E-fólio B | Folha de resolução para E-fólio

NOME | Luciano Eusébio Marques Marafona

N.º DE ESTUDANTE | 2003730

CURSO | Licenciatura Engenharia Informática

TURMA | 02

DATA DE ENTREGA | 29-05-2023

1.1.1 Inserir os itens 5 2 8 4 pela ordem indicada

Começamos com um heap vazio e inserimos os elementos um a um

Inserir o item 5

O 5 será inserido na última posição e por ser o único elemento, é a raiz do heap.

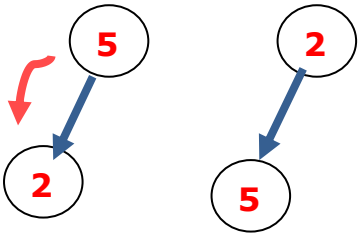
Vetor	Diagrama da árvore:
5	

Inserir o item 2

No caso desta inserção, o elemento 2 é menor que o seu pai, onde tem que trocar para restaurar a propriedade do heap e 2 passa a ser a raiz.

5	2

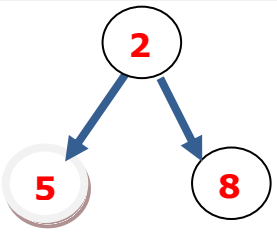
2	5



Inserir o item 8

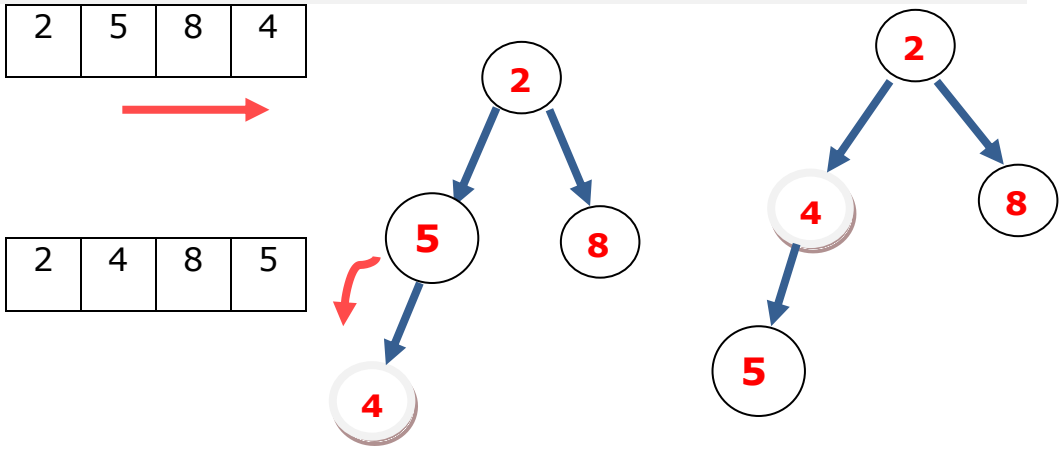
O 8 simplesmente passa para a folha direita, uma vez que a esquerda já está ocupada com o 5.

2	5	8



Inserir o item 4

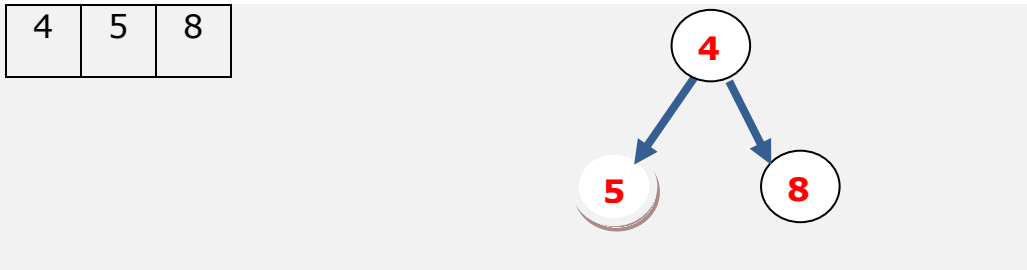
No caso desta inserção, o elemento 4 passaria para a folha esquerda mas como é menor que o seu pai 5, troca com o seu pai.



1.1.2 Remover o menor item

Remover o menor item (2)

Ao remover o menor item do heap (da raiz) e para manter a propriedade do heap, o último elemento do vetor é movido para a raiz e, em seguida, é realizada uma operação chamada "down-heapify", onde o elemento é trocado com o menor de seus filhos, se necessário, até que a propriedade do heap seja restaurada.



1.1.3 Remover o menor item

Remover o menor item (4)

Após a remoção do menor item (4), novamente é feita a operação de "down-heapify" para restaurar a propriedade do heap.

1.2.1 Para um nó com índice i, os índices dos nós filho esquerdo e direito

Se o índice do nó atual é i, podemos calcular os índices dos nós filho esquerdo e direito usando a seguinte aritmética de inteiros:

O índice do nó filho esquerdo: 2*i + 1

O índice do nó filho direito: 2*i + 2

Por exemplo, para um nó com índice i = 3, os índices dos nós filho esquerdo e direito seriam:

Nó filho esquerdo: 2*3 + 1 = 7

Nó filho direito: 2*3 + 2 = 8

1.2.2 Para um nó com índice j, o índice do nó pai

Se o índice do nó atual é j, podemos calcular o índice do nó pai usando a seguinte aritmética de inteiros:

O índice do nó pai: (j - 1) / 2

Por exemplo, para um nó com índice j = 6, o índice do nó pai seria:

Nó pai: (6 - 1) / 2 = 2

1.2.3 Para um Heap com um total de n nós, o índice do último nó não folha

O último nó não folha em um heap completo está localizado no nível mais baixo da árvore binária completa, antes dos nós folhas, onde para o calcular usa-se a seguinte aritmética de inteiros:

Índice do último nó não folha: (n/2) - 1

Por exemplo, se um heap possui um total de n = 10 nós, o índice do último nó não folha seria:

Último nó não folha: (10/2) - 1 = 4

se um heap possui um total de n = 15 nós, portanto ímpar, o

índice último do nó não folha = (15 / 2) - 1 = 7.5 - 1 = 6.5

Entretanto, como os índices dos nós são inteiros e começam do 0, o índice do último nó não folha é o número inteiro mais próximo menor ou igual a 6.5, o que é igual a 6.

1.3 Relatório do programa disponibilizado no recurso VPL

O programa que está em baixo no final do relatório, onde obti 100%

de percentagem de sucesso nos casos de teste do recurso VPL

disponibilizado na página da unidade curricular, consiste em uma estrutura de dados chamada "min-heap" (heap mínimo) usando uma class em C++.

minh.h:

Aqui defini as variáveis os seus setters e getters da class IMINH: v vetor para os nós, n como contador dos nós e nv para a capacidade do heap.

É também definido um destructor e um constructor com a variável nmax como parâmetro que definirá a capacidade do heap, assim como os protótipos de todos os métodos usados no programa.

minh.cpp:

Concisamente, irei descrever o que cada método faz:

IMINH(int nmax): Agora o construtor da classe IMINH, inicializa a estrutura de dados, onde aloca memória para um vetor de inteiros com tamanho máximo definido por nmax, inicializa o número de nós usados como 0 e passa a capacidade máxima do vetor como nmax que inicializa nv.

~IMINH(): Destrutor da classe IMINH que liberta a memória alocada, removendo o vetor de inteiros com delete.

insert(int item): Vai inserindo item a item no heap começando por o final, onde vai corrigindo o heap, subindo o elemento inserido até a posição correta, para ser mais específico, explico passo a passo:

v[n] = item; Insere o elemento item no final do vetor v, onde n conta o número de nós usados atualmente no heap.

int i = n; Inicializa a variável i com o valor de n, que é o índice do elemento recém-inserido.

int parent = (i - 1) / 2; Calcula o índice do pai do nó atual que permite encontrar a posição correta para o novo elemento no heap.

while (i > 0 && v[i] < v[parent]) executa um loop que enquanto i for maior que 0 (ainda não chegou à raiz do heap) e o valor do nó atual (v[i]) for menor que o valor do pai (v[parent]).

Se o caso, realiza a troca entre o nó atual e seu pai:

swap(v[i], v[parent]), onde esta troca garante me que o menor valor esteja subindo no heap.

i = parent; Atualiza o valor de i para o índice do pai, onde vai verificando se o nó atual está na posição correta.

parent = (i - 1) / 2; Calcula o novo índice do pai com base no valor atual de i e incrementa (n++) para contar os nós.

No final deste relatório, abaixo do código, deixo um rascunho de um diagrama do heap com a execução deste método.

print_min(): Imprime o menor item no heap. O menor item é o elemento raiz do heap.

print(): Imprime todos os itens do heap, onde são impressos em ordem, com quebras de linha para cada nível.

dim(): Imprime o número de itens no heap.

dim_max(): Imprime o número máximo de itens ou capacidade do heap.

deleteMin(): Uma vez que delete é uma palavra reservada, optei por nomear este método como deleteMin.

Remove o menor item no heap (raiz), onde troca com o último elemento do heap e decrementa o contador, eliminando-o, em seguida, é realizada as respetivas correções no heap, descendo o

elemento trocado até a posição correta e até que a propriedade do heap seja restaurada.

Para o comando **"clear"** inicializei o heap com o setter de n diretamente no main sem usar qualquer método, definindo o número de nós usados como zero, onde o heap fica vazio.

O comando "heapify_up" chama o método "buildMinHeap()" que percorre os nós pais do heap (começando pelo último nó pai (lastParent = (n - 1) / 2)) e chama o método **heapify()** para ajustar cada nó e seus descendentes para garantir que o heap seja um min heap válido.

O método "heapify()" ajusta um nó específico na heap, verificando se ele precisa ser trocado com um dos filhos para manter a propriedade de min heap, onde se a troca for necessária, realiza a troca e chama-se recursivamente para ajustar o nó trocado e os seus descendentes, onde realiza as operações necessárias para ajustar um nó específico e construir um min heap bottom-up.

Para ser mais específico, o método "heapify()" recebe um índice como parâmetro e realiza as seguintes operações:

1. Inicializa a variável smallest com o valor do índice fornecido.
2. Calcula os índices dos filhos esquerdo (left) e direito (right) do nó atual.
3. Verifica se o valor do filho esquerdo é menor que o valor do nó atual e, se for, atualiza o valor de smallest com o índice do filho esquerdo.
4. Verifica se o valor do filho direito é menor que o valor do nó atual e, se for, atualiza o valor de smallest com o índice do filho direito.
5. Se o valor de smallest for diferente do valor do índice original, realiza a troca entre o nó atual e o nó de menor valor (swap(v[index], v[smallest])).
6. Chama-se recursivamente para o novo índice de smallest, de forma a ajustar a subárvore.

No final deste relatório, abaixo do código, deixo um rascunho de um diagrama do heap com a execução deste método.

redim_max(int newNv): Redimensiona o número máximo de itens ou capacidade do heap, onde liberta a memória alocada do vetor anterior com delete, aloca um novo vetor com o novo tamanho especificado por newNv, atualiza a capacidade máxima do heap, onde zera o contador.

Main-minh.cpp:

O main lê os comandos fornecidos pelo input com getline do cin std, ignora as linha em branco e comentadas, cria um stringstream a partir da linha completa e realiza as operações correspondentes dos comandos suportados pelo heap.

Como a função printf() é uma função da linguagem C deve esperar strings no formato C-style e não objetos de string do C++, por isso, uso a função c_str().

minh.h:

```
/*
** file: minh.h
**
** min Heap por comandos
** (binary tree on array)
** UC: 21046 - EDAF @ UAb
** e-fólio B 2022-23
**
** Aluno: 2003730 - Luciano Marafona
*/

// Defina:
// em minh.h as classes da estrutura de dados
// em minh.cpp a implementação dos métodos das classes da estrutura de
dados

#ifndef _MINH_H
#define _MINH_H
```

```
class IMINH {
private:
    int* v;     // vetor com nós
    int n;      // num. nós usados
    int nv;     // dim max do vetor (capacidade)

public:
    IMINH(int nmax = 15);   // cria heap vazio c/ capacidade nmax nós
    ~IMINH();

    // getters e setters
    int* getV() const {
        return v;
    }
    void setV(int* newV) {
        v = newV;
    }
    int getN() const {
        return n;
    }
    void setN(int newN) {
        n = newN;
    }
    int getNv() const {
        return nv;
    }

    void setNv(int newNv) {
        nv = newNv;
    }

    void insert(int item);          // insere um item no heap
    void print_min();               // imprime o menor item no heap
    void print();                   // imprime toda a árvore do heap
    void dim();                     // imprime o número de itens no heap
    void dim_max();                 // imprime o número máximo de itens
ou capacidade do heap
    void deleteMin();               // remove o menor item no heap
    void heapify(int);              // corrige o heap com bottom-up
subindo o elemento até a posição correta
    void buildMinHeap();            // constroi a heap começando do
último nó não folha
    void redim_max(int newNv);      // redimensiona o número máximo de
itens ou capacidade do heap
};

#endif
// EOF
```

minh.cpp:

```
/*
** file: minh.cpp
**
** min Heap por comandos
** (binary tree on array)
** UC: 21046 - EDAF @ UAb
** e-fólio B 2022-23
**
** Aluno: 2003730 - Luciano Marafona
*/

// Defina:
// em minh.h as classes da estrutura de dados
// em minh.cpp a implementação dos métodos das classes da estrutura de
dados
#include <iostream>
#include <cstdio>
#include "minh.h"

using namespace std;

IMINH::IMINH(int nmax) {
    v = new int[nmax];   // Aloca um novo vetor de inteiros com tamanho
nmax
    n = 0;               // Inicializa o número de nós usados como 0
    nv = nmax;           // Define a capacidade máxima do vetor como nmax
}

IMINH::~IMINH() {
    delete[] v;          // Liberta a memória alocada para o vetor
}

void IMINH::insert(int item) {
    v[n] = item;         // Insere o item no final do vetor
    int i = n;
    int parent = (i - 1) / 2;   // Calcula o índice do pai do nó atual

    // Realiza a correção do heap, subindo o elemento inserido até a
posição correta
    while (i > 0 && v[i] < v[parent]) {
        swap(v[i], v[parent]);  // Troca o nó com seu pai se necessário
        i = parent;             // Atualiza o índice para o pai
        parent = (i - 1) / 2;   // Calcula o novo índice do pai
    }
    n++;    // Incrementa o número de nós usados
}
```

```
void IMINH::print_min() {
    if(n != 0)
       printf("Min= %d\n", v[0]);// Imprime o valor mínimo, que está na
raiz do heap
}

void IMINH::print() {
    int levelSize = 1;
    int levelCount = 0;
      if (n != 0)
         printf("Heap=\n");





// Imprime os elementos do heap em ordem, com quebras de linha para cada
nível
   for (int i = 0; i < n; i++) {
      if(levelCount + 1 == levelSize || i == n-1)
          printf("%d\n", v[i]);
      else
          printf("%d ", v[i]);

       levelCount++;
       if (levelCount == levelSize) {
           levelSize *= 2; // O próximo nível terá o dobro do tamanho do
nível atual
           levelCount = 0; // Reinicia a contagem de elementos para o
próximo nível
       }
    }
}


void IMINH::dim() {
    printf("Heap tem %d itens\n", n);   // Imprime o número de itens no
heap
}

void IMINH::dim_max() {
   printf("Heap tem capacidade %d itens\n", nv); // Imprime capacidade do
heap
}
```

```
void IMINH::deleteMin() {
    if (n == 0) {
        printf("Comando delete: Heap vazio!\n");
        return;
    }

    swap(v[0], v[n - 1]);  // Troca o menor elemento com o último
elemento
    n--;                   // Decrementa o número de nós usados

    // Realiza a correção do heap, descendo o elemento trocado até a
posição correta
    int i = 0;
    while (true) {
        int leftChild = 2 * i + 1;
        int rightChild = 2 * i + 2;
        int smallest = i;

        // Verifica se o filho esquerdo existe e se é menor que o nó
atual
        if (leftChild < n && v[leftChild] < v[smallest]) {
            smallest = leftChild;
        }

        // Verifica se o filho direito existe e se é menor que o nó atual
ou o filho esquerdo
        if (rightChild < n && v[rightChild] < v[smallest]) {
            smallest = rightChild;
        }

        // Se o nó atual não for o menor, troca com o filho de menor
valor
        if (smallest != i) {
            swap(v[i], v[smallest]);
            i = smallest;
        } else {
            break;
        }
    }
}
```

```
// Função para ajustar um nó específico na heap
void IMINH::heapify(int index) {
    int smallest = index; // Inicialmente, o nó atual será considerado o
menor
    int left = 2 * index + 1; // Filho esquerdo
    int right = 2 * index + 2; // Filho direito

    // Verifica se o filho esquerdo é menor que o nó atual
    if (left < n && v[left] < v[smallest])
        smallest = left;

    // Verifica se o filho direito é menor que o nó atual
    if (right < n && v[right] < v[smallest])
        smallest = right;

    // Se o menor valor não for o nó atual, faz a troca e ajusta
recursivamente
    if (smallest != index) {
        swap(v[index], v[smallest]);
        heapify(smallest);
    }
}


// Função para construir uma min heap bottom-up
void IMINH::buildMinHeap() {
    // Último nó com filhos 8não folha) é o nó pai do último nó da heap
    int lastParent = (n - 1) / 2;

    // Percorre os nós pais da heap e realiza o ajuste com heapify(i)
    for (int i = lastParent; i >= 0; i--)
        heapify(i);
}



void IMINH::redim_max(int newNv) {
    delete[] v;         //remove o vetor
    v = new int[newNv]; // aloca a memória para o novo vetor
redimensionado
    nv = newNv;         //atualiza capacidade do heap
    n = 0;              // descarta todo o conteúdo anterior
}


// EOF
```

Main-minh.cpp:

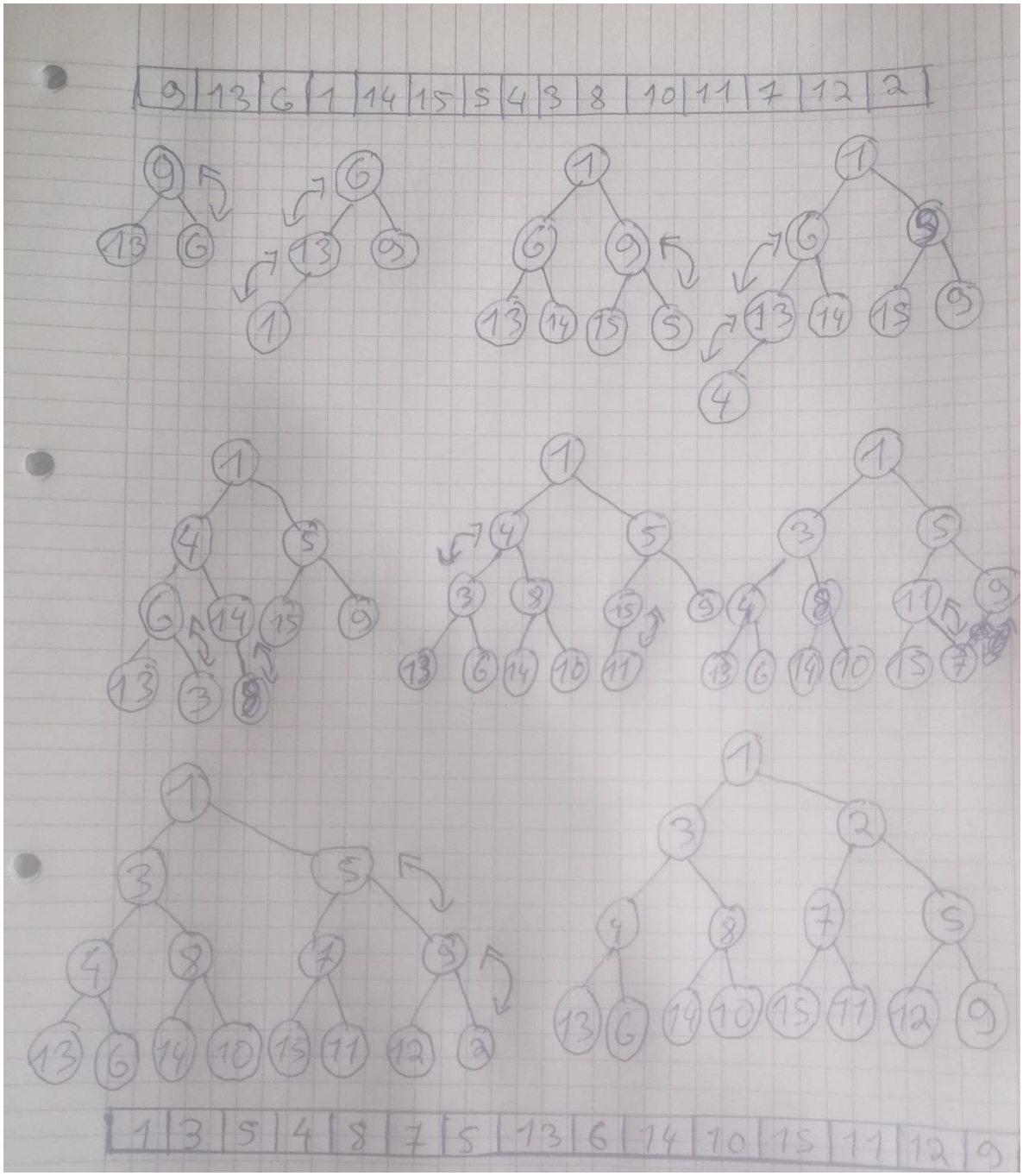
```
/*
** file: main-minh.cpp
**
** min Heap por comandos
** (binary tree on array)
** UC: 21046 - EDAF @ UAb
** e-fólio B 2022-23
**
** Aluno: 2003730 - Luciano Marafona
*/

// Defina:
// em minh.h as classes da estrutura de dados
// em minh.cpp a implementação dos métodos das classes da estrutura de
dados
// em main-minh.cpp
//   A entrada/saída de dados
//   As instâncias da classe da estrutura de dados
//   A implementação dos comandos através dos métodos da classe
//   Código auxiliar
//   Não utilize variáveis globais


#include <iostream>
#include <cstdio>
#include <sstream>
#include <string>
#include "minh.h"



int main()
{
   IMINH minheap;   // exemplo

    std::string line;
    // Lê os comandos do utilizador enquanto houver entrada
    while (std::getline(std::cin, line)) {
        // Se for linha vazia ou comentário, ignora
        if (line.empty() || line[0] == '#') {
            continue;
        }

        // Cria um stringstream a partir da linha completa
        std::stringstream ss(line);

        std::string command;
```

```
        // Lê o comando da primeira palavra na linha
        ss >> command;

        // Comando "insert": insere os itens fornecidos no heap
        if (command == "insert") {
          int item;
          while (ss >> std::ws >> item) {
            if (minheap.getN() == minheap.getNv()) {
               //c_str() para o printf que espera formato C-style e não
objetos de string do c++.
                printf("Comando %s: Heap cheio!\n", command.c_str());
                break;// pára de inserir
            }
             minheap.insert(item);
          }
        }


        // Comando "print_min": imprime o menor item no heap
        else if (command == "print_min") {
            if (minheap.getN() == 0){
                printf("Comando %s: Heap vazio!\n", command.c_str());
            }
            minheap.print_min();
        }
        // Comando "print": imprime todos os itens no heap
        else if (command == "print") {
            if (minheap.getN() == 0){
                printf("Comando %s: Heap vazio!\n", command.c_str());
            }
            minheap.print();
        }
        // Comando "dim": imprime o número de itens no heap
        else if (command == "dim") {
            minheap.dim();
        }
        // Comando "dim_max": imprime o número máximo de itens no heap
        else if (command == "dim_max") {
            minheap.dim_max();
        }
        // Comando "clear": inicializa o heap com zero elementos
        else if (command == "clear") {
           minheap.setN(0);
           printf("Comando %s: Heap vazio!\n", command.c_str());
        }
```

```
        // Comando "delete": remove o menor item no heap
        else if (command == "delete") {
            if (minheap.getN() == 0) {
               printf("Comando %s: Heap vazio!\n", command.c_str());
            } else {
               minheap.deleteMin();
            }
        }
        // Comando "redim_max N": redimensiona o número máximo de itens
ou capacidade do heap
        else if (command == "redim_max") {
           int newNv;
           if (ss >> newNv) {
              minheap.redim_max(newNv);
           }
        }




        // Comando "heapify_up index": converte o vetor definido pelos
itens "item ..." num min Heap. Todo o conteúdo anterior do heap é
descartado/perdido
        else if (command == "heapify_up") {
           int i, item;
           minheap.setN(0); // limpa o vetor anterior
           for (i = 0; i < minheap.getNv() && ss >> std::ws >> item; i++)
{
               minheap.getV()[i] = item;
               minheap.setN(i+1);// aproveito o i para contar o n
           }
           minheap.buildMinHeap();
        }
        // Comando inválido
        else{
             printf("Comando %s é inválido!\n", command.c_str());
        }
     }// fim while

   return 0;
}
// EOF
```

Rascunho de um diagrama do heap com a execução do método “inserir” com os valores do teste 11 do VPL.

Aqui conforme os valores forem inseridos, são automaticamente corrigidos um a um até às folhas do heap.



Rascunho de um diagrama do heap com a execução do método “heapify” do algoritmo bottom-up com os valores do teste 11 do VPL.

Aqui depois dos valores inseridos, começa a corrigir de baixo para cima até à raiz, subindo sempre o mais pequeno dos seus descendentes.