Estrutura de Dados

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

7 de fevereiro de 2024

Heaps e Aplicações

Heap

- Heap é conceito mais amplo de árvores que pode ser implementado na forma de um vetor.
- Um exemplo de Heap é o Heap Binário (*Binary Heap*) que implementa uma árvore binária na forma de um vetor.
- Como a estrutura segue uma classificação de vetor, o Heap Binário representa uma árvore binária (quase-)completa.
- Outros tipos de Heap:
  - Heap d-ário: baseado em uma árvore d-ária.
  - Heap Binomial: Uma coleção de árvores binomiais (de fatores de potência de 2).
  - Heap de Fibonacci, Beap, Heap 2-3, ...

Heap

- O Heap é uma estrutura baseada em árvores ordenadas (Heap Máximo, ou Heap Mínimo).
  - O nó pai não é maior que seus filhos. (Ou menor).
  - O heap pode ser uma coleção de árvores deste tipo.
- Como estrutura de dados dinâmica o Heap traz consigo algumas operações:
  - *BuscaMax* ou *BuscaMin* dependendo de ser um Heap Máximo ou Mínimo como determinante de prioridade.
  - Inserir - Insere um elemento na fila, respeitando a prioridade
  - RemoverMax (RemoverMin) - Remove um elemento na fila.
  - AlterarChave - Altera o valor de um elemento da fila, alterando sua prioridade.
  - ConstruirHeap - A partir de uma sequência de valores, transformar a sequência em um Heap.

Heap

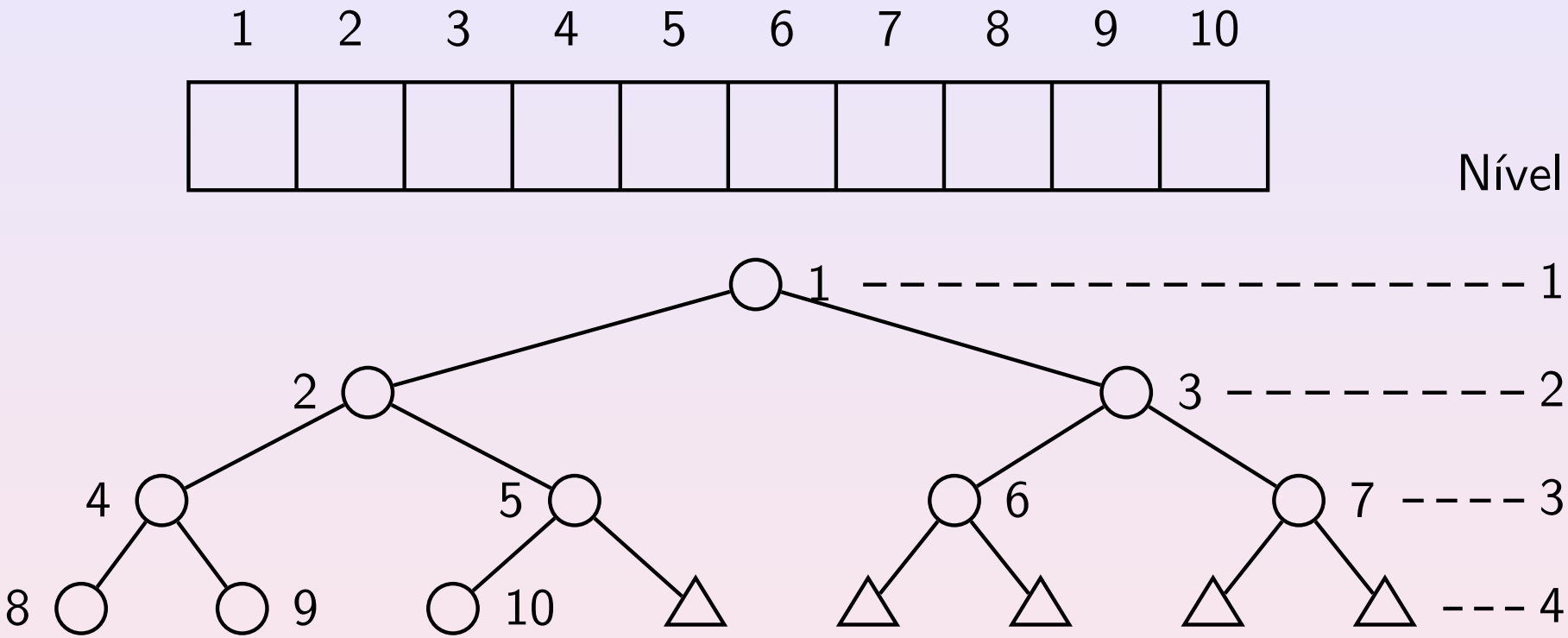
- O Heap é eficiente para implementar filas de prioridade.
- Podemos comparar os tempos em uma fila de prioridade considerando algumas implementações de Heap:

Operação	**Fila Simples**	**Fila Ordenada**	**Heap Binário**	**Heap Binomial**	**Heap de Fibonacci**
BuscaMax	$O(n)$	$O(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
Inserir	$\Theta(1)$	$O(n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$
RemoverMax	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
AlterarChave	$\Theta(1)$	$O(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$

Heap Binário

- O heap binário é a representação de uma árvore binária única (quase-)completa.
- A representação da árvore acontece em uma sequência linear de seus nós:
  - O raiz é o primeiro nó, seus filhos os nós seguintes, os descendentes de segundo nível...

Heap Binário



Definição

- Considere um vetor $A[1, .., n]$ representando um Heap.
- Cada posição do vetor corresponde a um nó da árvore do Heap.
- O pai de um nó i é $\lfloor i/2 \rfloor$.
- Um nó i tem $2i$ como filho esquerdo e $2i + 1$ como filho direito.
- Naturalmente, o nó i tem filho esquerdo apenas se $2i \leqslant n$. e tem filho direito apenas se $2i + 1 \leqslant n$.
- Um nó i é uma folha se não tem, ou seja, se $2i > n$.
- As folhas são $\lfloor n/2 \rfloor + 1, \ldots, n - 1, n$.

Níveis do Heap

- Cada nível p, exceto talvez o último, tem exatamente 2^{p-1} nós, e esses são:
  $2^{p-1}, 2^{p-1}+1, 2^{p-1}+2, \ldots, 2^p - 1$
- O nó i pertence ao nível $\lfloor \log i \rfloor$.
- O número total de níveis é $1 + \lfloor \log n \rfloor$

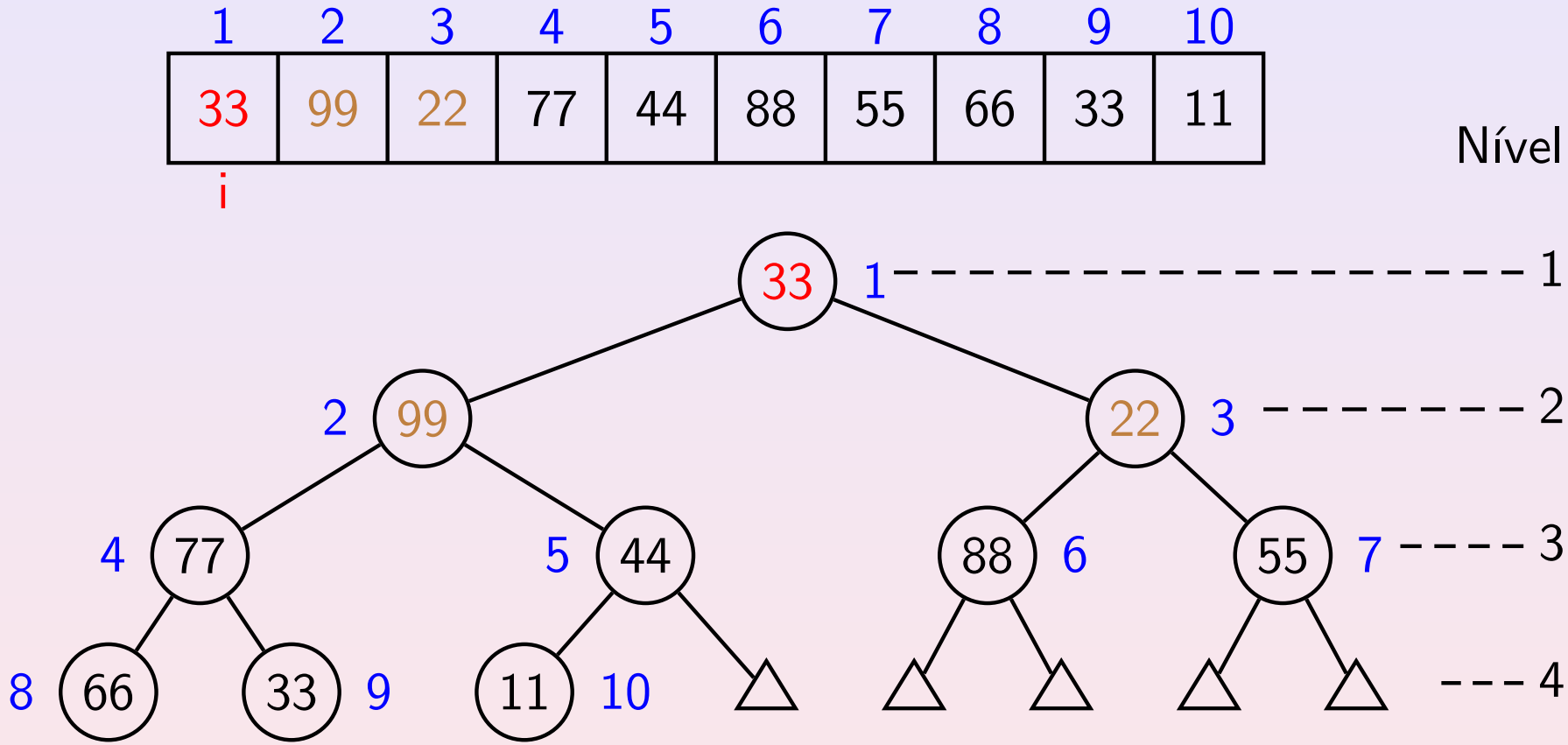
Max-Heap e Min-Heap

- Um nó i satisfaz a propriedade de (max)-heap se:
  - $A[\lfloor i/2 \rfloor] \geqslant A[i]$ (ou seja, pai $\geqslant$ filho).
- Uma árvore binária completa é um (max)-heap se todo nó distinto da raiz satisfizer a propriedade de (max)-heap.
- O (máx)imo ou (mai)or elemento de um (max)-heap está na raiz da árvore.
- De forma análoga é definido o Min-Heap (basta trocar max por min).

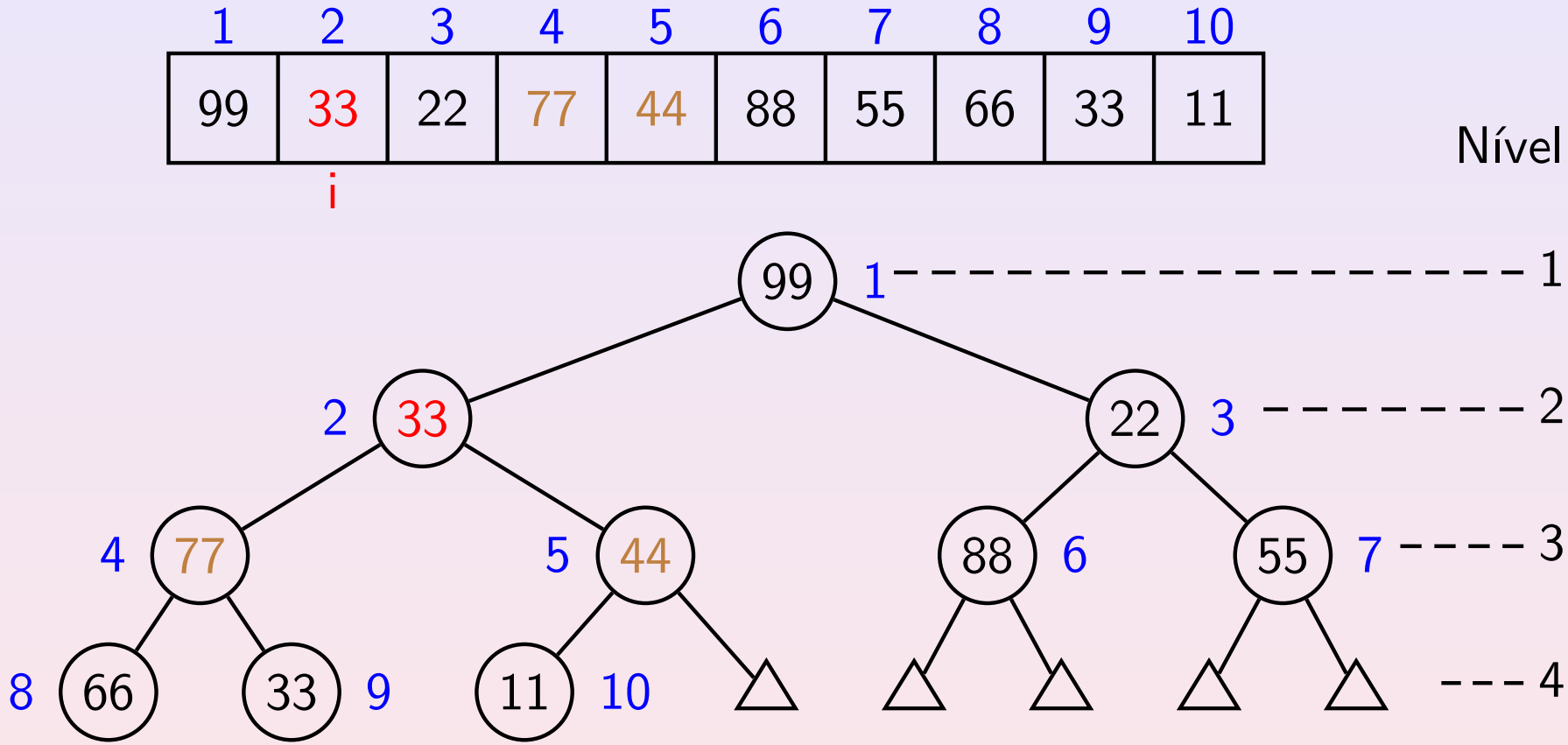
Construindo um Heap Máximo

- São Duas Etapas:
  1. Max-Heapfy (Joga um elemento menor para uma folha)
     - Pegamos um elemento na posição i e comparamos com seus filhos: $2i$ e $2i + 1$.
     - Trocamos com o maior.
     - Continuamos recursivamente a partir da nova posição deste elemento.
  2. Build-Max-Heap (Construir o Heap-Máximo)
     - A partir do primeiro elemento não folha (de trás para frente): $n/2$
     - Iterativamente aplicamos o algoritmo Max-Heapfy até o elemento na posição 1.
- Garantimos a cada iteração que os pais são sempre os maiores e que os menores elementos sempre são jogados para as folhas.

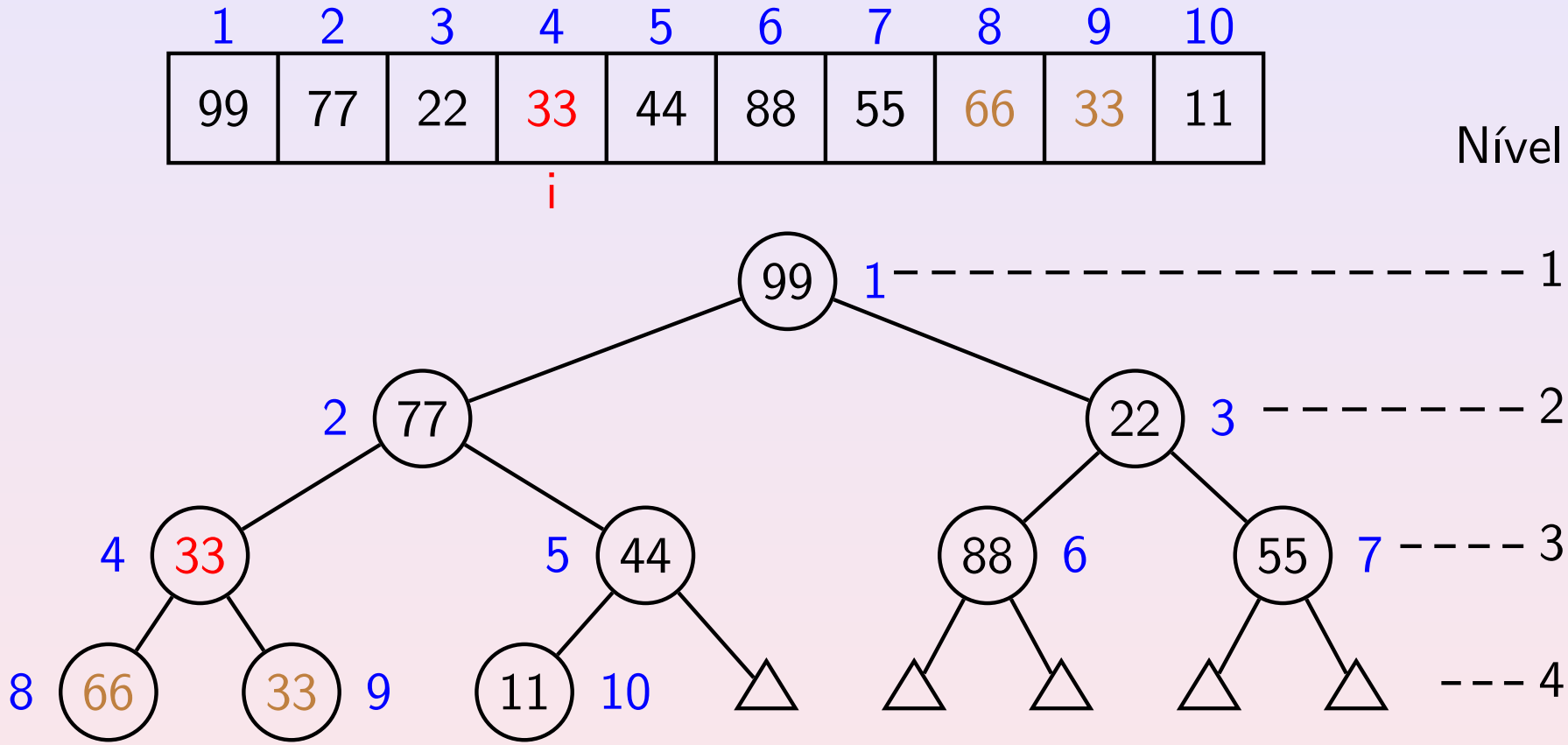
Max-Heapfy


1 2 3 4 5 6 7 8 9 10
33 99 22 77 44 88 55 66 33 11
i
Nível
33
1
1
2
99
22
3
2
4
77
5
44
88
6
55
7
3
8
66
33
9
11
10
4

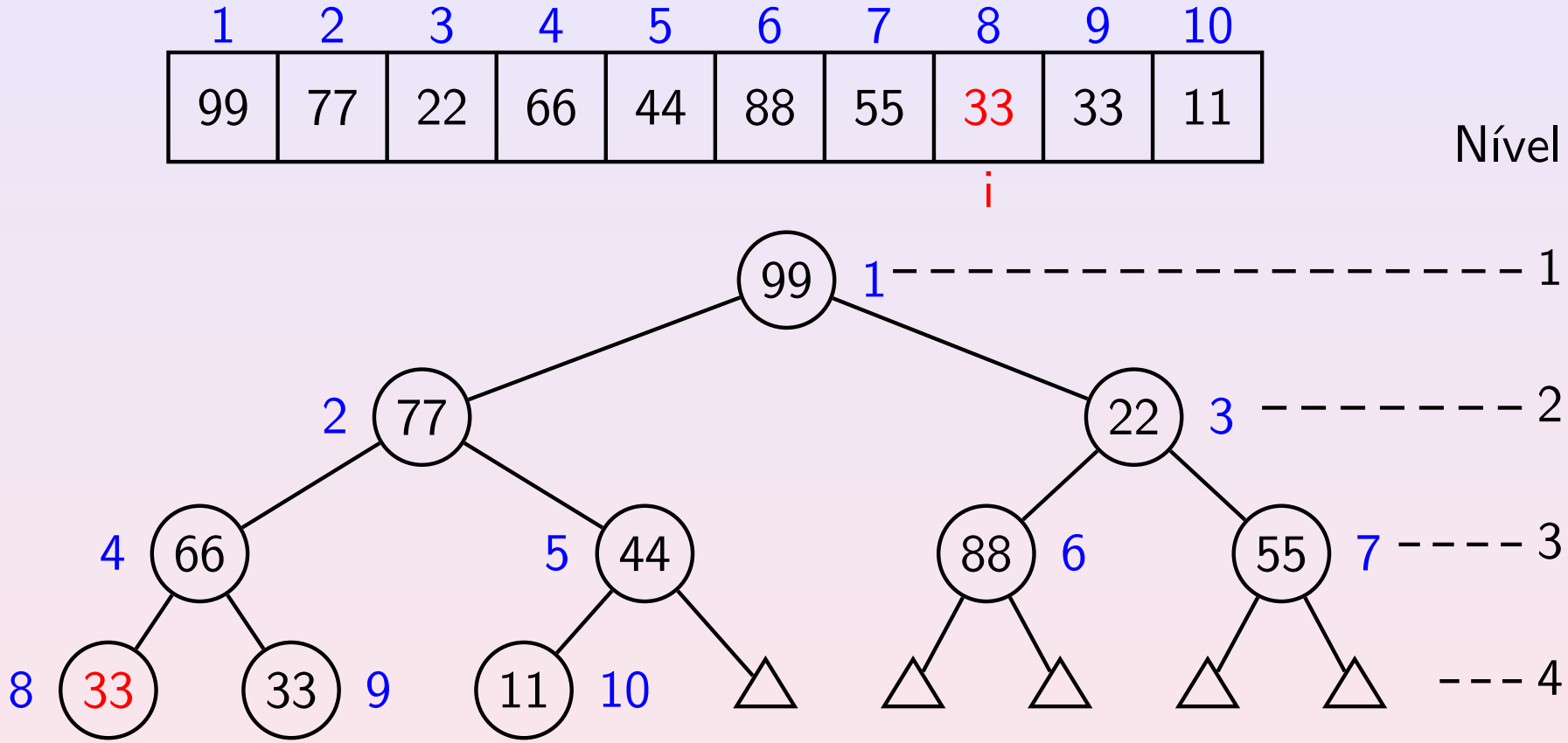
Max-Heapfy


1 2 3 4 5 6 7 8 9 10
99 33 22 77 44 88 55 66 33 11
i
Nível
99
1
1
2 33
22
3
2
4 77
5 44
88
6
55
7
3
8 66
33
9
11
10
4

Max-Heapfy


1 2 3 4 5 6 7 8 9 10
99 77 22 33 44 88 55 66 33 11
i
Nível
99
1
1
2
77
22
3
2
4
33
5
44
88
6
55
7
3
8
66
33
9
11
10
4

Max-Heapfy



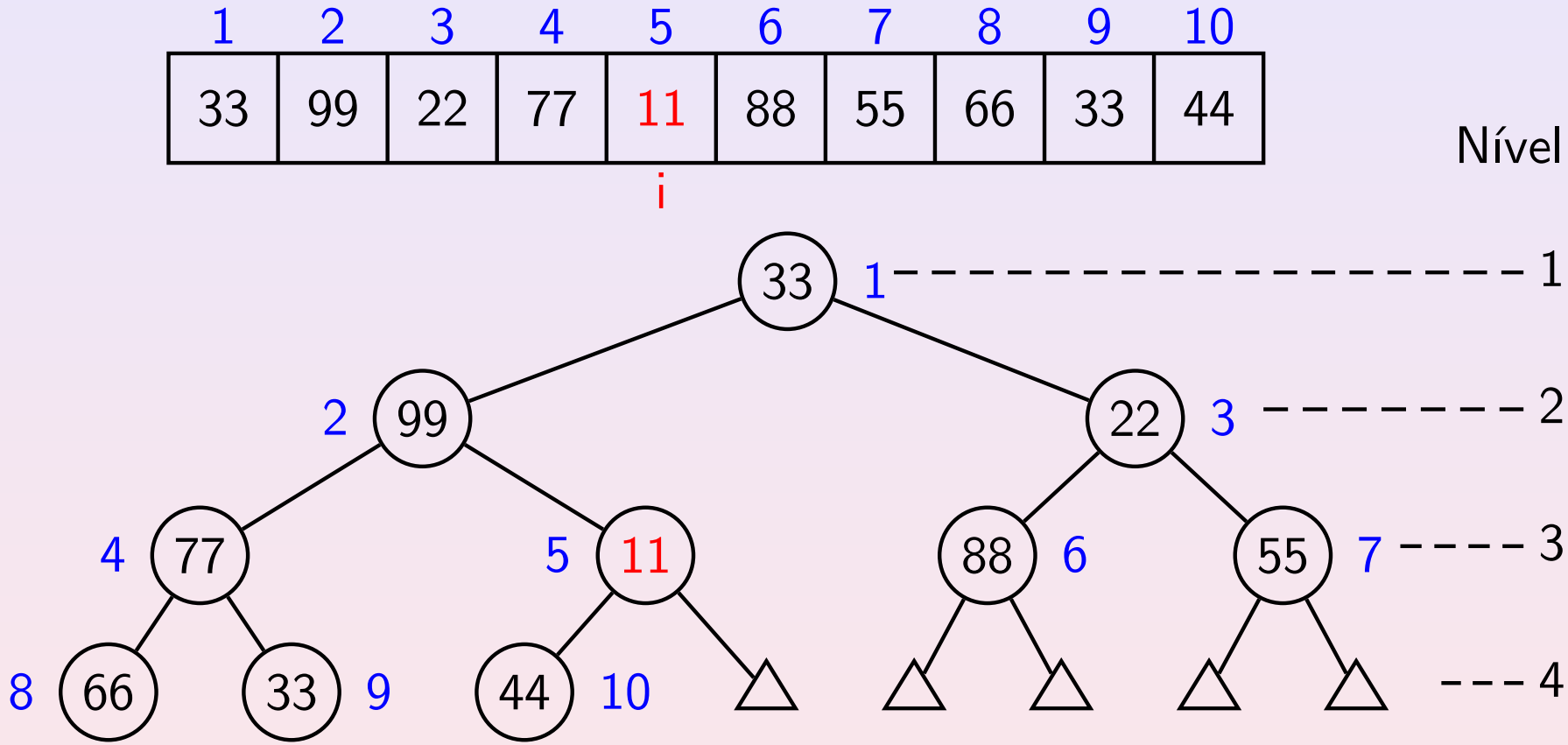
Este Heap não é máximo, mas o elemento “33” não tem descendentes menores que ele.

Algoritmo Max-Heapfy

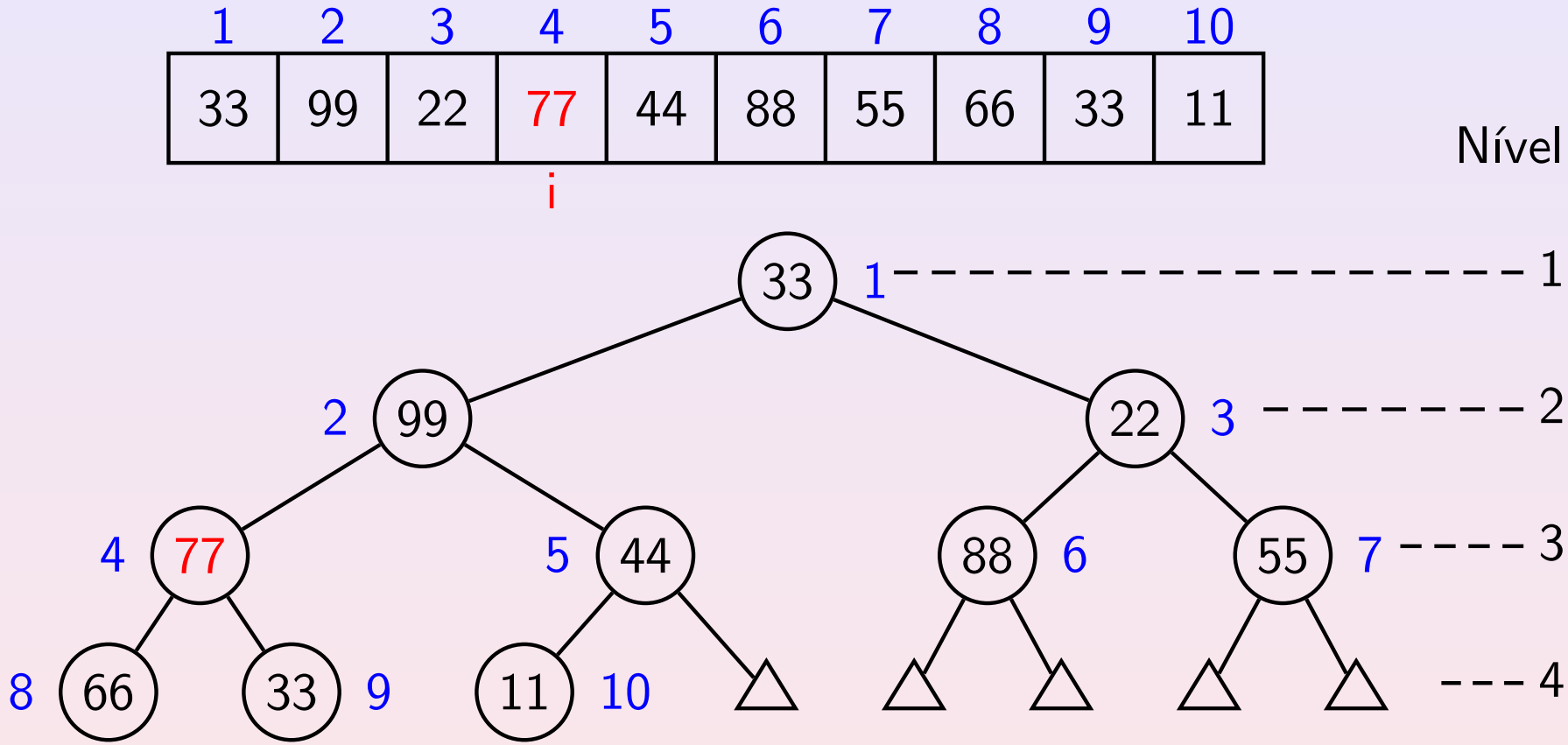
Algoritmo MAX_HEAPFY(A,n,i)

$e \leftarrow 2 * i$
$d \leftarrow 2 * i + 1$
se $e \leqslant n$ **e** $A[e] > A[i]$ **então**
    $max \leftarrow e$
senão
    $max \leftarrow i$
se $d \leqslant n$ **e** $A[d] > A[max]$ **então**
    $max \leftarrow d$
se $max \neq i$ **então**
    $A[i] \leftrightarrow A[max]$
    Max_Heapfy(A, n, max)

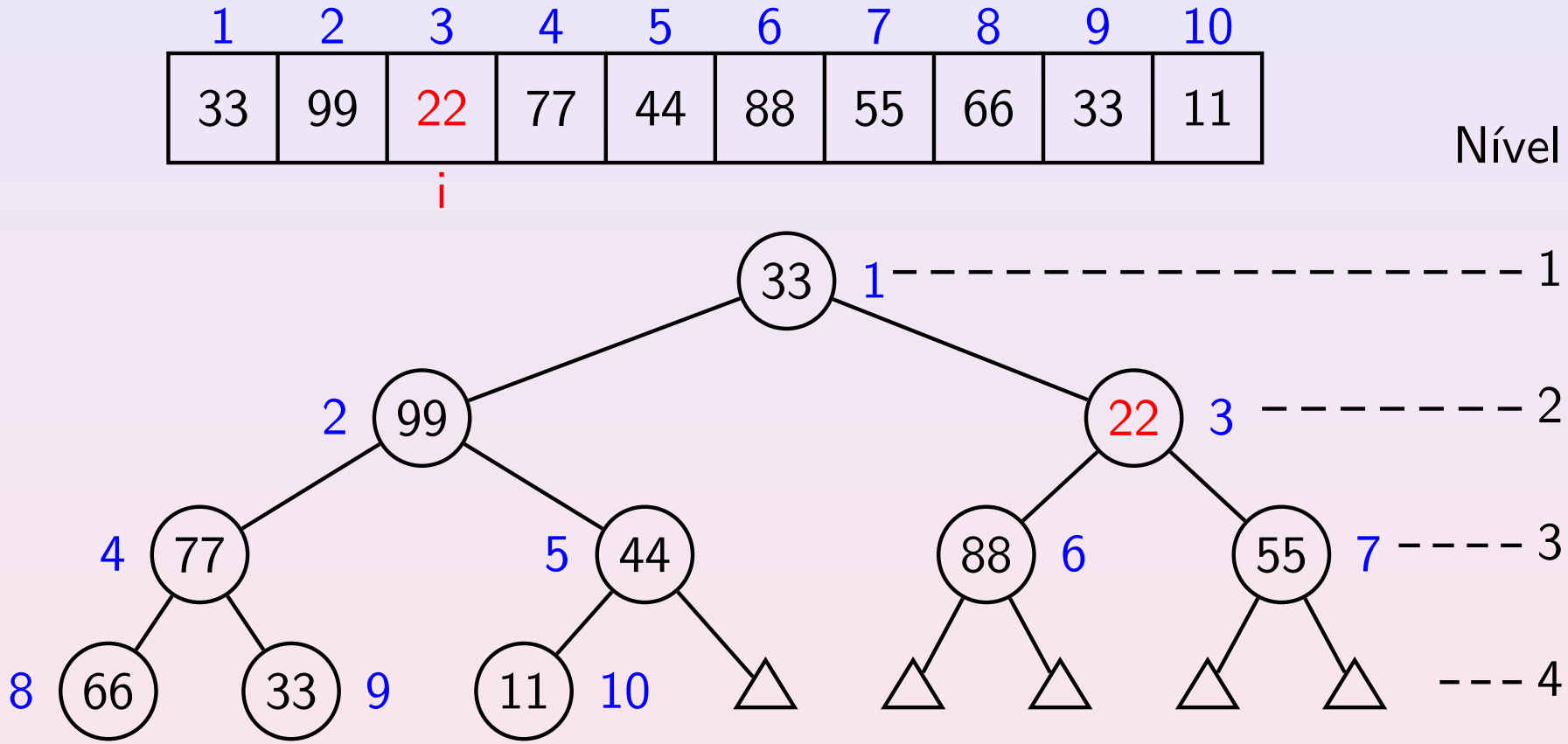
Build-Max-Heap



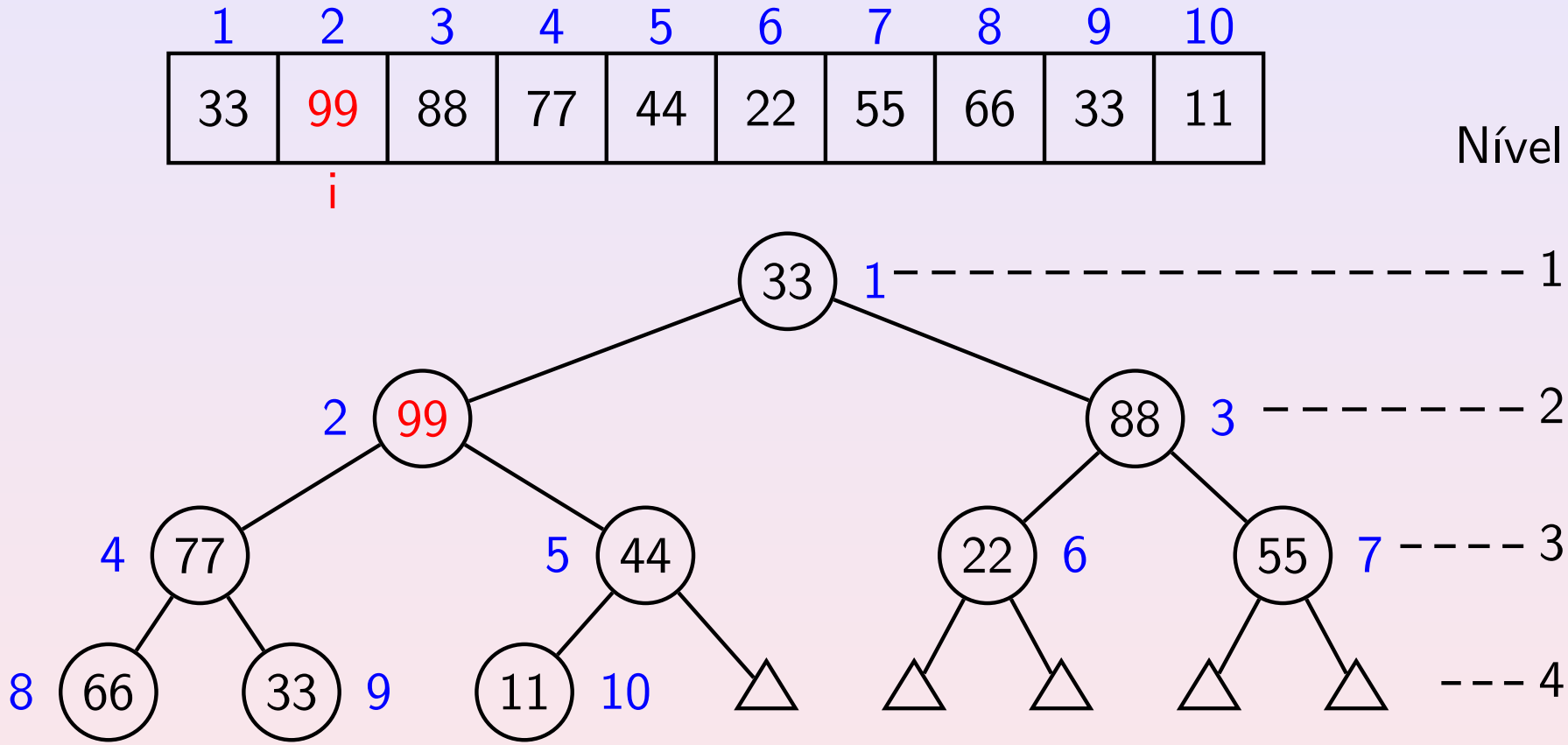
Build-Max-Heap



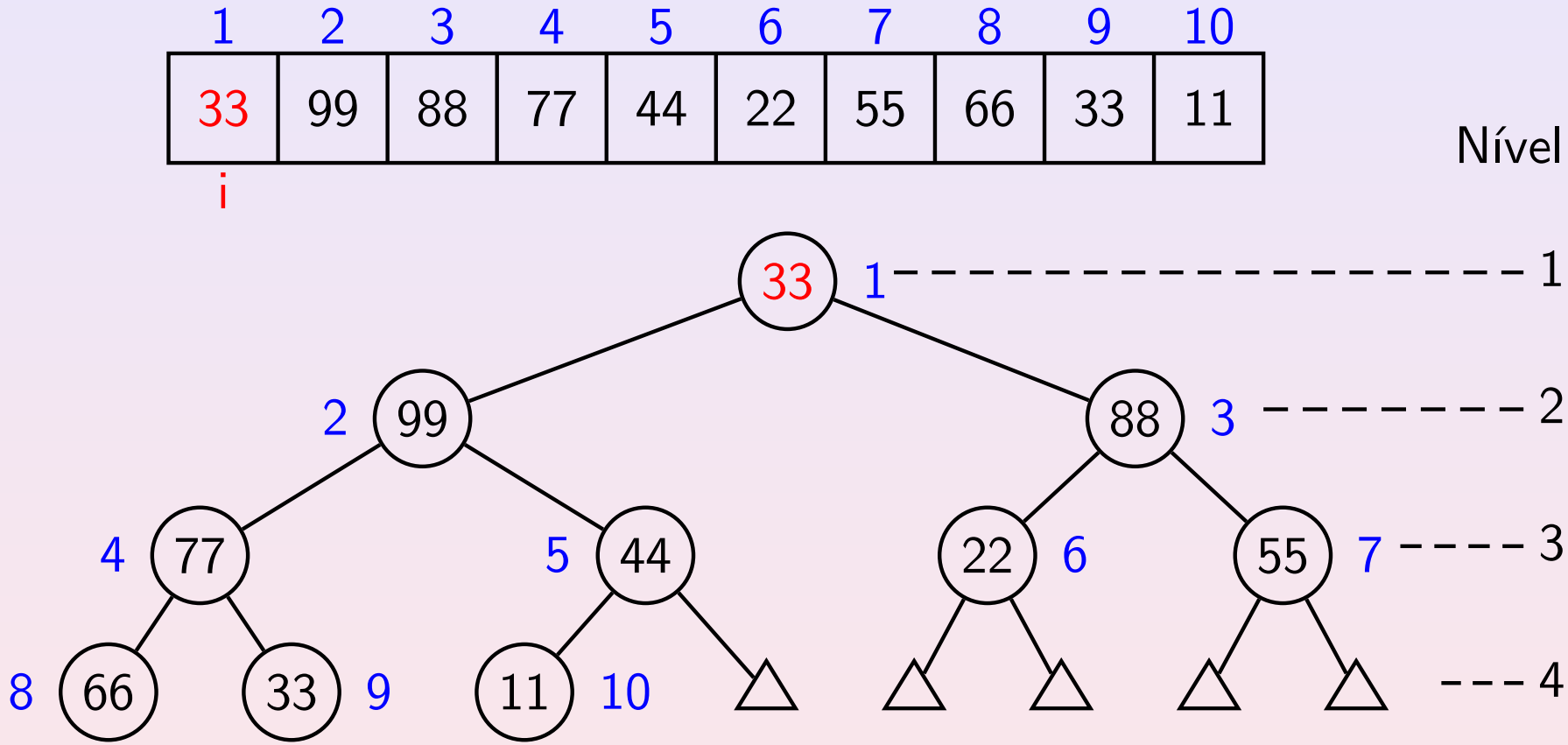
Build-Max-Heap



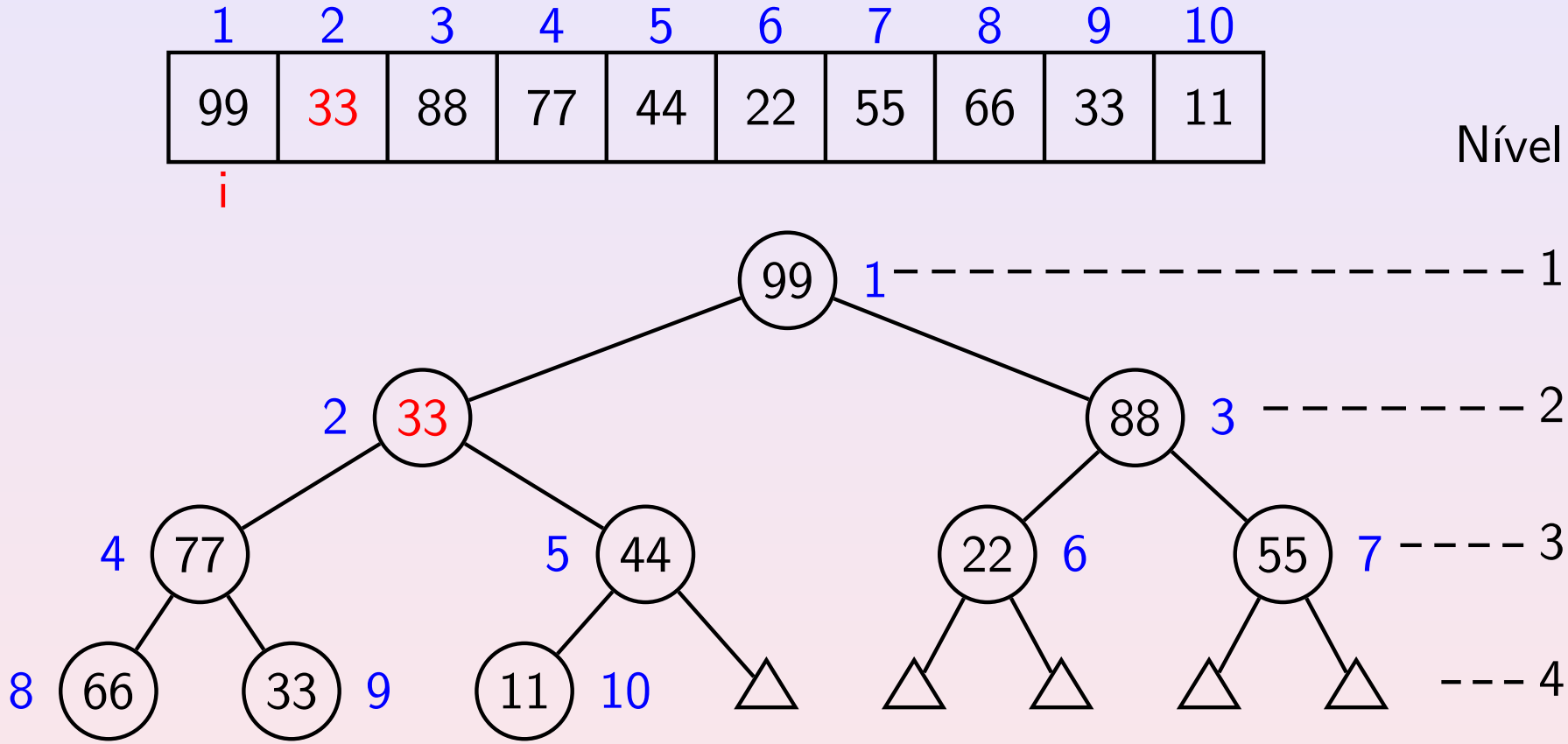
Build-Max-Heap



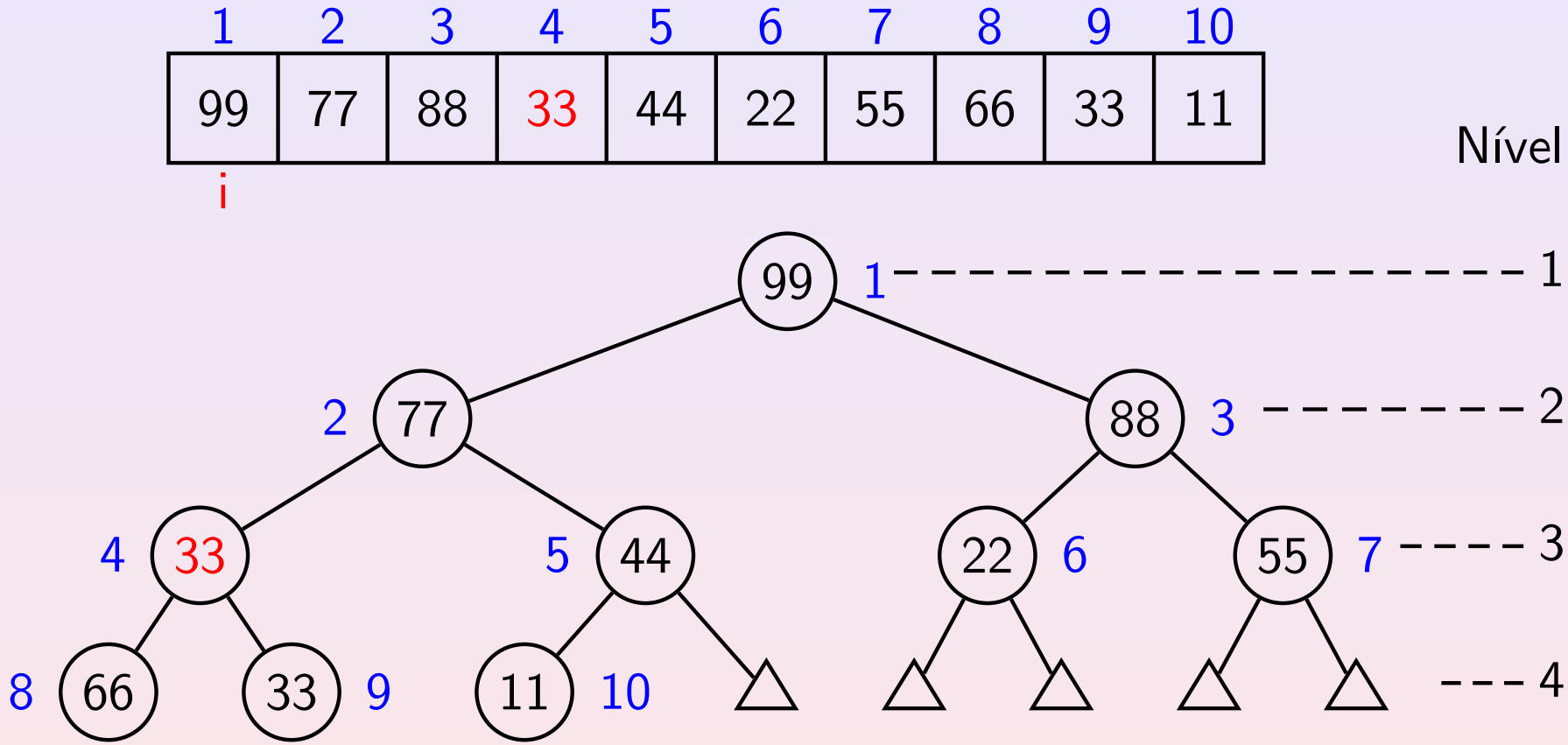
Build-Max-Heap



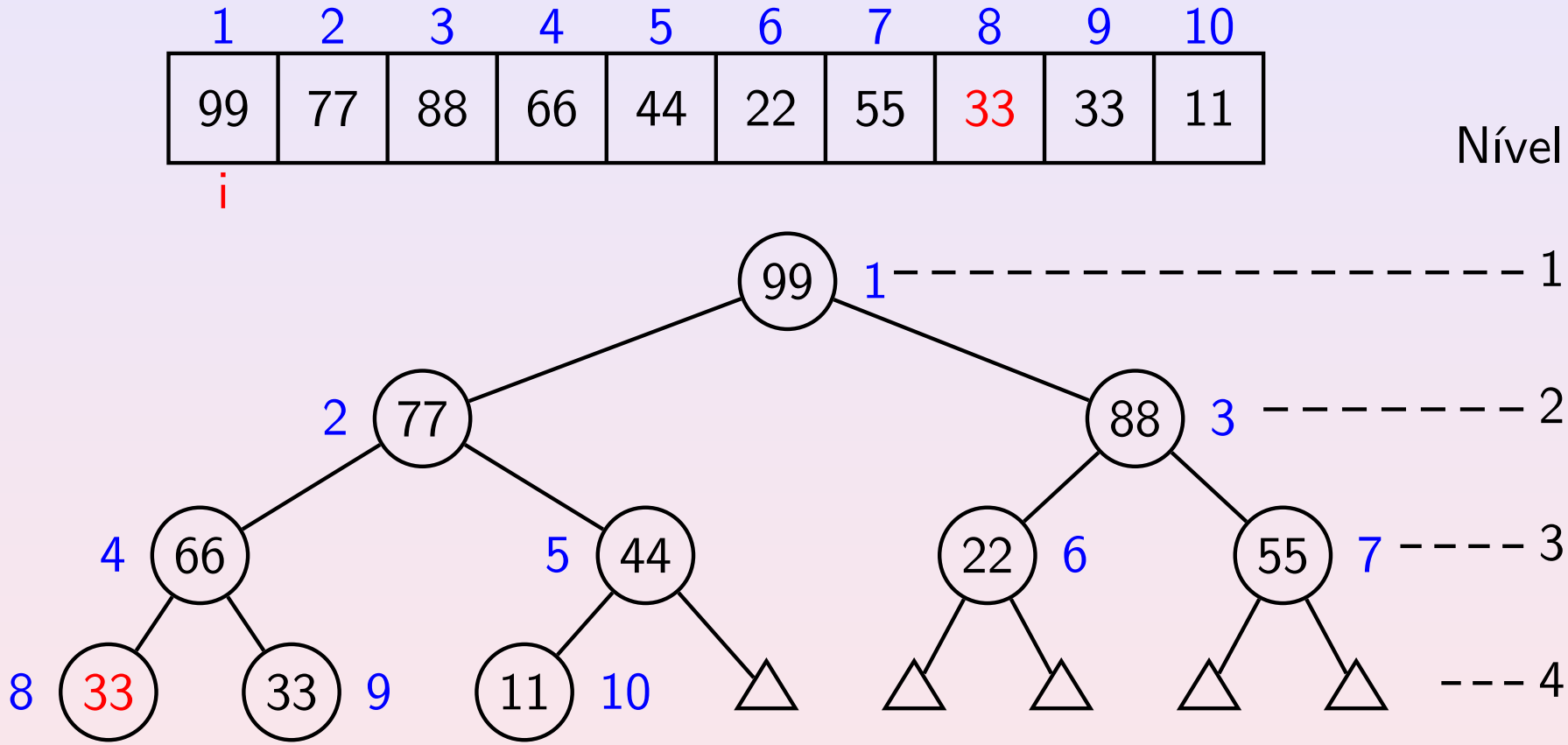
Build-Max-Heap



Build-Max-Heap

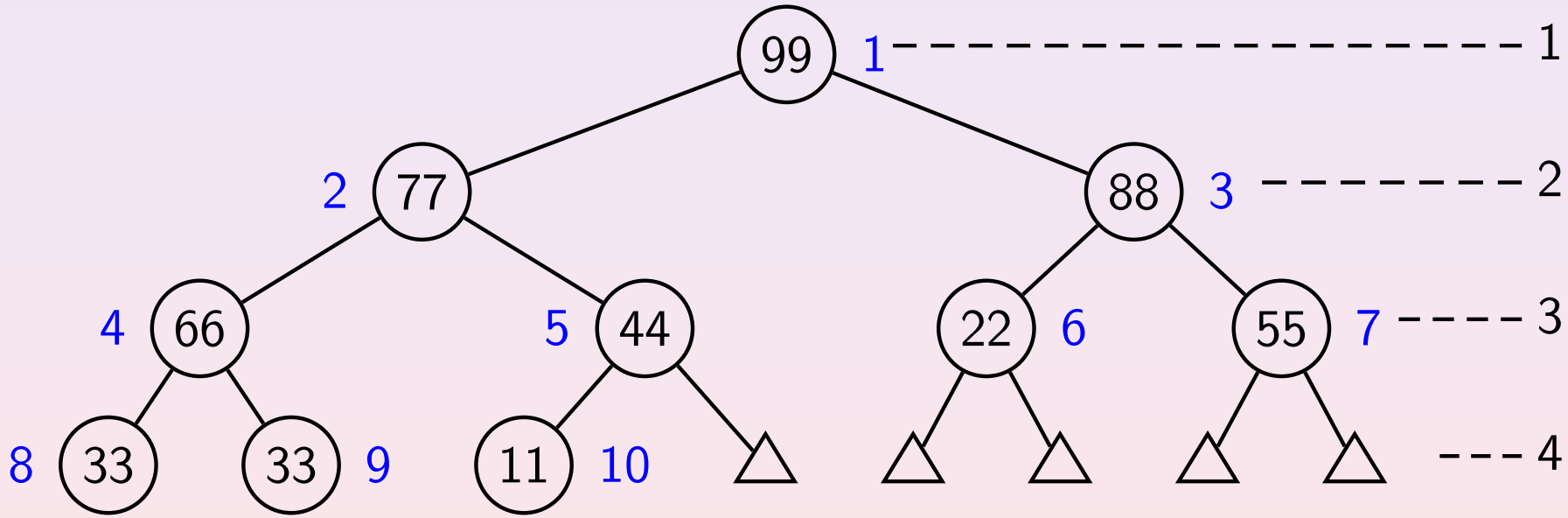


Build-Max-Heap



Build-Max-Heap

1	2	3	4	5	6	7	8	9	10
99	77	88	66	44	22	55	33	33	11



Heap Máximo atingido

Algoritmo Build_Max_Heap

Algoritmo BUILD_MAX_HEAP(A,n)
para $i \leftarrow \lfloor n/2 \rfloor$ **até** 1 **faça**
Max_Heapfy(A, n, i)

Heap Binário como Fila de Prioridade

- Inserir um elemento é $O(\log n)$:
  - A fila inicialmente deve ser um Heap-Máximo.
  - Insere-se o elemento no final da fila.
  - Ele vai trocando de posição com o pai, enquanto for maior que o mesmo.
  - No final, continuaremos com um Heap-Máximo.
- A busca de um elemento é $\Theta(1)$: é o primeiro da fila.
- Remover um elemento é $\Theta(\log n)$:
  - Pegamos o último elemento da fila e colocamos como primeiro.
  - Aplica-se o Max-Heapfy.

Definição de Fila de Prioridade baseada em heap

```
typedef struct heapfila {
  int itens[TAMANHOFILA];
  int fim; // Marcadores de 1 a n --> vetor de 0 a n-1.
} heapfila;
```

Inserir um elemento na Fila de Prioridade

```
void enfileirar(heapfila *p, int obj) {
  int i;

  if(p->fim != TAMANHOFILA) {
    p->itens[p->fim++]=obj;
    i = p->fim;
    while((i > 1) && (p->itens[i-1] > p->itens[i/2 - 1])) {
      int aux = p->itens[i - 1] ; //p->itens[i - 1] ↔ p->itens[i/2 - 1]
      p->itens[i - 1] = p->itens[i/2 - 1];
      p->itens[i/2 - 1] = aux;
      i = i/2;
    }
  }
  return;
}
```

Remover um elemento da Fila de Prioridade

```
int desenfileirar(heapfila *p) {
  int i = 1, max=1;
  int o, para = 0;
  o = p->itens[0]; //elemento que sai
  p->itens[0] = p->itens[--p->fim]; //ultimo vai para o começo da fila
  while(!para) { // max-heapfy
    if(((2*i) <= p->fim) && (p->itens[i-1] < p->itens[2*i-1])) max = 2*i;
    if(((2*i+1) <= p->fim) && (p->itens[max-i] < p->itens[2*i]))
      max = 2*i+1;
    if(i != max) {
      int aux = p->itens[i-1];
      p->itens[i-1] = p->itens[max-1];
      p->itens[max-1] = aux;
      i = max;
    }
    else para = verdade;
  }
  return o;
}
```

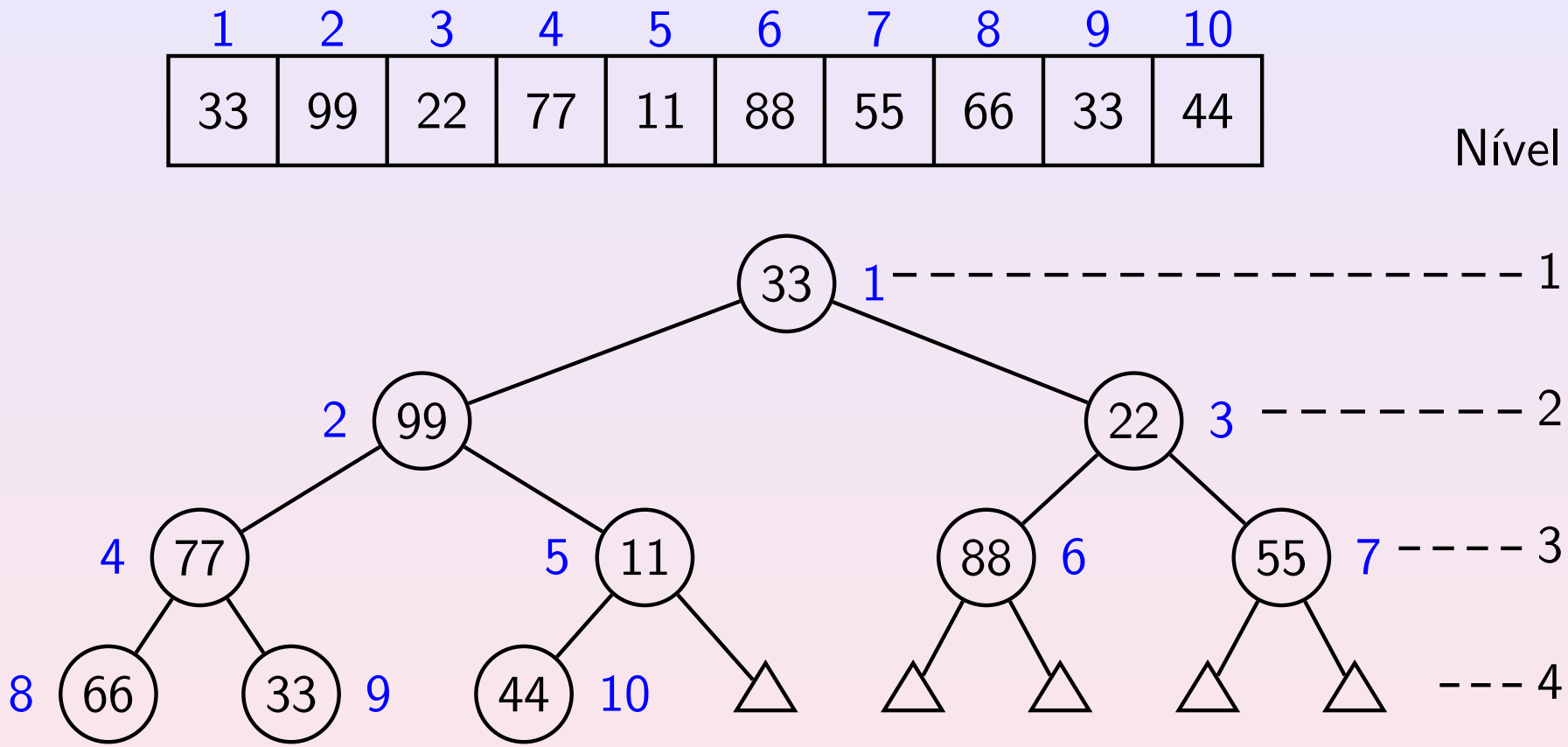
Aplicações de filas de prioridade

- Vários algoritmos importantes utilizam fila de prioridade para processar seus elementos, veja alguns:
  - Algoritmo de Ordenação por Heap.
  - Compressão de dados de Huffamn.
  - Algoritmo de Dijkstra de caminhos mínimos em Grafos.
  - Gerenciamento de fila de processamento (quando baseada em prioridade).
  - Roteamento de pacotes (pacote de voz tem prioridade).
  - Algoritmo de Prim para árvore geradora mínima em Grafos.

Heap Sort

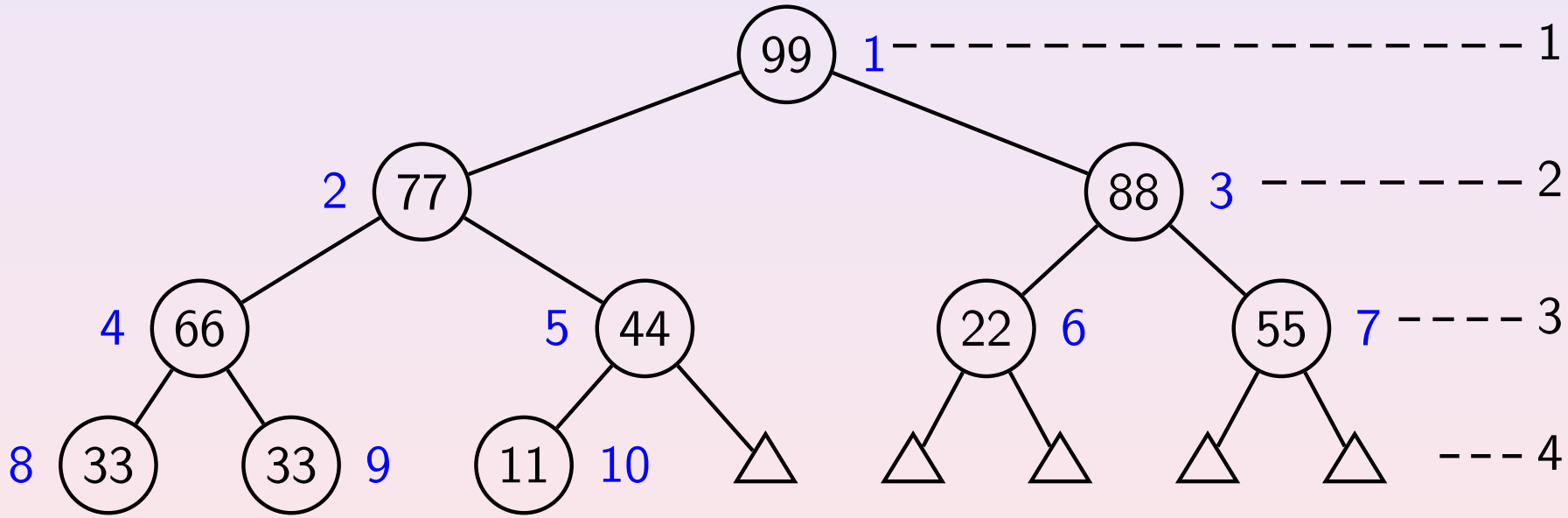
- O algoritmo de ordenação por Seleção em estrutura Heap segue as sequintes etapas:
  1. Primeiro construimos um Heap Máximo.
  2. O primeiro elemento é o maior da sequência. Trocamos este com o último, o último já está na posição. Vamos considerar agora a sequência sem este elemento.
  3. Aplicamos um Max-Heapfy nos $n - 1$ elementos restantes e o elemento pequeno que ficou no raiz é jogado para uma folha.
  4. Repete-se o processo até que sobre somente um elemento na sequência considerada.

Ordenação: Build-Max-Heap



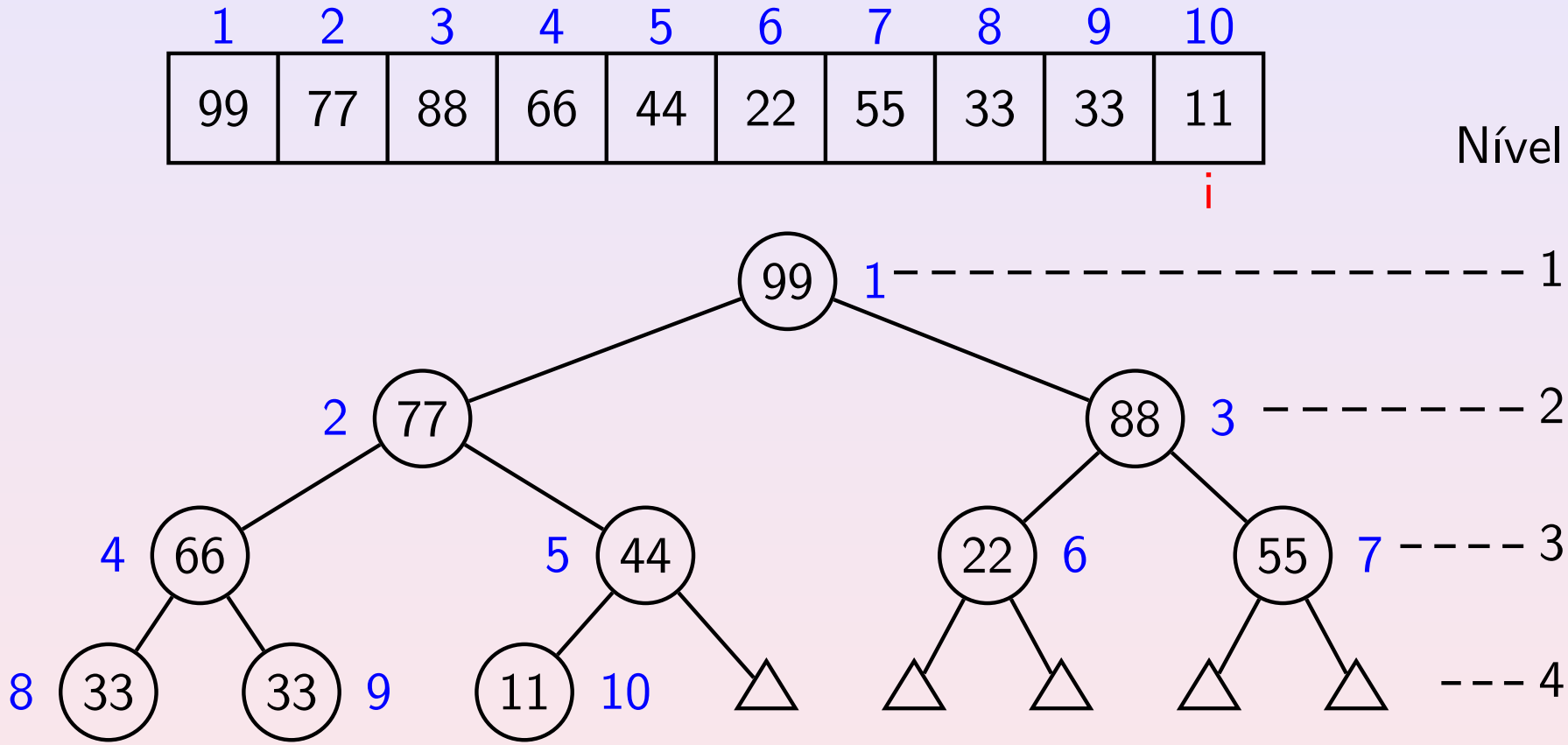
Ordenação: Build-Max-Heap

1	2	3	4	5	6	7	8	9	10
99	77	88	66	44	22	55	33	33	11

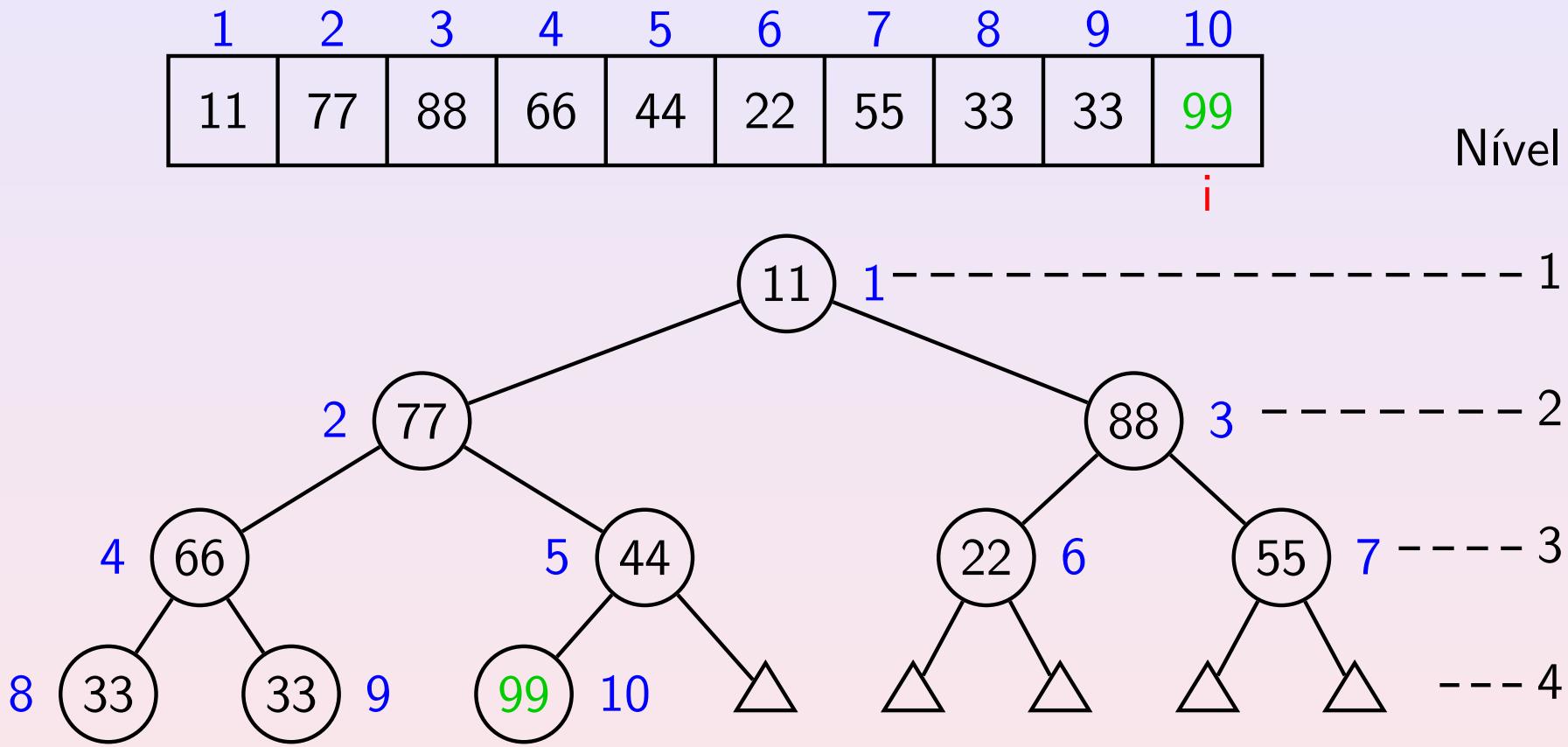


Heap Máximo atingido

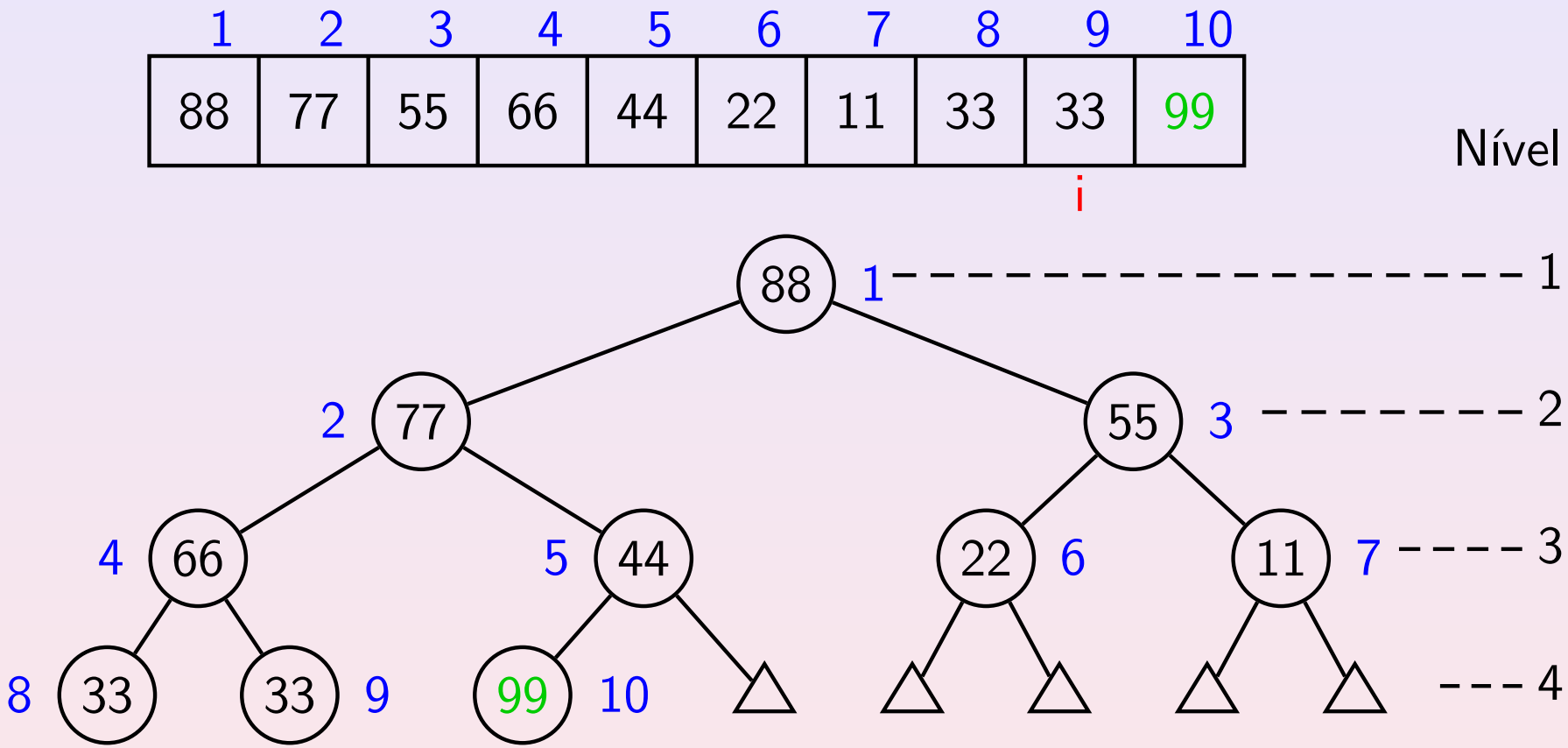
Ordenação: Troca de Elementos e Max-Heapfy



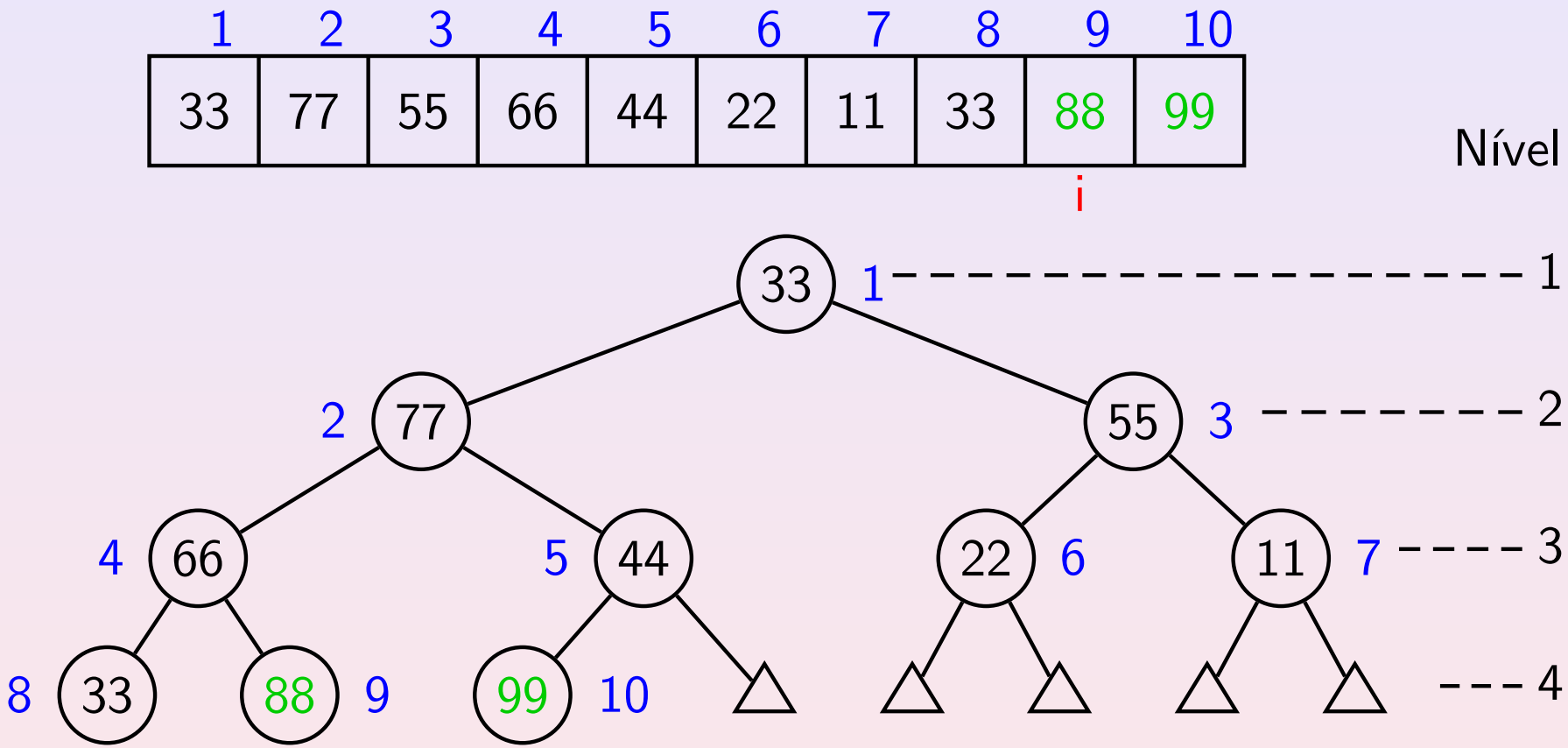
Ordenação: Troca de Elementos e Max-Heapfy



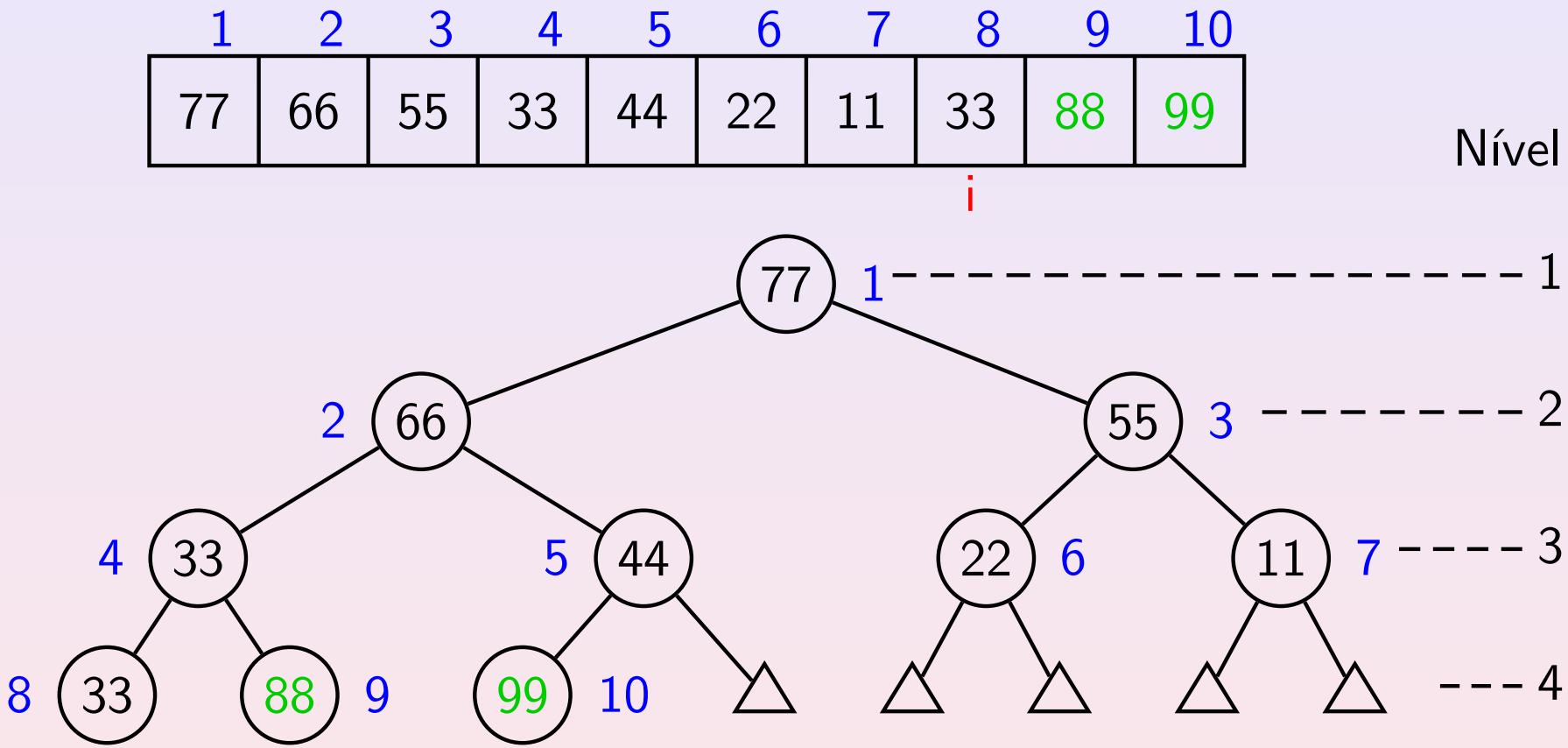
Ordenação: Troca de Elementos e Max-Heapfy



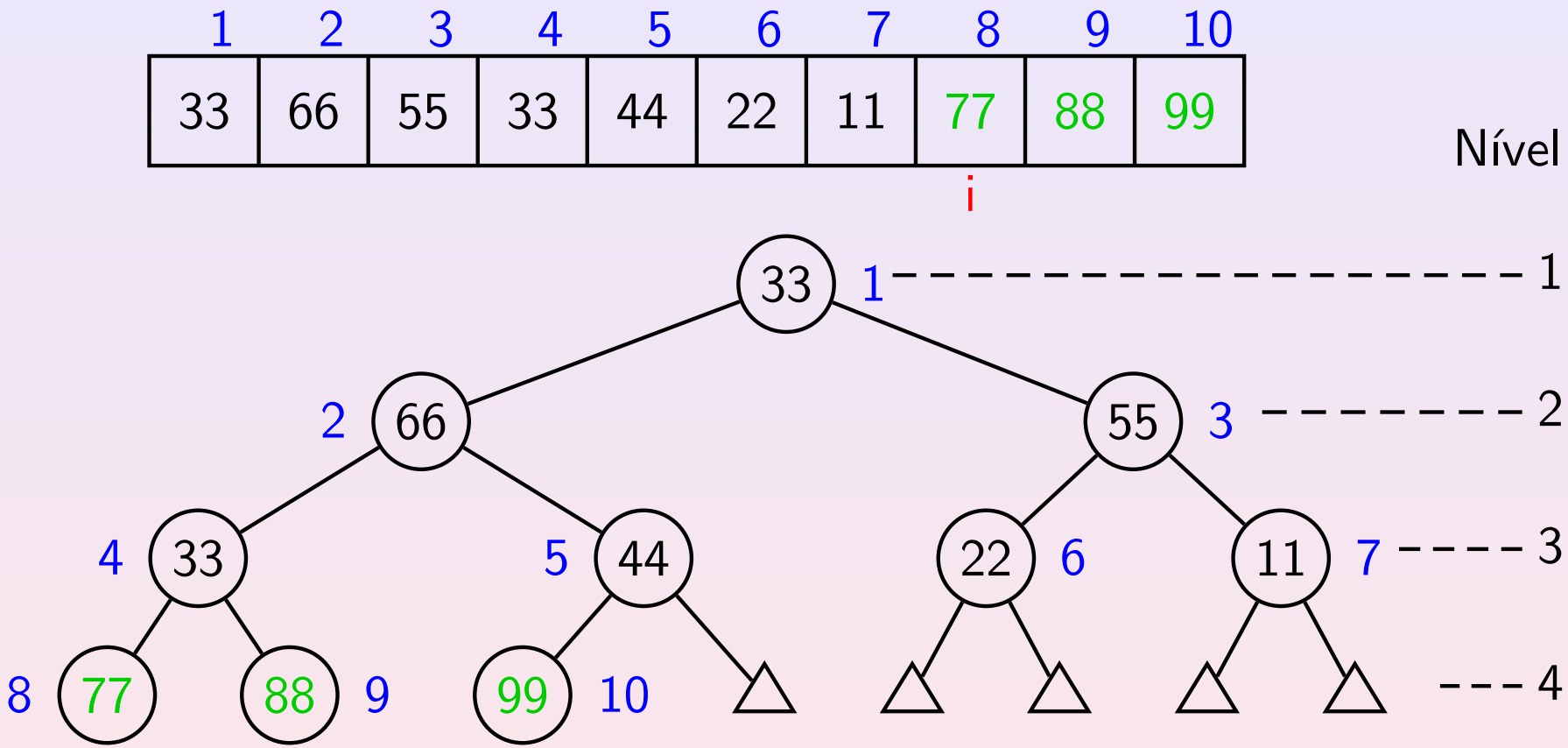
Ordenação: Troca de Elementos e Max-Heapfy



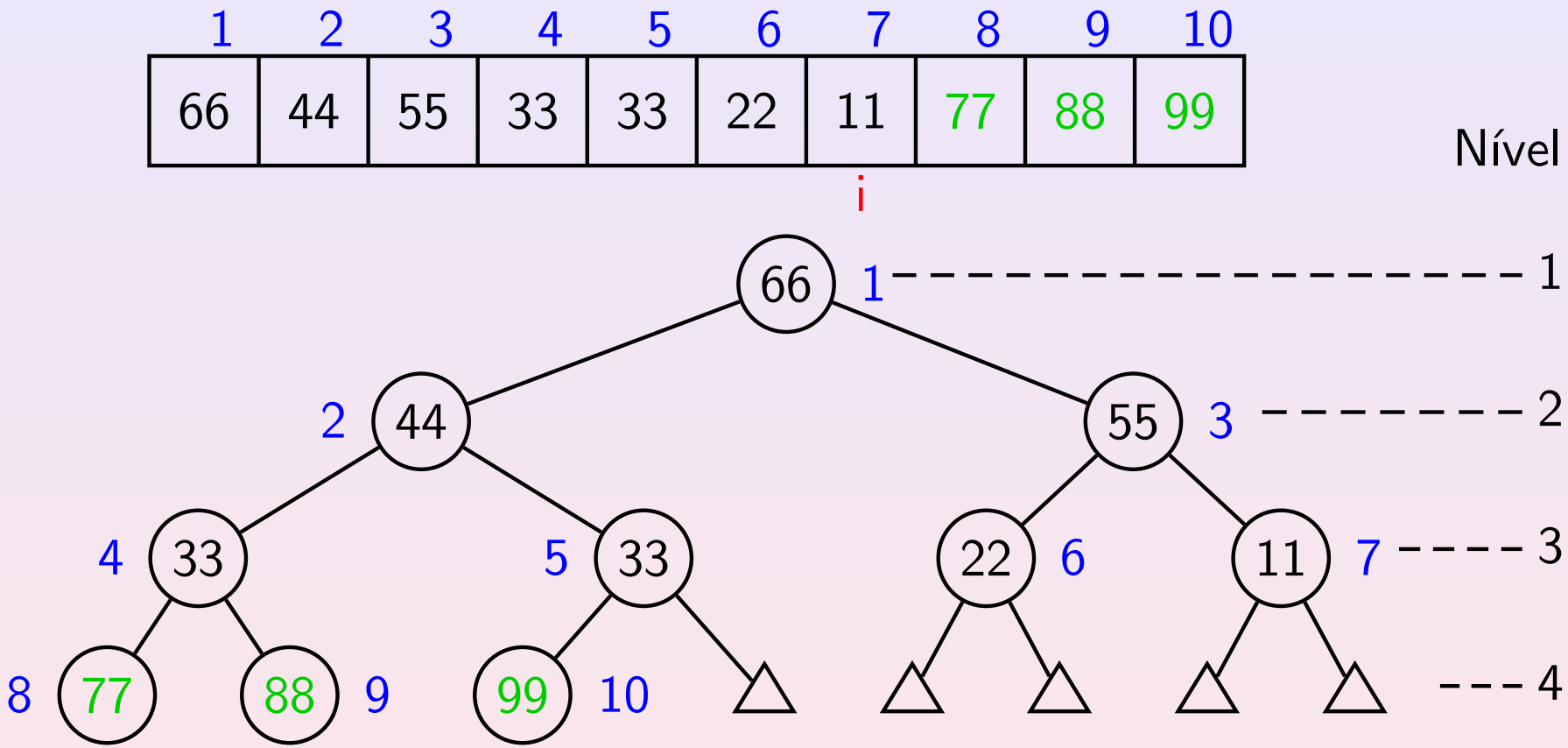
Ordenação: Troca de Elementos e Max-Heapfy



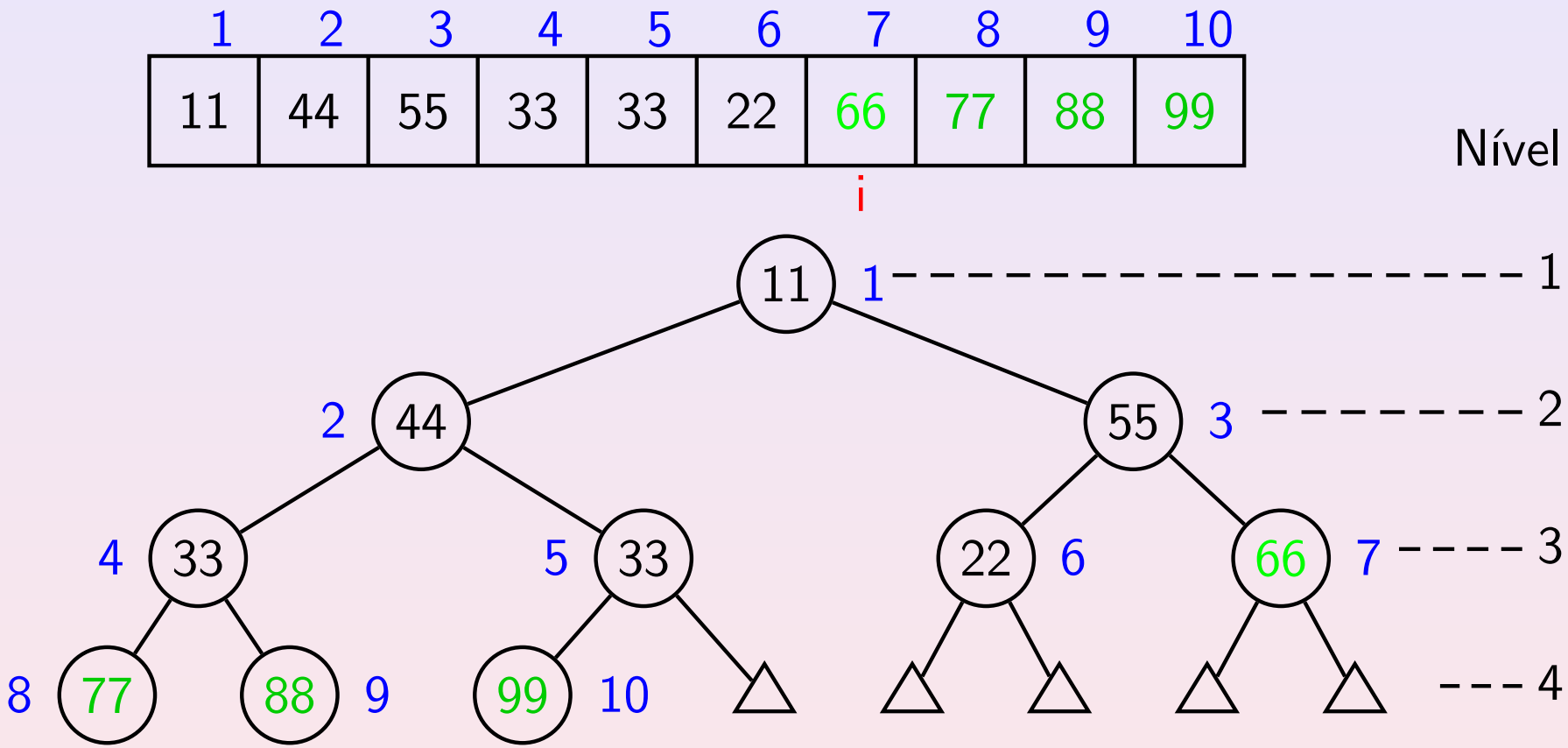
Ordenação: Troca de Elementos e Max-Heapfy



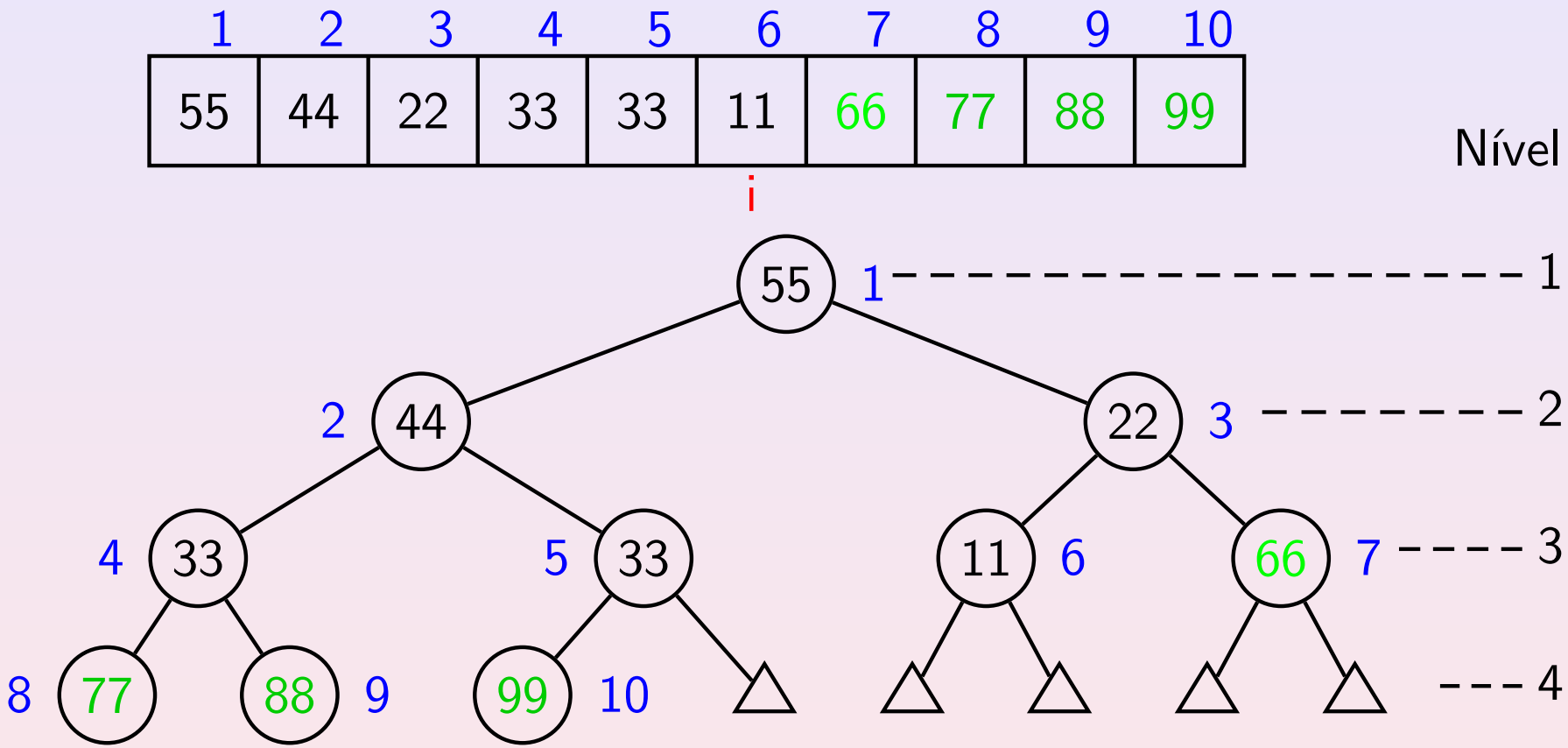
Ordenação: Troca de Elementos e Max-Heapfy



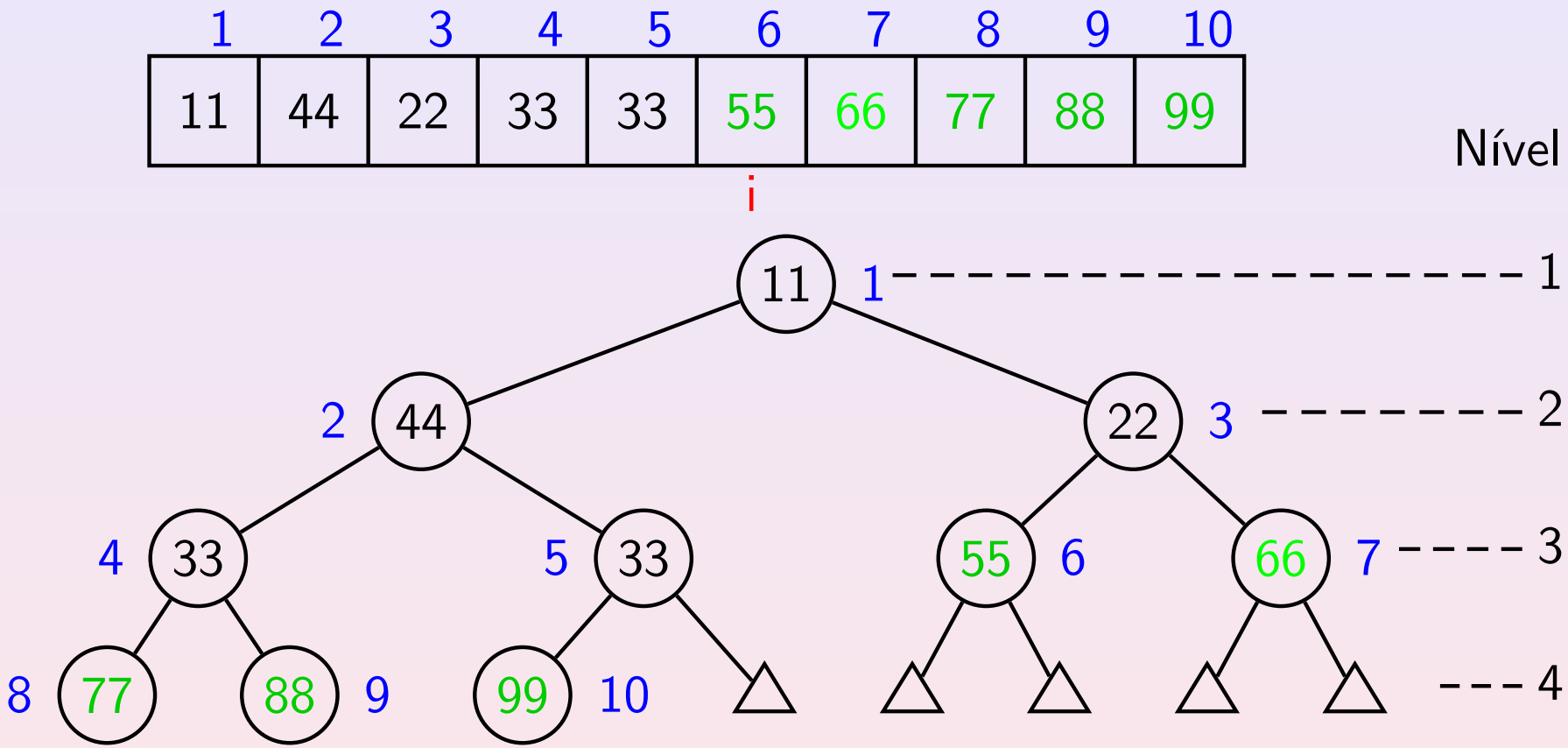
Ordenação: Troca de Elementos e Max-Heapfy



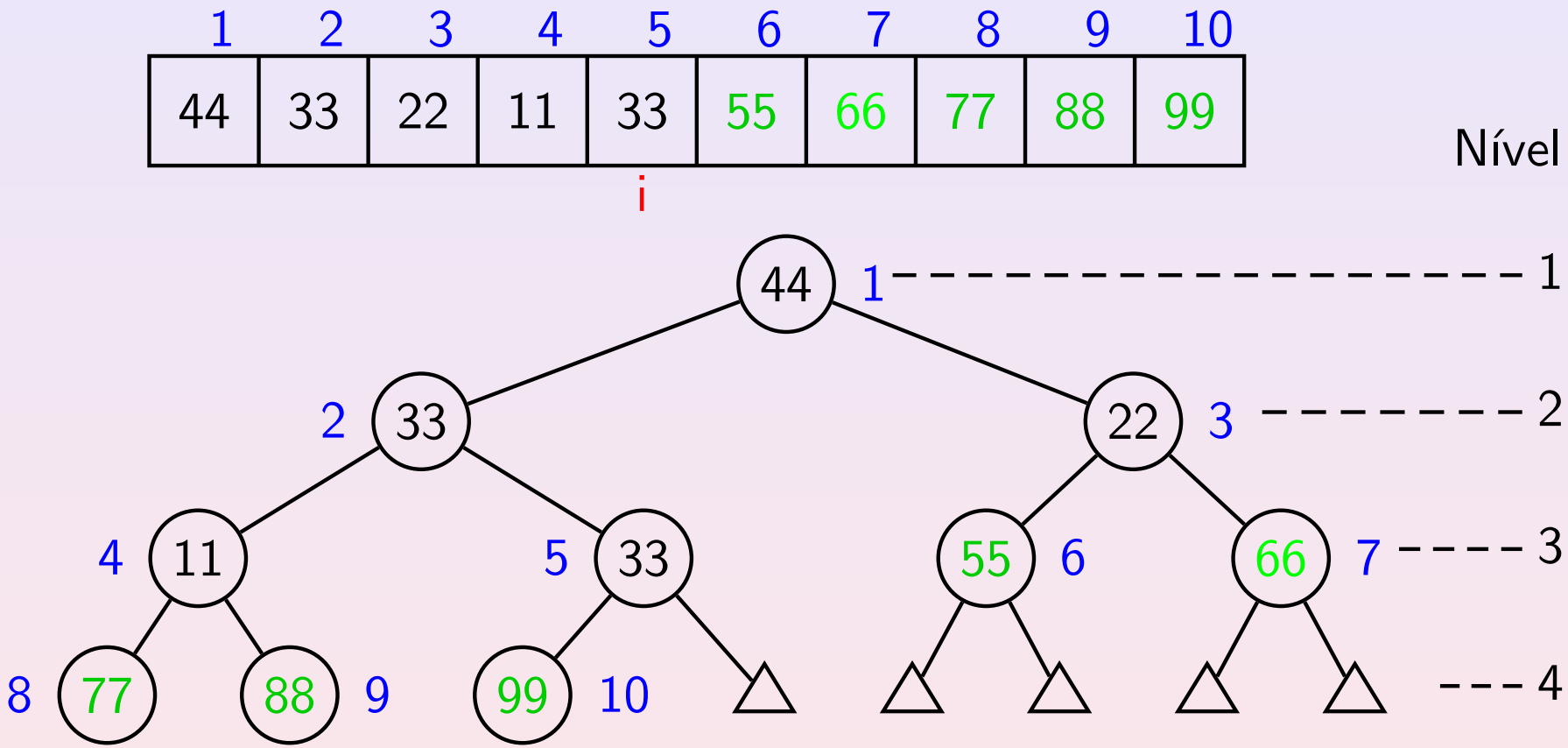
Ordenação: Troca de Elementos e Max-Heapfy



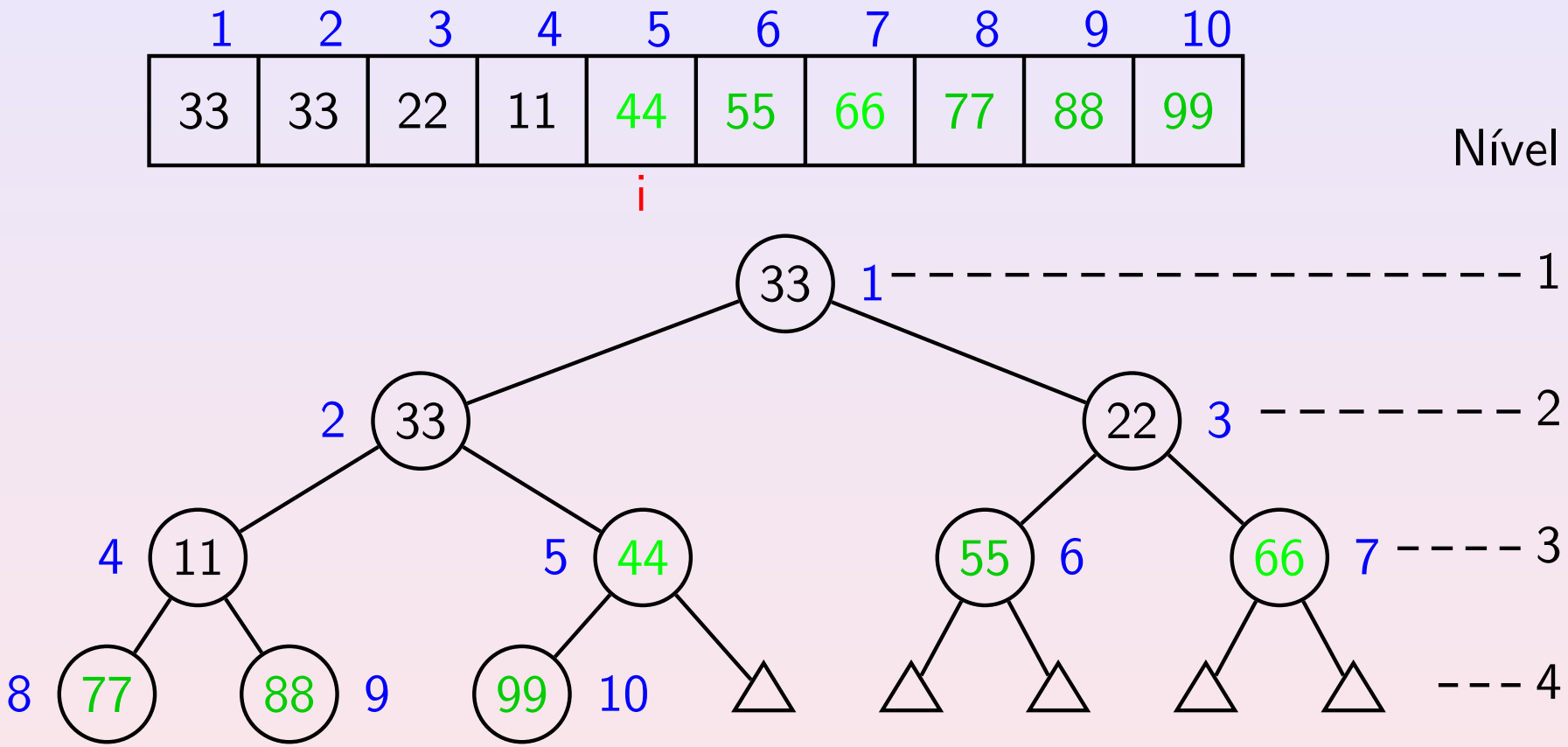
Ordenação: Troca de Elementos e Max-Heapfy



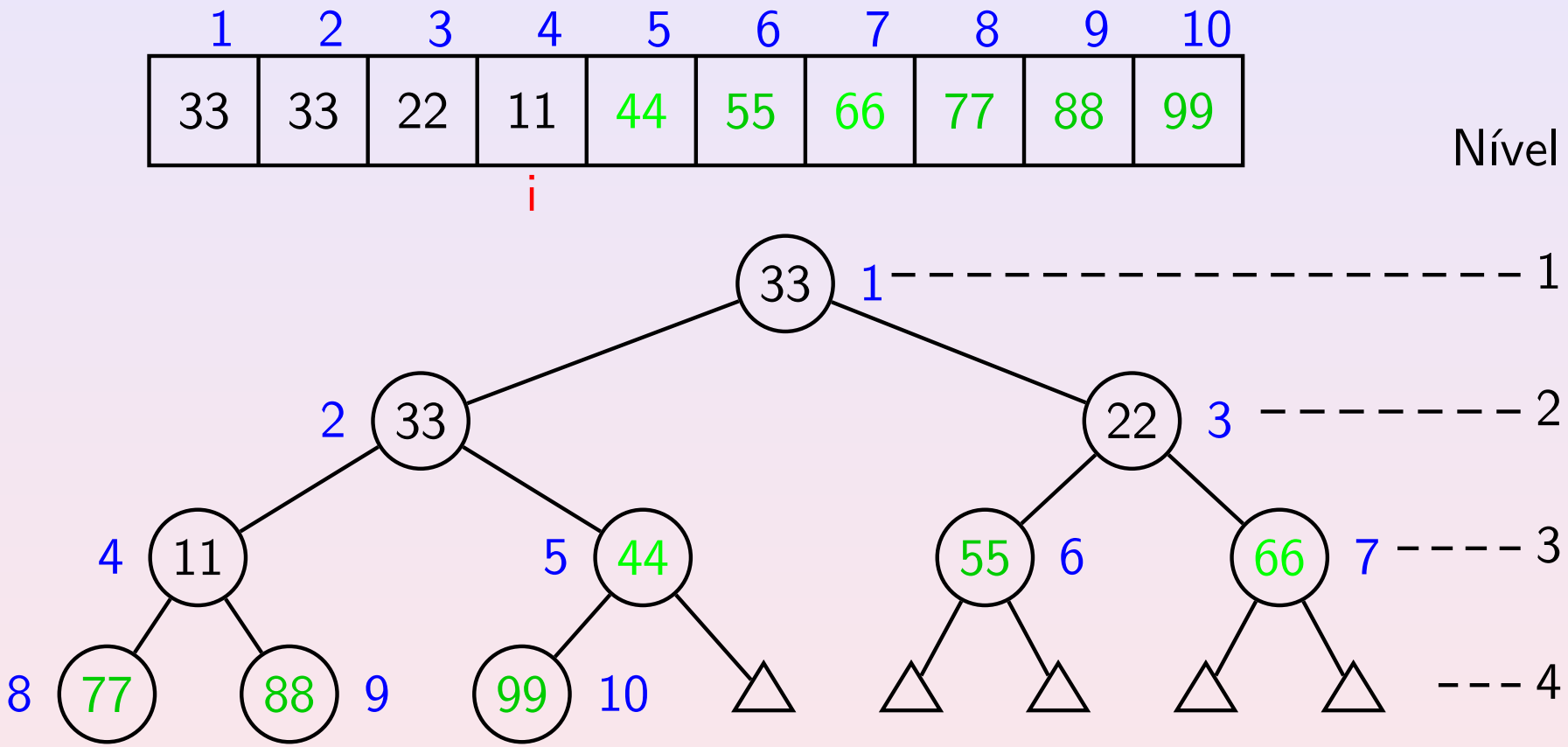
Ordenação: Troca de Elementos e Max-Heapfy



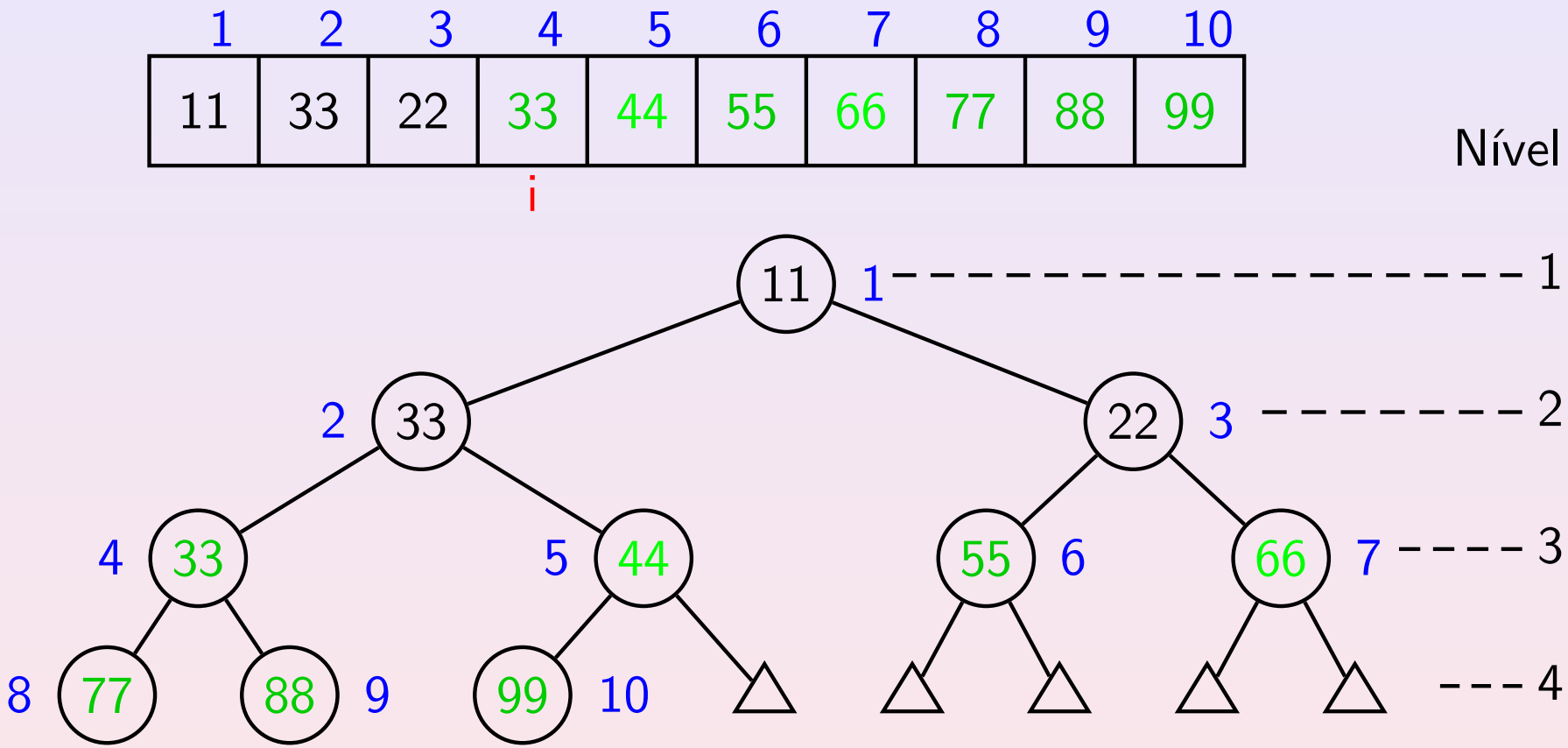
Ordenação: Troca de Elementos e Max-Heapfy



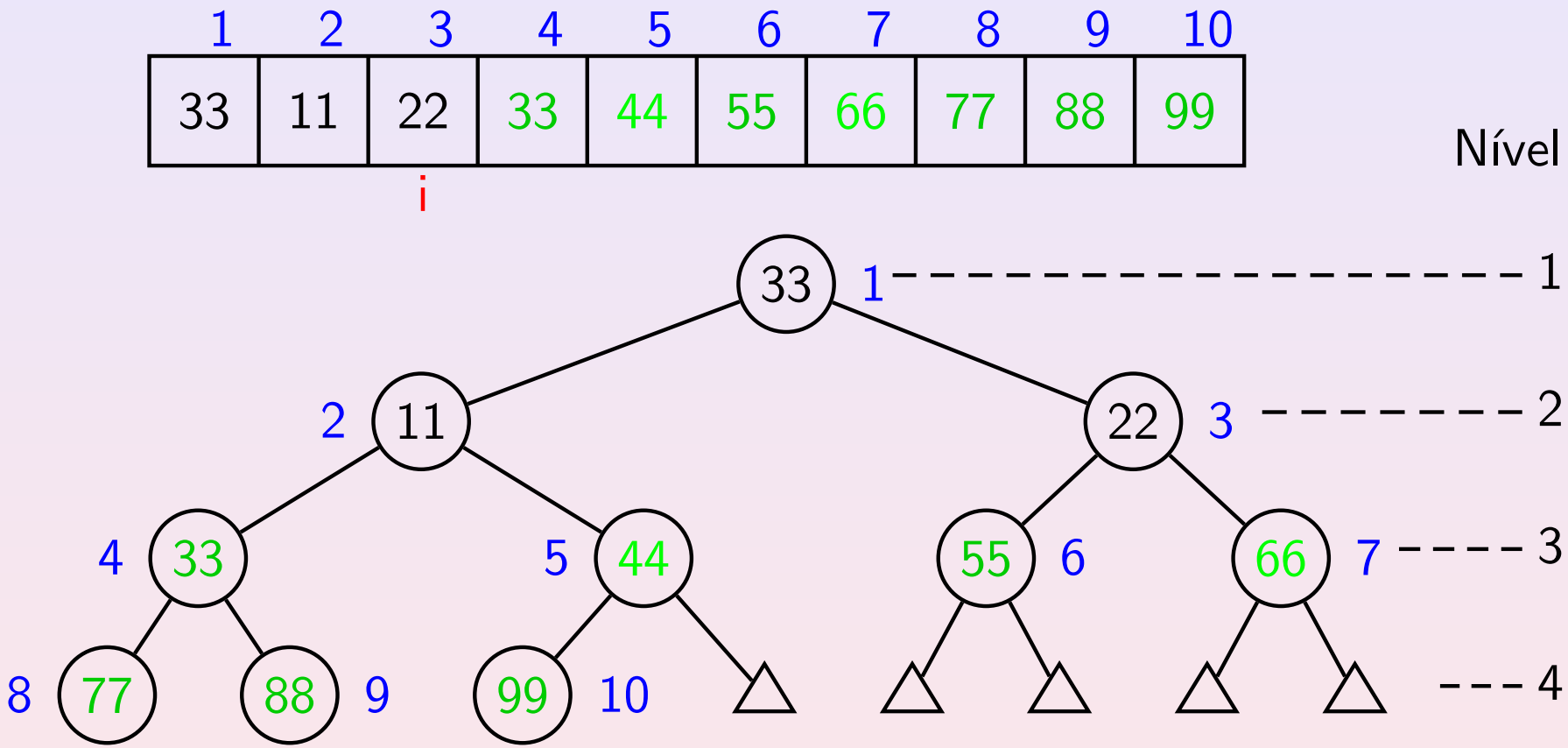
Ordenação: Troca de Elementos e Max-Heapfy



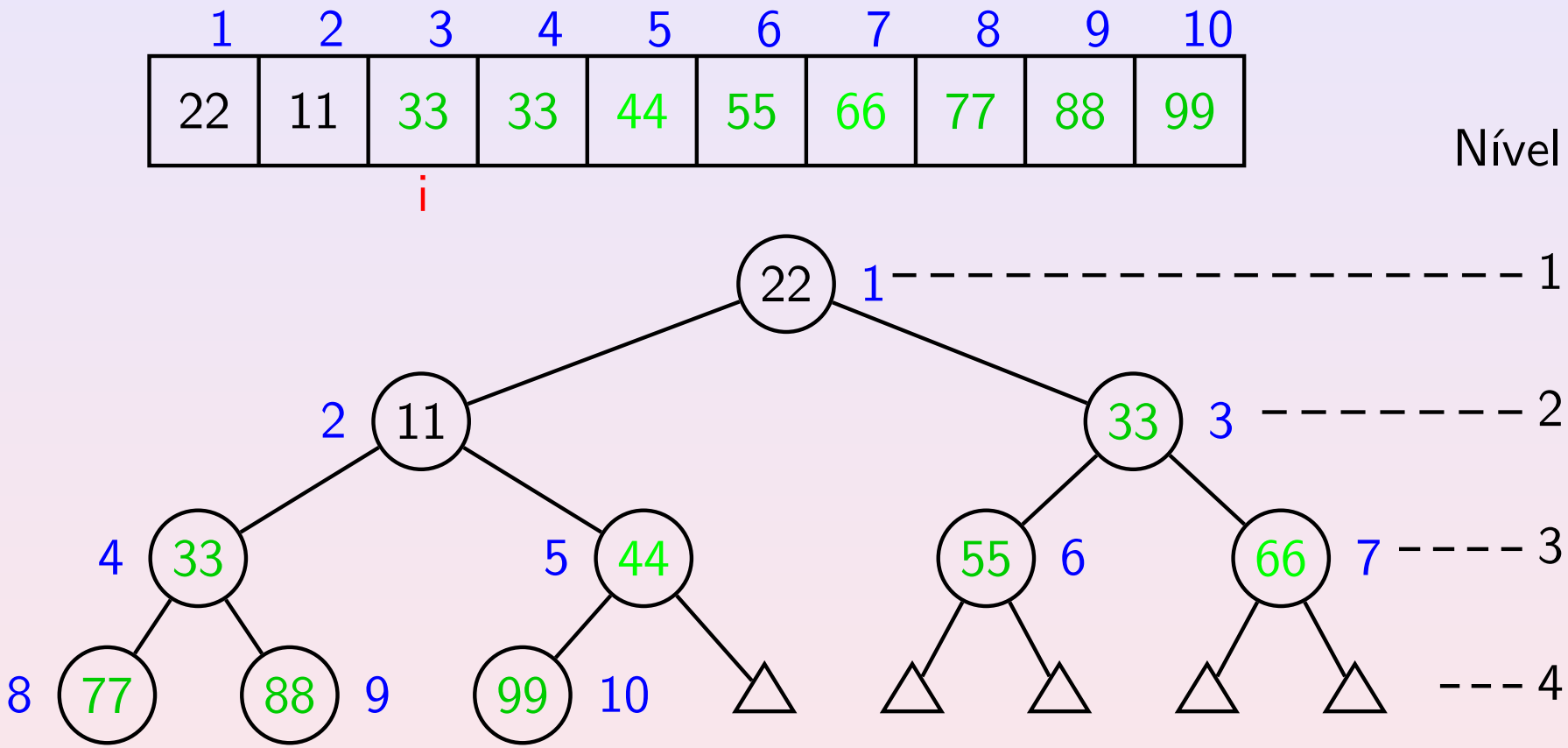
Ordenação: Troca de Elementos e Max-Heapfy



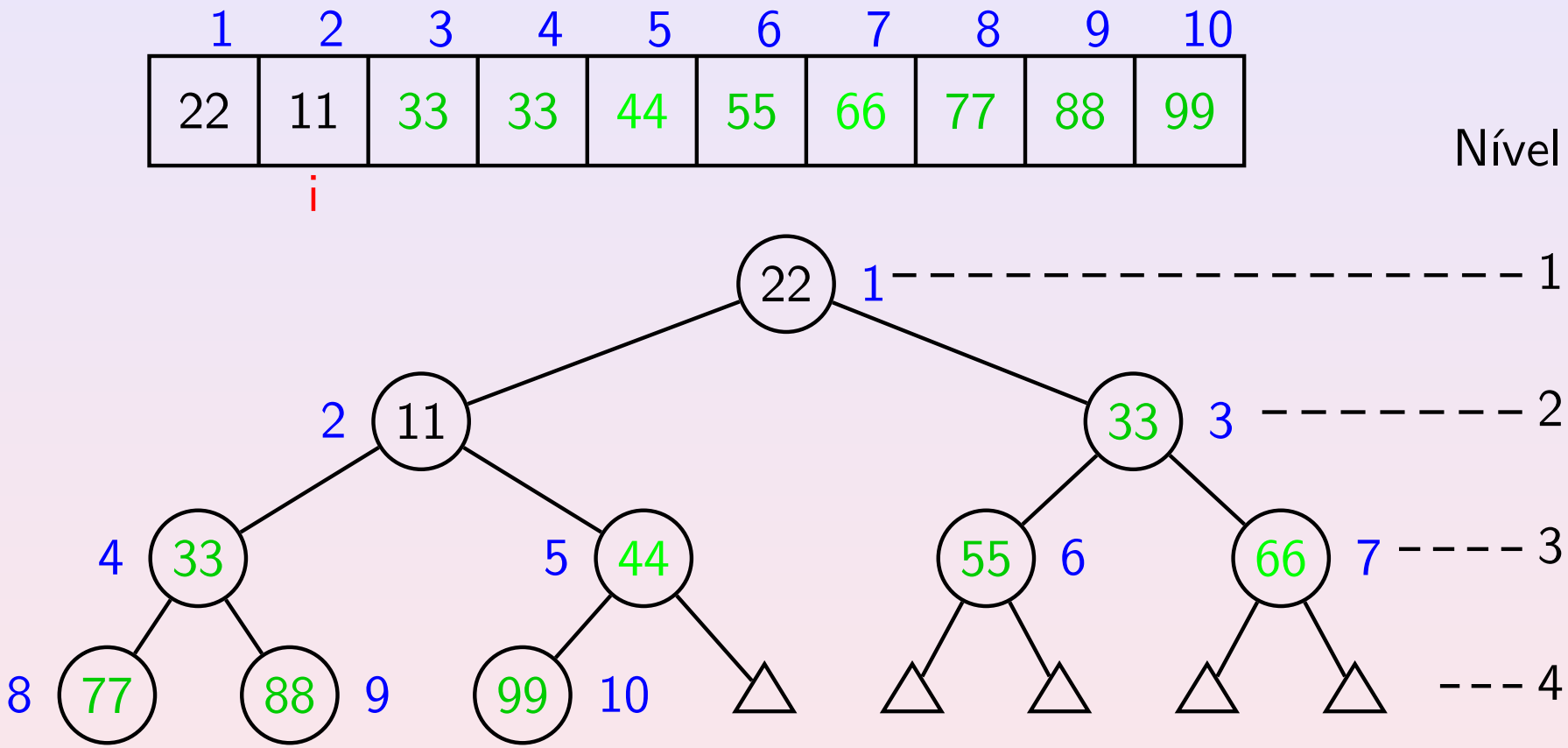
Ordenação: Troca de Elementos e Max-Heapfy



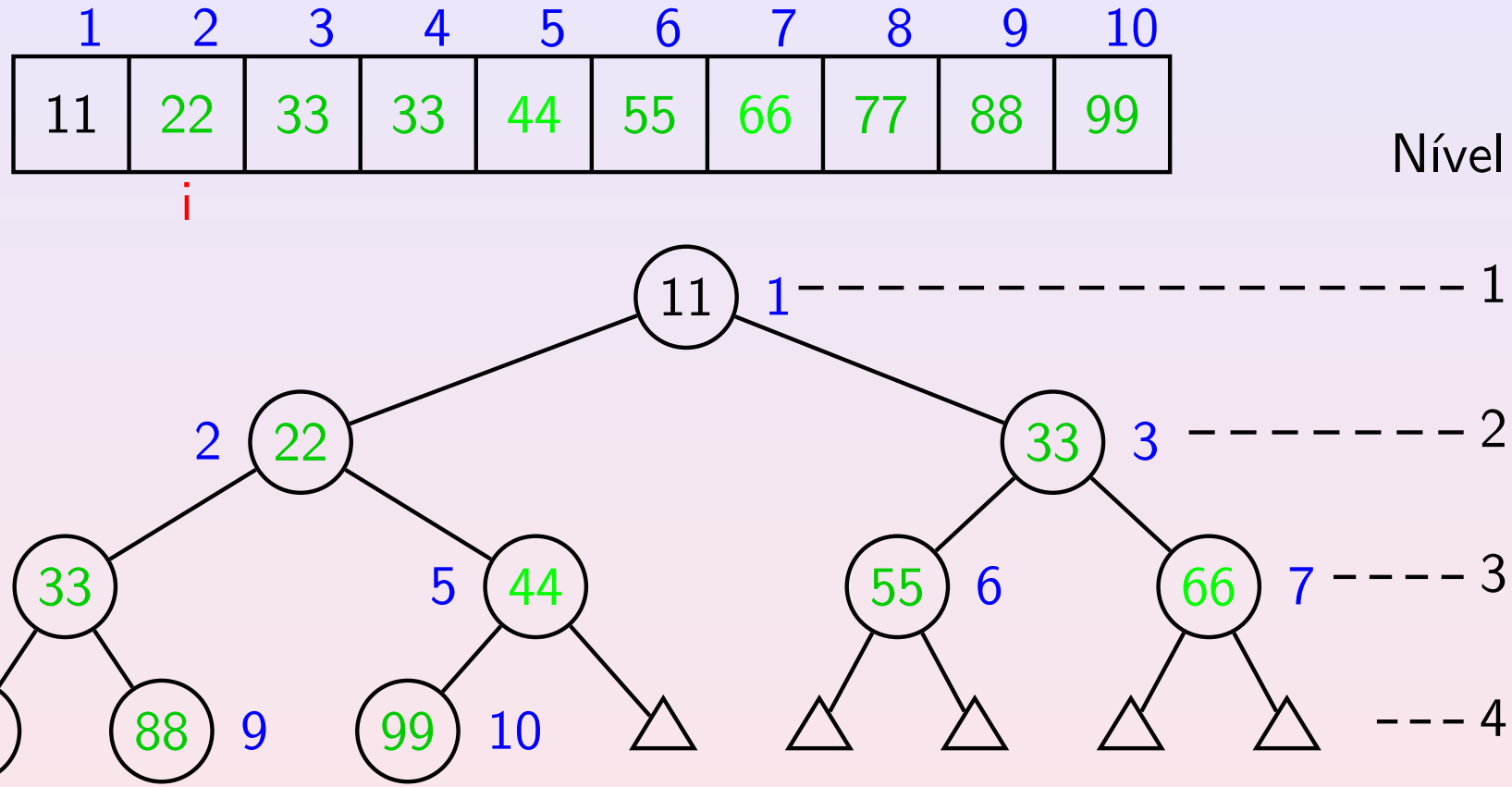
Ordenação: Troca de Elementos e Max-Heapfy



Ordenação: Troca de Elementos e Max-Heapfy

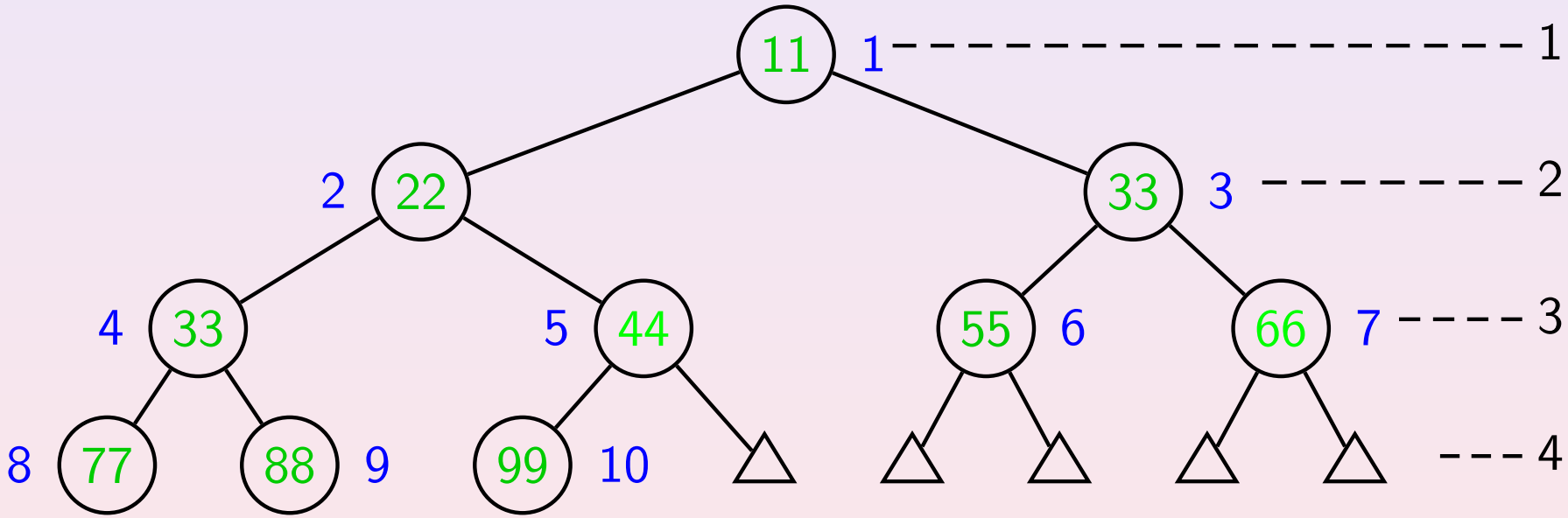


Ordenação: Troca de Elementos e Max-Heapfy



Ordenação: Troca de Elementos e Max-Heapfy

1	2	3	4	5	6	7	8	9	10
11	22	33	33	44	55	66	77	88	99



Sequência classificada em ordem crescente

Algoritmo Heap-Sort

Algoritmo HEAP_SORT(A, n)
    Build_Max_Heap(A, n)
    para $i \leftarrow n$ **até** 2 **faça**
        $A[1] \leftrightarrow A[i]$
        Max_Heapfy($A, i - 1, 1$)

Algoritmo Heap-Sort - Custo

- O algoritmo Max-Heapfy executa em $O(\log n)$, que é a altura da árvore.
- O algoritmo Build-Max-Heap possui um cálculo mais complexo de complexidade, mas é $O(n)$.
- O Algoritmo Heap-Sort aplica n vezes o Max-Heapfy, portanto é $O(n \log n)$.

Códigos de Huffman

- Representa uma técnica de compressão de dados que pode atingir valores entre 20 e 90%.
- É aplicado em arquivos de símbolos (texto) onde existe um distribuição diferenciada na freqüência com que cada símbolo aparece.
- Codifica-se os símbolos com tamanhos distintos de bits. Símbolos mais frequentes recebem menos bits, símbolos menos frequentes recebem mais bits.
- Na média o número de bits do arquivo será menor.

Exemplo do código de Huffman

- Considere o alfabeto $C = \{a, b, c, d, e, f\}$.
- Um dado arquivo possui a freqüência indicada na tabela abaixo para os caracteres do alfabeto.
- Também na tabela estão indicadas duas possíveis codificações para cada objeto, uma de tamanho fixo e outra de tamanho variável.

	a	b	c	d	e	f
Freqüência (milhares)	45	13	12	16	9	5
Código: *Tamanho Fixo*	000	001	010	011	100	101
Código: *Tamanho Variável*	0	101	100	111	1101	1100

- Qual o tamanho do arquivo para cada uma das codificações?

Calculando o custo para cada tipo de codificação

- Codificação com códigos de tamanho fixo:

$$Totaldebits = 3 \times 100.000 = 300.000 bits$$

- Codificação com códigos de tamanho variável:

$$\underbrace{1 \times 45}_{a} + \underbrace{3 \times 13}_{b} + \underbrace{3 \times 12}_{c} + \underbrace{3 \times 16}_{d} + \underbrace{4 \times 9}_{e} + \underbrace{4 \times 5}_{f} = 224.000 bits$$

- Há um ganho de aproximadamente 25% se utilizarmos a codificação de tamanho variável.

O método

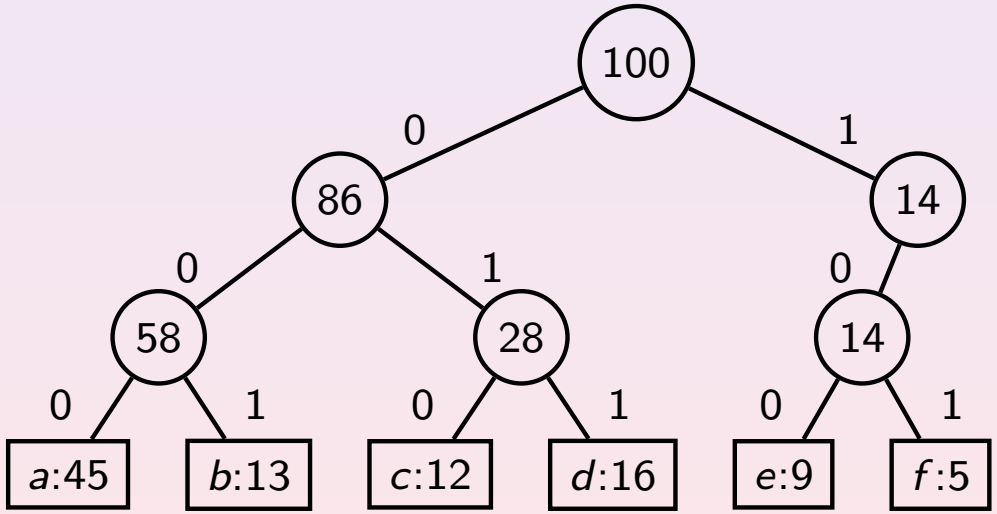
- A solução implica no uso de uma "codificação livre de prefixo".
- Em uma codificação livre de prefixo, para quaisquer símbolos distintos i e j codificados, a codificação de i não é prefixo da codificação de j.
- No exemplo anterior, usando a codificação variável para a palavra "abc" obtemos: 0101100.
  - O único caracter começado com 0 e que portanto utiliza somente um bit é o 'a';
  - A seqüência 101 define o caracter 'b' e não há qualquer outro caracter que inicie com o código 101;
  - O restante 100 representa o caracter 'c'.

Representando o código

- Precisamos identificar uma estrutura que associe um código ao caracter, de forma que na decodificação encontremos facilmente o símbolo utilizando o código fornecido.
- Uma solução é utilizar uma árvore binária:
  - Um filho esquerdo está associado a um bit `0`.
  - Um filho direito a um bit `1`.
  - Nas folhas se encontram os símbolos.
  - O código lido `0` ou `1` faz com que na navegação na árvore chegue a um símbolo.
  - Ao achar um símbolo o próximo código é aplicado a partir do raiz.

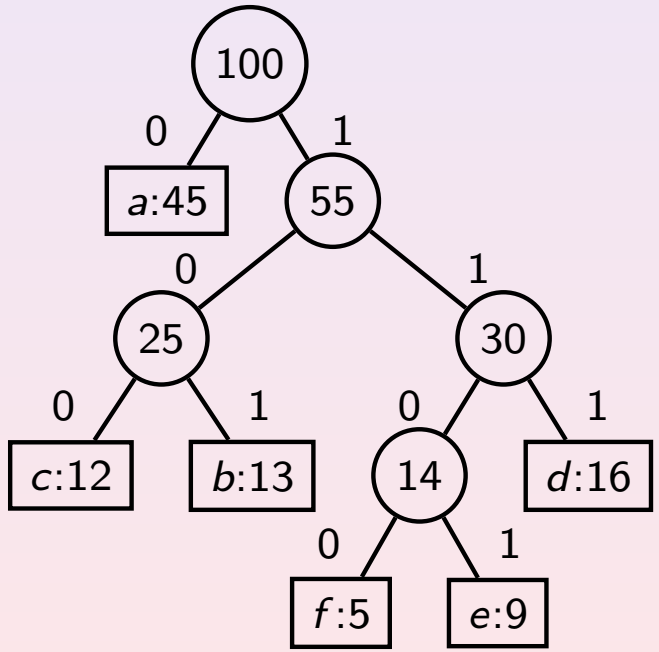
Código de tamanho fixo na forma de árvore

	a	*b*	*c*	*d*	*e*	*f*
Freqüência (milhares)	45	13	12	16	9	5
Código	000	001	010	011	100	101



Código de tamanho variável na forma de árvore

	a	b	c	d	e	f
Freqüência (milhares)	45	13	12	16	9	5
Código	0	101	100	111	1101	1100



Propriedades da árvore de código

- Cada código é livre de prefixo: Só há um único caminho para chegar a uma folha, que não passa por outra folha, assim o código de um símbolo não é um prefixo de outro símbolo.
- Uma codificação ótima deve ser representado por uma árvore binária cheia, cada vértice interno tem dois filhos. Seja uma codificação com um vértice interno que só tenha um filho:
  - Se o filho for uma folha. Podíamos colocar esta folha no lugar do vértice e economizaríamos um bit para o código deste símbolo.
  - Se o filho for outro vértice. A partir deste vértice buscamos uma folha, colocamos esta folha como segundo filho do vértice. Economizaríamos no mínimo um bit.
- Buscamos uma árvore binária cheia com $|C|$ folhas (o tamanho do alfabeto) e $|C - 1|$ vértices internos.

Entendendo a proposta de solução

- Começar com $|C|$ árvores folhas isoladas e realizar seqüencialmente $|C-1|$ operações de agregação, agregando duas árvores a um novo vértice raiz comum. O raiz passa a ter como "peso" a soma dos custos de cada árvore agregada.
- A escolha do par de árvores que serão agregadas dependerá do custo de cada árvore. As duas árvores de menor custo serão escolhidas.
- O raiz de uma árvore carrega como informação o custo da árvore.

O algoritmo de Huffman

Entrada: Conjunto de caracteres de C e a freqüências f de cada caracter

Saída: Raiz da árvore binária representando codificação ótima livre de prefixo

Algoritmo HUFFMAN(C)

$n \leftarrow |C|$

$Q \leftarrow C$ ▷ Q é fila prioriadade de árvores

para $i \leftarrow 1$ **até** $n - 1$ **faça**

$z \leftarrow$ **novo** *Arvore*

$z.esq \leftarrow Extrai\_Minimo(Q)$

$z.dir \leftarrow Extrai\_Minimo(Q)$

$z.info = z.esq.info + z.dir.info$

$Insere(Q, z)$

retorne $Extrai\_Minimo(Q)$