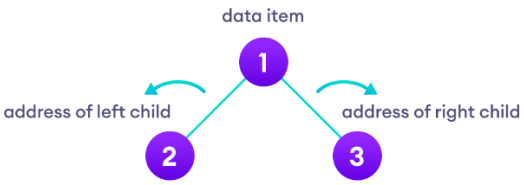
Árvore Binária ou Binary Tree

Árvore Binária é uma forma de organizar os dados com base uma estrutura de dados que tem formato de uma árvore geneológica que consiste em ter 3 elementos, sendo:

1. Data – A informação na qual cada nó da árvore deverá armazenar;
2. Left – Um ponteiro que irá apontar para uma sub-árvore ou folha do lado esquerdo;
3. Right – Um ponteiro que irá apontar para uma sub-árvore ou folha do lado direito;

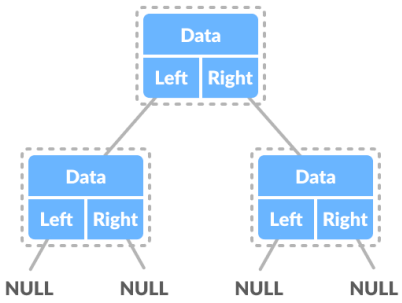
Representação Gráfica de uma Árvore Binária





Sub-Árvore ou subTree

Uma árvore que tem como parente um outro nó que não seja o nó raiz, e podem apresentar o formato de um triangulo equilateral.



Folha

Último elemento da árvore com os ponteiros nulos.

Divide-and-Conquer Algorithm

Esta estrutura usa um algoritmo chamado “dividir para conquistar” na qual consiste em dividir recursivamente o problema original em sub-problemas do mesmo tipo até se tornar simples o suficiente para ser solucionado diretamente, de seguida juntar todas as soluções em uma só.

A complexidade deste algoritímo é O(nlog n), sendo esta a melhor, numa média de melhores ou piores casos. Esta complexidade pode ser facilmente compreendida apartir da seguinte função:

T(n) =2T(n/2)+n.

Saber Mais sobre…

Implementação de uma árvore binária

BinaryTree conta com as seguintes funções principais:

1. Init( ) – Nesta função a árvore binária seria iniciada vazia , então deve retornar um ponteiro do tipo BinaryTree.
2. AddNode(struct node*) – Nesta função serão adicionados os novos nós com base o algoritmo “Divide-and-Conquer”, organizando os menores valores no lado esquerdo da raiz e os maiores a direita.
3. Transversals functions – Estas funções são maneiras de poder apresentar uma árvore binária.
    a. PreOrderTransversal(struct node*) – Apresenta primeiro a raiz de seguida todos os elementos do filho à esquerda de seguida os da direita.
    b. InOrderTransversal(struct node*) – Apresenta primeiro todos os elementos à esquerda da raiz, de seguida a própria raiz e depois os elementos a sua direita.
    c. PostOrderTransversal(struct node*) – Apresenta primeiro os elementos da esquerda e direita da raiz, e por último a própria raiz.

Nota : Apenas irei fazer a implementação das funções que precisamos para poder entender o Heap Sort.

No Código

Irá usar-se a **linguagem de programação C**, contudo, todo conteúdo abordado neste documento pode ser implementado em qualquer outra linguagem de programação.
Sendo assim, Bora para o código!

BiTree.h

```h
#ifndef BITREE_H
#define BITREE_H
// BinaryTree implementation

struct node
{
    // Tipo de dados
    /*
        Neste caso irá se usar o tipo inteiro
        Mas pode se utilizar qualquer tipo
        ORIGINAL : void *data;
    */

    int data;

    // Ramificações
    struct node *left, *right;
};

struct BinaryTree
{
    // A raiz da árvore
    struct node *root;
};

struct node *newNode(int);

struct BinaryTree *Init();

struct node *AddNodeRecursive(struct node *, int);

void AddNode(struct BinaryTree *, int);

void PreOrderTransversal(struct node *);
void InOrderTransversal(struct node *);
void PostOrderTransversal(struct node *);
#endif
// Implemented by Ivandro Neto
```

BiTree.c

```c
#include "BiTree.h"
#include <stdlib.h>
#include <stdio.h>

struct node *newNode(int data)
{
    struct node *newData = (struct node *)malloc(sizeof(struct node));
    newData->data = data;
    newData->left = newData->right = NULL;
    return newData;
}

struct BinaryTree *Init()
{
    struct BinaryTree *Bitree = (struct BinaryTree *)malloc(sizeof(struct
BinaryTree));
    Bitree->root = NULL;
    return Bitree;
}

struct node *AddNodeRecursive(struct node *root, int data)
{
    struct node *newData = newNode(data);
    if (root == NULL)
        root = newData;
    else
    {
        if (data > root->data)
            root->right = AddNodeRecursive(root->right, data);
        else
            root->left = AddNodeRecursive(root->left, data);
    }

    return root;
}

void AddNode(struct BinaryTree* tree, int data)
{
   tree->root = AddNodeRecursive(tree->root, data);
}
```

Main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "BiTree.c"

int main(int argc, char const *argv[])
{
    struct BinaryTree* tree = Init();

    AddNode(tree, 50);
    AddNode(tree, 25);
    AddNode(tree, 75);
    AddNode(tree, 80);
    AddNode(tree, 70);
    AddNode(tree, 100);
    AddNode(tree, 200);
    AddNode(tree, 12);
    AddNode(tree, 37);
    AddNode(tree, 43);
    AddNode(tree, 30);

    PreOrderTransversal(tree->root);
    printf("\n\n");
    InOrderTransversal(tree->root);
    printf("\n\n");
    PostOrderTransversal(tree->root);
    printf("\n\n");

/*
      OUTPUT::
      50 -> 25 -> 12 -> 37 -> 30 -> 43 -> 75 -> 70 -> 80 -> 100 -> 200 ->
      25 -> 12 -> 37 -> 30 -> 43 -> 50 -> 75 -> 70 -> 80 -> 100 -> 200 ->
      25 -> 12 -> 37 -> 30 -> 43 -> 75 -> 70 -> 80 -> 100 -> 200 -> 50 ->
    */

    return 0;
}
```

Heapify implementation

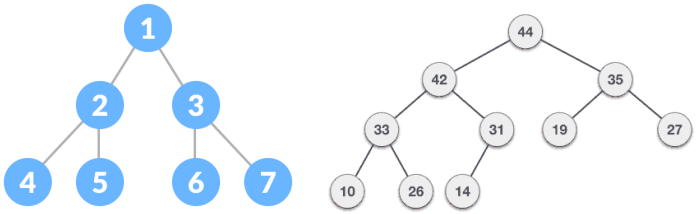
Já com alguma ideia sobre o funcionamento e organização de uma árvore binária, estamos prontos implementar a nossa função de troca.
Heapify é uma função recursiva que consiste em comparar e trocar os elementos presentes no array organizando-os em formato de uma árvore binária.

Code

```
void heapify(int arr[], int size, int current)
{
    // primeiramente, localiza-se os elementos da esquerda e direita
    int largest = current; // Inicialmente o maior valor será sempre o elemento raiz
    int left = 2 * current + 1; // fórmula para achar o elemento esquerdo da árvore
    int right = 2 * current + 2; // fórmula para achar o elemento direito da árvore

// Verifica-se se o indíce do elemento esquerdo está dentro do array e se o seu valor é maior que o maior valor
    if(left < size && arr[left] > arr[largest])
        largest = left;

// Verifica-se se o indíce do elemento direito está dentro do array e se o seu valor é maior que o maior valor
    if(right < size && arr[right] > arr[largest])
        largest = right;

// Verifica-se se o elemento selecionado se é o maior também
    if(current != largest)
    {
        // Caso não for os troque
        swap(&arr[largest], &arr[current]);
        // Depois pega o maior como o selecionado da próxima verificação
        heapify(arr, size, largest);
    }
}
```

Max-Heap and Min-Heap

Max-Heap é a forma na qual os elementos do array serão organizados, colocando os menores valores no final e os maiores no ínicio, enquanto o min-heap é o seu inverso.



Neste código à cima contém a implementação do max-heap, para organizar o array com o min-heap apenas precisamos alterar algumas variáveis.

Code

```c
void heapify(int arr[], int size, int current) {
    int smallest = current;
    int left = 2 * current + 1;
    int right = 2 * current + 2;

    // Encontra o menor elemento entre o nó atual e seus filhos esquerdo e direito
    if (left < size && arr[left] < arr[smallest])
        smallest = left;

    if (right < size && arr[right] < arr[smallest])
        smallest = right;

    // Se o nó atual não for o menor, troca ele com o menor e chama heapify recursivamente para o subárvore afetada
    if (smallest != current) {
        swap(&arr[current], &arr[smallest]);

        heapify(arr, size, smallest);
    }
}
```

Agora com a função feita, podemos montar as nossa função que fará está mudança em toda árvore.

Max-Heap

```
void MaxHeap(int arr[], int size)
{
// Começando pelo último nó não-folha, chama heapify para cada nó em ordem reversa de nível para garantir um max heap válido
    for(int i = size/2 -1; i>=0; i--) heapify(arr, size, i);
}
```

Min-Heap

```
void MinHeap(int arr[], int size)
{
    // Começando pelo último nó não-folha, chama heapify para cada nó em ordem reversa de nível para garantir um min heap válido
    for(int i = size/2 -1; i>=0; i--) heapify(arr, size, i);
}
```

Heap Sort

Tendo todos os elementos prontos, já só falta a cereja no topo do bolo!

Na função HeapSort primeiro precisamos organizar os elementos de maior para menor (MaxHeap) ou de menor para maior(MinHeap), de seguida trocar o último elemento do array com o primeiro elemento e eliminar o último elemento da nossa árvore, e proceder desta forma até restar apenas um elemento na nossa árvore.

How it works

1 12 9 5 6 10 20

1 12 20 5 6 10 9

20 12 1 5 6 10 9

20 12 10 5 6 1 9

12 9 10 5 6 1

10 9 1 5 6

9 6 1 5

6 5 1

5 1

1 5 6 9 10 12 20

Code

```
void HeapSort(int arr[], int size)
{
    // Gera um max heap apartir do array dado.
    MaxHeap(arr, size);

    // Heap Sort
    for(int i = size-1; i>=0; i--)
    {
        //Troca a posição do primeiro elemento com o último do array
        swap(&arr[0], &arr[i]);
        //Elimina o último elemento e volta a reorganiza-los até sobrar um único
elemento
        heapify(arr, i, 0);
    }

}
```

Para a implementação com min-heap é só chamar a função MinHeap ao invés do MaxHeap.

Exemplo

```
int main(int argc, char const *argv[])
{
    int arr[] = {1, 12, 9, 5, 6, 10, 20};
    int size = sizeof(arr) / sizeof(arr[0]); // Obtendo o tamanho do array
    HeapSort(arr, size);
    printArray(arr, size);
    // OUTPUT::1 5 6 9 10 12 20
    return 0;
}
```

Todo código

```c
#include <stdio.h>
#include <stdlib.h>
#include "BiTree.c"

void swap(int *x, int *y)
{
    int aux = *x;
    *x = *y;
    *y = aux;
}
void min_heapify(int arr[], int size, int current)
{
    int smallest = current;
    int left = 2 * current + 1;
    int right = 2 * current + 2;

    // Encontra o menor elemento entre o nó atual e seus filhos esquerdo e direito
    if (left < size && arr[left] < arr[smallest])
        smallest = left;

    if (right < size && arr[right] < arr[smallest])
        smallest = right;

    // Se o nó atual não for o menor, troca ele com o menor e chama heapify recursivamente para o subárvore afetada
    if (smallest != current)
    {
        swap(&arr[current], &arr[smallest]);

        min_heapify(arr, size, smallest);
    }
}

void heapify(int arr[], int size, int current)
{
    // primeiramente, localiza-se os elementos da esquerda e direita
    int largest = current;       // Inicialmente o maior valor será sempre o elemento raiz
    int left = 2 * current + 1;  // fórmula para achar o elemento esquerdo da árvore
    int right = 2 * current + 2; // fórmula para achar o elemento direito da árvore

        // Troca a posição do primeiro elemento com o último do array
```

```c
    // Verifica-se se o indíce do elemento esquerdo está dentro do array e se o seu valor é maior que o maior valor
    if (left < size && arr[left] > arr[largest])
        largest = left;

    // Verifica-se se o indíce do elemento direito está dentro do array e se o seu valor é maior que o maior valor
    if (right < size && arr[right] > arr[largest])
        largest = right;

    // Verifica-se se o elemento selecionado se é o maior também
    if (current != largest)
    {
        // Caso não for os troque
        swap(&arr[largest], &arr[current]);
        // Depois pega o maior como o selecionado da próxima verificação
        heapify(arr, size, largest);
    }
}
void MaxHeap(int arr[], int size)
{
    // Começando pelo último nó não-folha, chama heapify para cada nó em ordem reversa de nível para garantir um max heap válido
    for (int i = size / 2 - 1; i >= 0; i--)
        heapify(arr, size, i);
}
void MinHeap(int arr[], int size)
{
    // Começando pelo último nó não-folha, chama heapify para cada nó em ordem reversa de nível para garantir um min heap válido
    for (int i = size / 2 - 1; i >= 0; i--)
        min_heapify(arr, size, i);
}
void HeapSort(int arr[], int size)
{
    // Gera um max heap apartir do array dado.
    MaxHeap(arr, size);
    /*
        Para min heap comenta a linha à cima e descomenta a linha abaixo
    */
    // MinHeap(arr, size);

    // Heap Sort
    for (int i = size - 1; i >= 0; i--)
    {
```

```
        swap(&arr[0], &arr[i]);
        // Elimina o último elemento e volta a reorganiza-los até sobrar um único
elemento
        heapify(arr, i, 0);
        /*
            Para min heap comenta a linha à cima e descomenta a linha abaixo
        */
        // min_heapify(arr, i, 0);
    }
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
}
int main(int argc, char const *argv[])
{
    int arr[] = {1, 12, 9, 5, 6, 10, 20};
    int size = sizeof(arr) / sizeof(arr[0]); // Obtendo o tamanho do array
    HeapSort(arr, size);
    printArray(arr, size);
    // OUTPUT::1 5 6 9 10 12 20
    return 0;
}

//Coded by Ivandro Neto
```

Utilidade

O Heap sort pode ser útil em situações onde:

1. Você precisa de um algoritmo de ordenação com uma complexidade de tempo de pior caso garantida de O(n log n): O Heap sort tem uma complexidade de tempo de pior caso de O(n log n), que é a mesma de outros algoritmos de ordenação baseados em comparações, como o merge sort e o quicksort. Isso faz dele uma boa escolha quando você precisa de um algoritmo de ordenação com características de desempenho previsíveis.
2. Você tem memória limitada disponível: O Heap sort tem uma complexidade de espaço de O(1) para a implementação em tempo real, o que significa que ele usa uma quantidade fixa de memória independentemente do tamanho da matriz de entrada. Isso pode ser uma vantagem em situações onde a memória é limitada, como em sistemas embarcados ou em aplicações de tempo real.
3. Você tem dados que não estão previamente classificados ou parcialmente classificados: O Heap sort não depende da ordem inicial dos dados de entrada, portanto, pode lidar eficientemente com matrizes aleatórias e quase classificadas.
4. Você não precisa de um algoritmo de ordenação estável: O Heap sort não é um algoritmo de ordenação estável, o que significa que ele não preserva a ordem relativa dos elementos iguais na matriz de entrada. Se você não precisa de um algoritmo de ordenação estável, o Heap sort pode ser uma boa escolha devido à sua eficiência e desempenho previsível.

Conclusão

No geral, o Heap sort pode ser uma boa escolha em uma ampla variedade de situações em que você precisa de um algoritmo de ordenação com uma complexidade de tempo de pior caso previsível e uso limitado de memória, e não exige um algoritmo de ordenação estável.

Nota: Todo conteúdo neste documento foi gerado de conclusões tomadas por mim, sendo por este motivo poderá haver alguma coisa mal interpretada da minha parte, caso encontrar algum ponto relevante que não cheguei de abordar ou algum erro, por favor entre em contacto comigo.

Referências

www.geeksforgeeks.org/heap-sort/?ref=lbp

www.simplilearn.com/tutorials/c-tutorial/heap-sort-in-c-program

www.geeksforgeeks.org/introduction-to-binary-tree-data-structure-and-algorithm-tutorials/

www.youtube.com/watch?v=eiMMtyRBYCE&list=TLPQMTYwMzIwMjOfZbO0se1dbg&index=2

www.programiz.com/dsa/binary-tree

www.programiz.com/dsa/heap-sort