Estruturas de Dados e Algoritmos Fundamentais

(ano letivo 2023-24)

UNIDADE CURRICULAR: Estruturas de dados e algoritmos fundamentais

CÓDIGO: 21046

DOCENTE: Paulo Shirley

A preencher pelo estudante

NOME: Bruno Ricardo de Sá Ferreira

N.º DE ESTUDANTE: 2201529

CURSO: Engenharia Informática

DATA DE ENTREGA: 9 de Maio de 2024

Relatório

Conforme solicitado, foi desenvolvido um programa em c++ padrão que aceita comandos para a gestão de uma árvore binária heapmax.

Dentro de uma perspectiva de auto avaliação acredito ter atendido aos critérios propostos pelo professor.

O heap é representado como uma instância da classe IMAXH, que possui um vetor de nós alocado dinamicamente, um contador que indica quantos nós o heap tem e um valor máximo que define a capacidade do heap.

O comando insert insere itens no heap na ordem apresentada, onde na chamada do mesmo não é utilizado o algoritmo heapify.

Os métodos são utilizados de forma simples e devidamente comentado para a compreensão de cada algoritmo.

Imprime os alertas de “heap vazio” ou “heap cheio”.

Lê os ficheiros de entrada de forma eficiente , não são utilizadas variáveis globais nem STL e passa a todos os testes propostos pelo professor.

Funcionamento do programa.

Trabalhamos com 3 ficheiros, maxh.h , maxh.cpp , main-maxh.cpp.

Maxh.h:

Aqui definimos a estrutura do nosso heap, representados pela classe IMAXH, com um vetor de inteiros, contador de elementos no heap, e a sua capacidade máxima.

São definidos também os construtor e destrutor da classe, que inicializam o nosso objeto IMAXH com capacidade máxima de 15 e mais tarde o destrutor, liberta essa memória alocada.

Definimos getters e setters, como boa pratica de programação para acessar atributos privados da classes de forma simples.

E por fim temos os métodos do heap:

```
insert(int item);//insere item no heap
void print_max();//imprime maior item no heap
void print();//imprime todos os elementos no heap
void dim();//imprimir quantos elementos estao presentes no heap
void dim_max();//imprime a capacidade que o heap tem
void clear();//limpa o heap
void deleteMax();//apaga o maior item do heap
void heapify(int i);//metodo para ajustar o heap atraves do indice definido i
void buildMaxHeap();// metodo que constroi um heap maximo atravez do vetor atual
```

```
void redim_max(int newNv);// redefinir a capacidade do heap
void heapify_up(int i);//ajuste do heap movento ele para cima
```

Estes são os métodos que realizam as ações no heap.

maxh.cpp

Aqui ficheiro implementamos os algoritmos. Mas como eles funcionam ?

Sabendo os métodos que temos, vamos ver como podemos usa-los.

Vamos supor que temos o heap vazio, e que queremos inserir um número.

O comando “insert” vai ser chamado e vamos introduzir esse item na posição [0] do nosso heap.

Vamos lembrar que o heap já tem uma capacidade máxima definida. Após este passo, temos dentro do método outro método chamado heapify_up, apenas responsável por manter a ordem no heap. Como o primeiro elemento é 0, ele retorna, e não faz nada.

Então vamos inserir o segundo elemento no heap que vai para a posição [1], agora já podemos usar o método heapify_up. Ele calcula a posição do nó pai do atual nó, e compara tamanhos.
Exemplo, temos o número 1 no posição [0] e 2 na posição [1], ele vai fazer essa comparação e se dois, que é o filho por maior que o pai, então fazemos essa troca, para que seja respeitada a ordem do maxheap.

Este algoritmo é recursivo, pois tem de ser chamado varias vezes por causa da troca. Pois o ultimo elemento a ser inserido, que vai para o final da arvore, pode ser maior que o segundo filho até ao momento, então, há uma troca, em que o atual inserido, passa para a posição correta, mas o segundo filho da raiz da árvore vai para o fim, então com a chamada recursiva ele vai subindo.

Este algoritmo é utilizado especialmente aqui, neste caso de inserção apenas.

Método print_max, verifica se o heap está vazio, e caso contrario, ele vai acessar a posição [0] do vetor da nossa árvore, que por ser maxheap, vai ser o maior elemento.

Método print, dos bons algoritmos deste exercício e é usado para imprimir a nossa árvore numa ordem lógica.

Primeiro verificamos se o heap está vazio e em seguida, declaramos variáveis para controlar, o número de elementos no nível atual, e outra para rastrear quantos elementos já foram impressos.

Fazemos um loop que vai iterar pelo heap, e imprimimos o primeiro elemento do heap, após isto, fazemos uma condição que verifica se o número de elementos no heap, corresponde ao número de elementos imprimidos.

Começamos com uma variável levelsize = 1, é sempre 1 porque estamos no topo da árvore, e verificamos se não estamos no ultimo elemento do heap.

Então vamos supor que queremos imprimir, 1, 2, 3.

Neste momento já imprimimos o 1. Levelsize, igualou levelcount.
Sendo levelcount, uma variável que declaramos no inicio do método, para rastrear a contagem que é incrementada,

```
        if (levelCount != levelSize - 1 && i != n - 1) {
            std::cout << ' ';
        }
        levelCount++; //
```

Passamos para a próxima condição:

if (levelCount == levelSize || i == n - 1) neste momento, 1 = 1 , e não estamos no final da nossa arvore 1,2,3. Então vamos identar.
std::cout << std::endl;

Após isto, fazemos este calculo que significa regra para as arvores binárias levelSize *= 2, isto é 1*2 = 2. Nivel dois tem dois elementos.

O próximo nível, tem sempre o dobro dos elementos do nível atual.

Atualizamos levelCount = 0, e o loop continua a rodar até fazer o seu trabalho.
Imprimindo 2 e depois 3, que estão no nível dois.

IMAXH::deleteMax(), removemos o maior elemento do heap, trocando a primeira posição com a ultima e decrementando a ultima posição. Após isto precisamos chamar o algoritmo heapify.

Este método vai receber o novo índice zero na nossa árvore, e calcular os índices para os nós abaixo da esquerda e direita. Vê qual deles é maior e troca com o nó pai. Esta função também é recurso, e faz o mesmo processo para baixo, até corrigir a árvore.

```
 // largest = right;
//  largest = left; // aqui troca o índice.
if (largest != i) {
        std::swap(v[i], v[largest]);// troca valores
        heapify(largest);
```

Até aqui vimos, heapify, heapify_up para manter a estrutura quando um algoritmo é inserido no fim, e agora temos, buildMaxHeap(), este converte o vetor definido pelos itens "item ..." num max Heap. Todo o conteúdo anterior do heap é descartado/perdido. O algoritmo Bottom-up (de baixo para cima) descrito no livro recomendado é aplicado, quando o comando heapify_up item … é acionado.
Este começa no ultimo nó da arvore, e chama a função heapify, até chegar ao topo.

Por fim temos void IMAXH::redim_max(int newNv), que redefine a dimensão do heap, libera a memória, aloca nova memória, atualiza a capacidade máxima, limpa o heap. Após a execução deste método o heap estará vazio e com nova capacidade.

Ficheiro **main-maxh.cpp**
É aqui onde chamamos todos os comandos, lendo a entrada linha por linha, encontrando os comandos e processando-os. Em suma, é aqui onde manipulamos a estrutura do programa