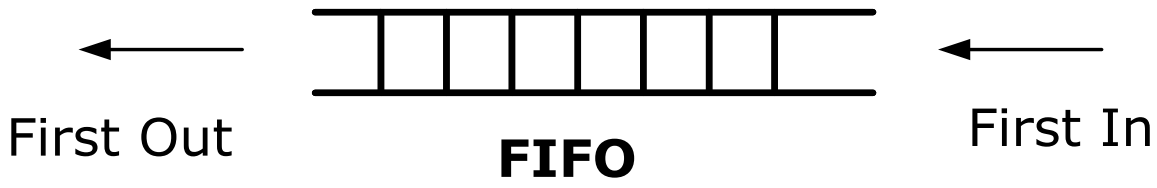
Estruturas de Informação

Fila de Prioridades

Departamento de Engenharia Informática (DEI/ISEP)
Fátima Rodrigues
mfc@isep.ipp.pt

Fila de Espera ("Queue")

- o acesso aos elementos é feito usando ambos os lados da fila
- no início da lista são retirados elementos
- no fim da lista são adicionados elementos
- os elementos permanecem na fila na mesma ordem em que chegaram, obtém-se uma ordenação do tipo "FIFO - First In, First Out"



Operações:

- Adicionar um elemento (push)
- Remover um elemento (pop)
- Qual o elemento que está à frente ? (front)
- Qual o comprimento ? (size)
- Está vazia ? (empty)

Fila de Prioridades (Priority Queue)

Uma Fila de Prioridades é um contentor com as mesmas funcionalidades que uma Fila de Espera, mas com as seguintes diferenças:

- cada elemento da fila tem uma prioridade associada
- o único elemento que pode ser acedido ou retirado da fila é o que tem maior prioridade

Operações:

- Adicionar um elemento
- Remover um elemento
- Qual o elemento com maior prioridade (frente) ?
- Qual o comprimento ?
- Está vazia ?

Aplicações

Para que serve uma Fila de Prioridades ?

- Fila de espera numa urgência de hospital

  a prioridade corresponde ao grau de gravidade do doente

- Ordenação de processos a executar pelo processador de um computador

  certos processos, por exemplo de sistema, têm maior prioridade de execução do que outros, independentemente da sua ordem de chegada

- Simulação baseada em eventos

- Ordem de aterragem de aviões num aeroporto

  aviões militares ou em dificuldades podem receber uma prioridade superior na ordem de aterragem

Implementação

- A implementação de uma Fila de Prioridades pode ser feita de diversas formas, usando várias estruturas de dados

- Se considerarmos uma implementação com uma estrutura de dados sequencial, há duas abordagens distintas:
  - os elementos da fila estão sempre ordenados
    - implica trabalho de ordenação na operação Enqueue, mas as operações Dequeue e Front são imediatas;

      (Ex: Array Ordenado)

  - os elementos não estão ordenados
    - a operação Enqueue é imediata, mas as operações Dequeue e Front implicam pesquisar o elemento e eventualmente reorganizar a estrutura

      (Ex: Array, Lista Ligada)

Complexidade

	Array	**Array ordenado**	**Lista Ligada**
enqueue	O(1)	$O(n^2)$	O(1)
dequeue	$O(n^2)$	O(1)	$O(n^2)$
front	$O(n^2)$	O(1)	$O(n^2)$

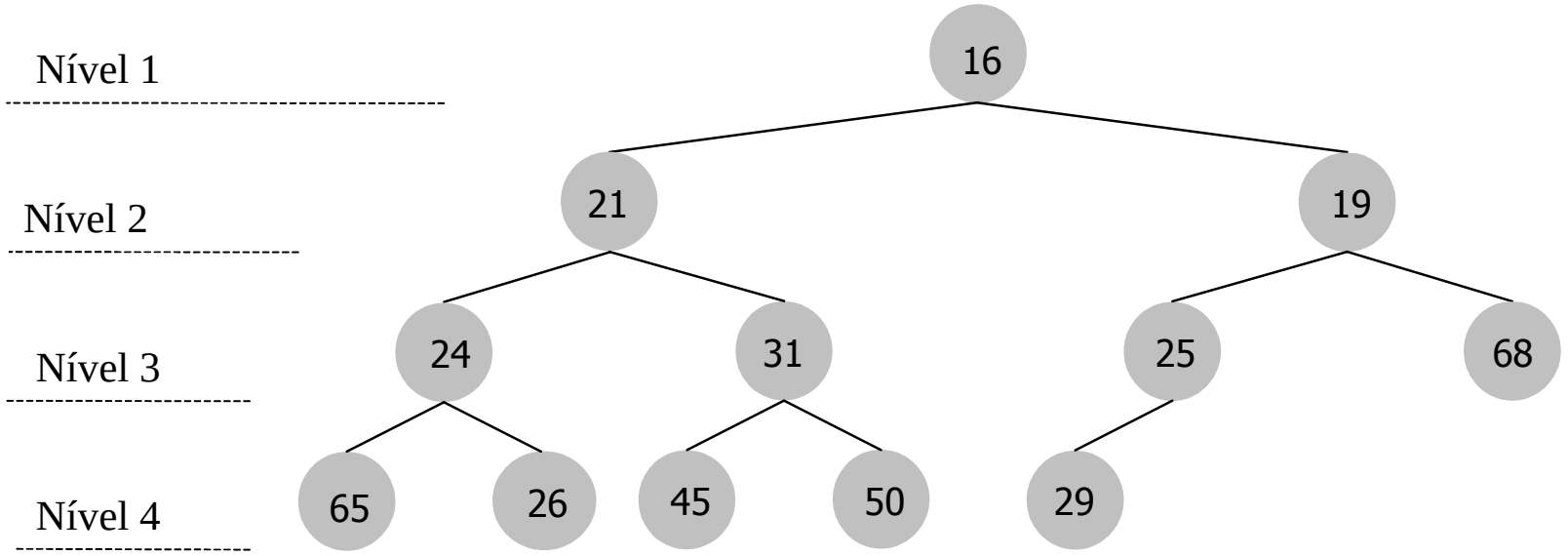
Qualquer implementação numa estrutura de dados sequencial implica operações de complexidade quadrática

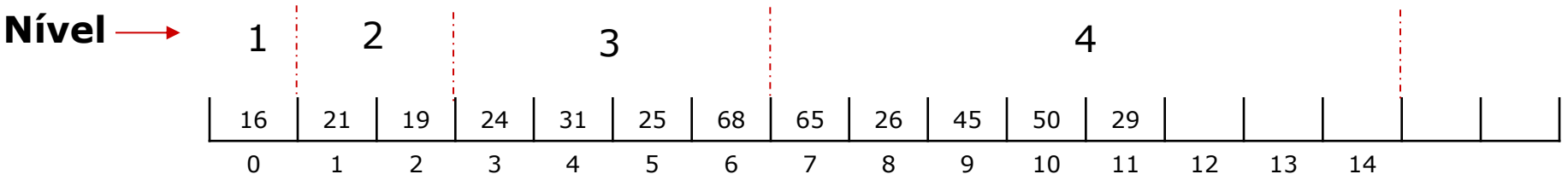
Alternativa → Heap

Árvores Binárias em Vectores

Uma Árvore Binária pode ser representada por um vector:

Preenche-se o vector com os elementos da árvore ordenados por nível:





Árvores Binárias em Vectores

Como aceder aos elementos ?

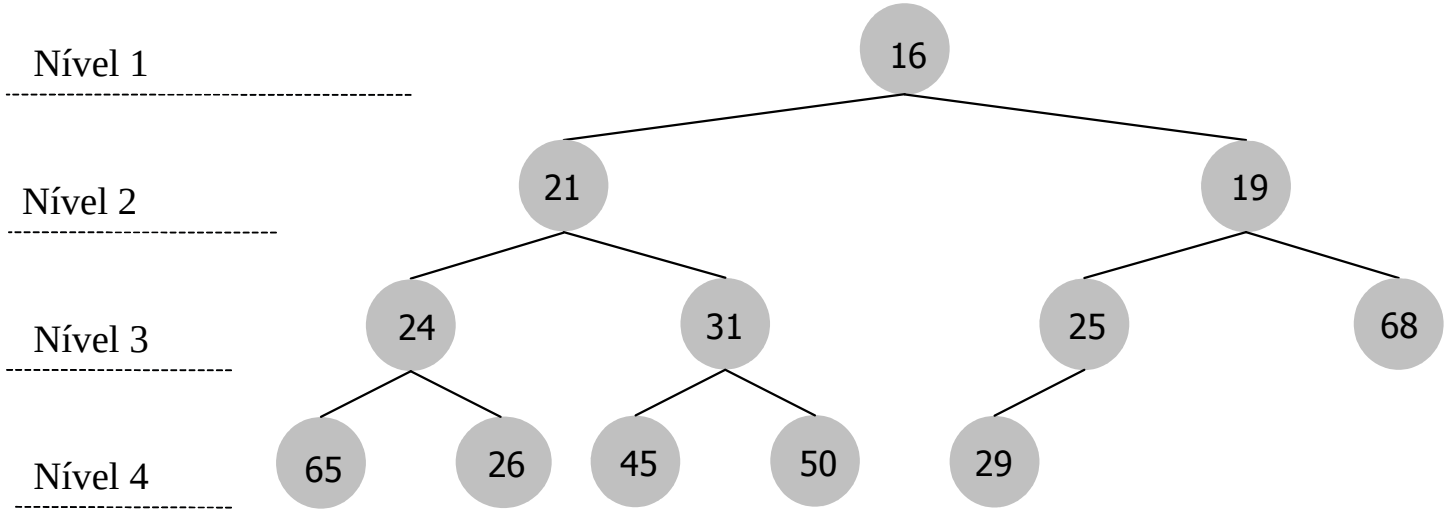
- Estando a raiz no índice 0
- Os seus descendentes directos estão nos índices 1 e 2
- Os descendentes de 1 estão nos índices 3 e 4
- ....

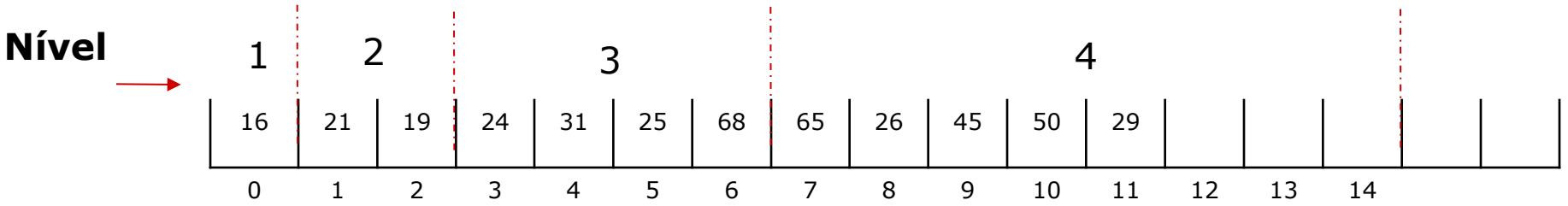
Em geral para um nó no índice n

- Os seus descendentes estão nos índices: 2n + 1 (filho esquerdo)
  2n + 2 (filho direito)
- O seu pai está no índice: (n-1)/2

Estas fórmulas permitem transitar entre os elementos dos diferentes níveis da árvore, usando uma forma alternativa ao uso de apontadores entre nós

Árvores Binárias em Vectores





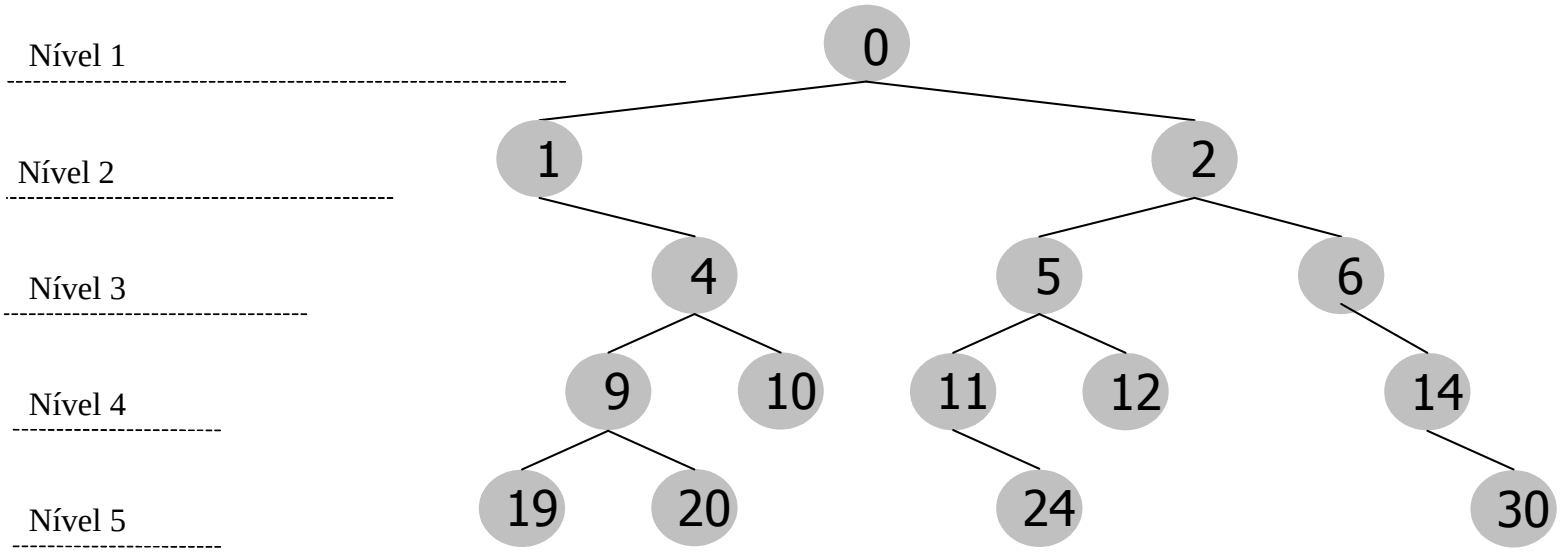
Antecessores do nó 50 (até à raiz):

50 → n=10 → (n - 1)/2 = 4 → vect[4] = 31

n= 4 → (n - 1)/2 = 1 → vect[1] = 21

n= 1 → (n - 1)/2 = 0 → vect[0] = 16

Árvores Binárias em Vectores



0	1	2		4	5	6			9	10	11	12		14					19	20		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14								...

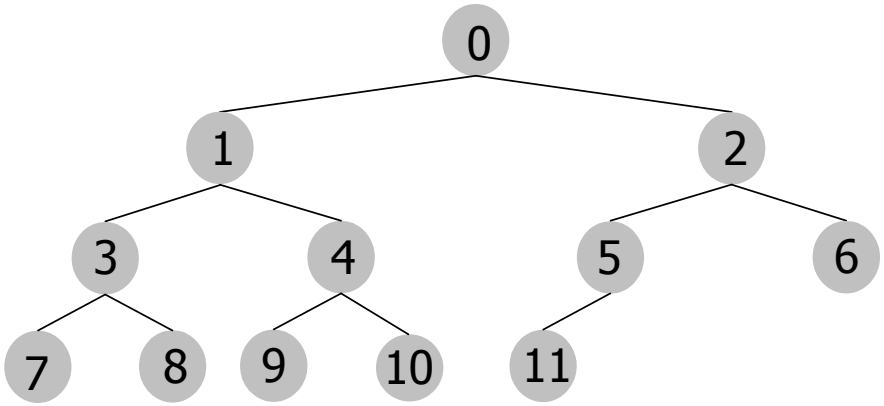
Problema da representação em vector:

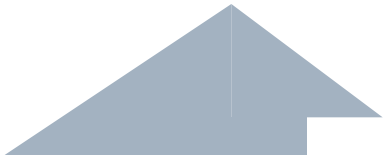
- Visto que cada nó potencial tem uma posição fixa no vector, esta abordagem só é eficiente se a árvore estiver cheia
- Senão desperdiçam-se muitas posições vazias no vector

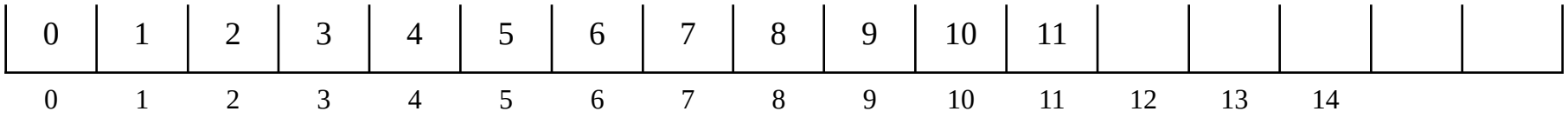
Árvores Binárias Completas

Definição

Uma árvore binária diz-se **completa** se estiver completamente cheia, com a possível excepção do seu último nível, que deve estar preenchido da esquerda para a direita





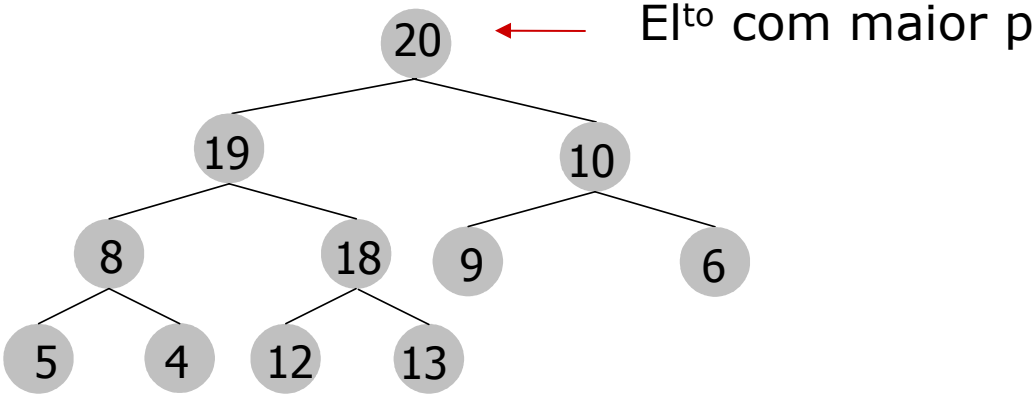


Relação com vector:
Uma árvore completa pode ser eficientemente armazenda num vector, pois o vector correspondente não contém elementos vazios, todos os elementos utilizados encontram-se contíguos

Heap

É uma **árvore binária completa** com uma propriedade de ordenação específica. Para cada nó N:

- todos os nós **descendentes de N** são **menores** (ou iguais) do que N relativamente à sua prioridade
- todos os nós **ascendentes de N** são **maiores** (ou iguais) do que N
- por consequência todos os nós são menores do que a raiz, **em prioridade**



Naturalmente, as operações de inserção e remoção na árvore devem preservar a propriedade de ordenação da Heap

Heap – Implementação em Vector

Uma Heap é uma **árvore binária completa**

Isto faz com que uma Heap possa aproveitar a implementação de árvores em vector de forma eficiente

Definitivamente:

uma Fila de Prioridades é implementada com uma Heap em Array

Atenção:

O factor de ordenação (de comparação entre elementos) deve ser a prioridade de cada elemento

é necessário implementar os operadores de comparação com base na prioridade

um elemento é maior/menor do que outro se a sua prioridade for maior/menor que a do primeiro

Relação entre Heap e Fila de Prioridades

A Heap é uma implementação ideal para as Filas de Prioridades

As operações que interessam estudar na Heap são:

- inserção, para a operação enqueue
- remoção do maior elemento (o que tem maior prioridade) para a operação dequeue
- pesquisa do maior elemento (o que tem maior prioridade) para a operação front

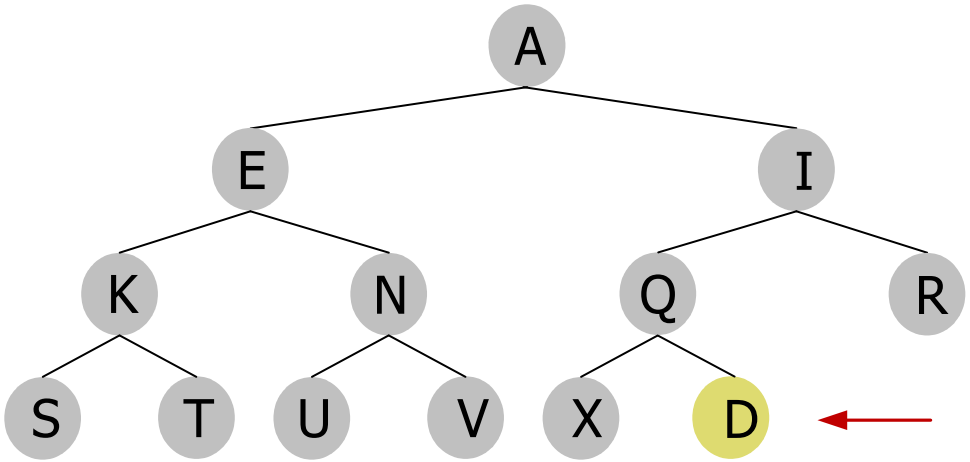
Heap – Inserção

Quando se insere ou remove um elemento numa Heap, existem duas propriedades que devem ser garantidas:

1. a árvore continua completa
2. a árvore mantém-se ordenada

Portanto, opera-se em duas fases:

para respeitar a propriedade 1. começa-se por inserir o novo elemento no último nível da árvore o mais à direita possível

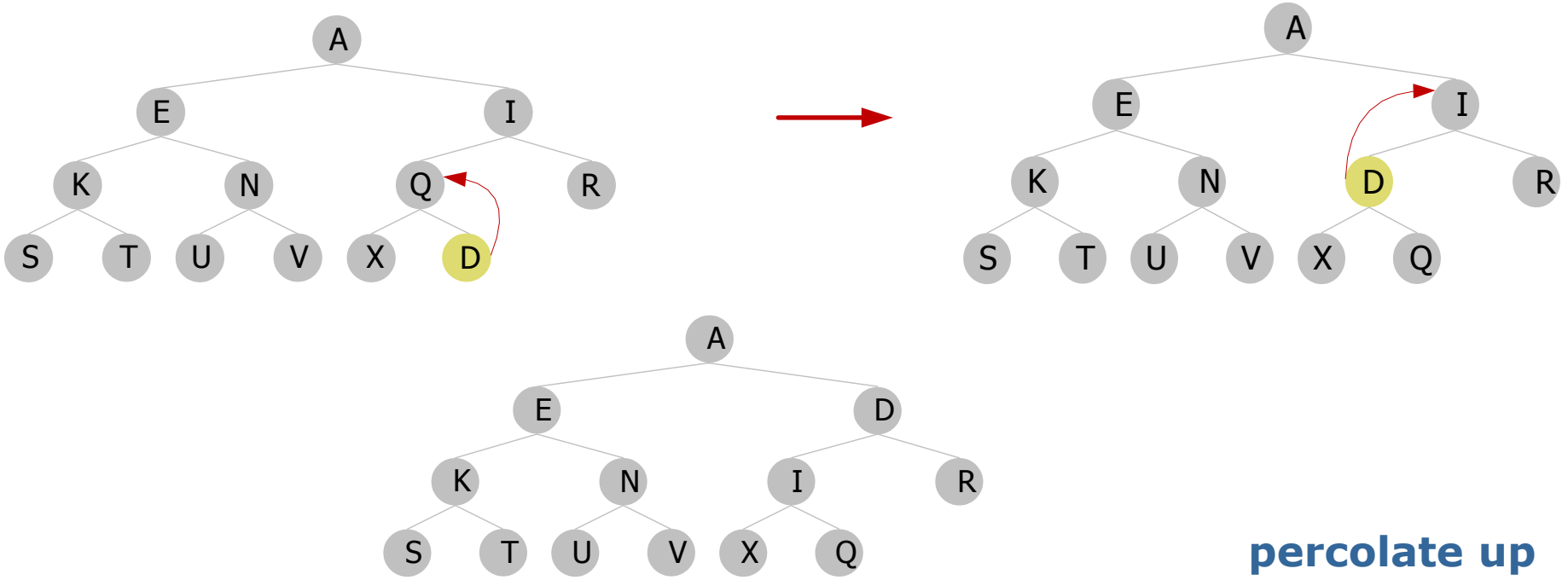


Neste exemplo, considera-se ordenação alfabética inversa

Heap – Inserção

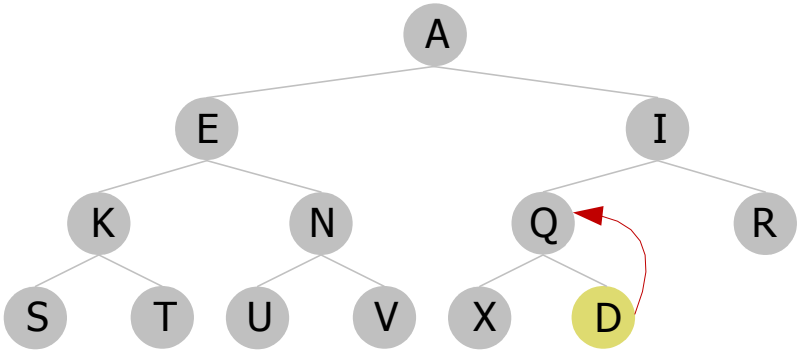
seguidamente, procura-se respeitar a propriedade 2. fazendo com que a árvore passe a estar ordenada:

- verifica-se se o novo elemento tem maior prioridade que o seu pai, e caso isso se verifique, **troca-se um pelo outro**
- repete-se o procedimento com o novo pai...
- ... e assim sucessivamente até não ser necessário subir mais o novo elemento (até ele estar ordenado)

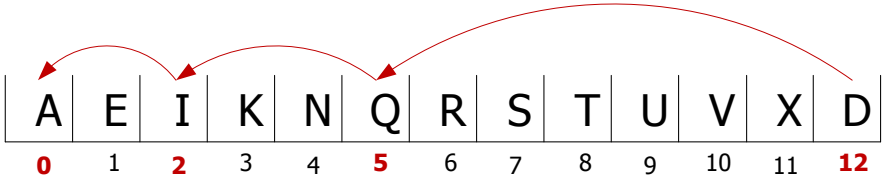


percolate up

Pseudo-código "*Percolate up*"



Para um nó no índice n
o seu pai → (n-1)/2



***Percolate_up* (elem)**

ind = vector.size( ) – 1 ;

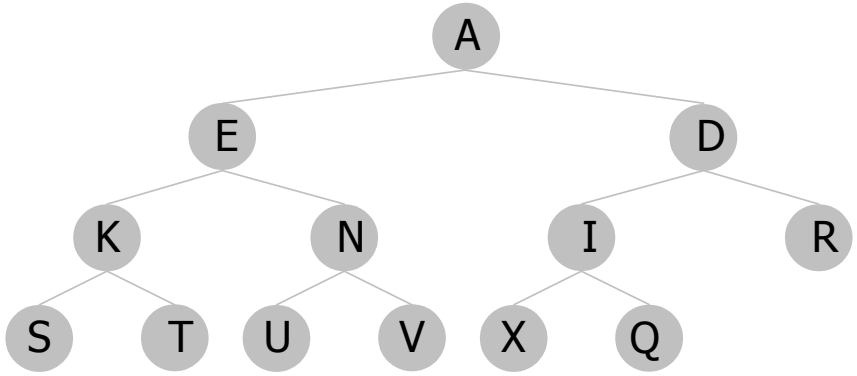
Enq^to (vector [ind] < vector[(ind-1)/2])

vector [ind] ⇆ vector [(ind-1)/2]

ind = (ind-1)/2

FEnq^to

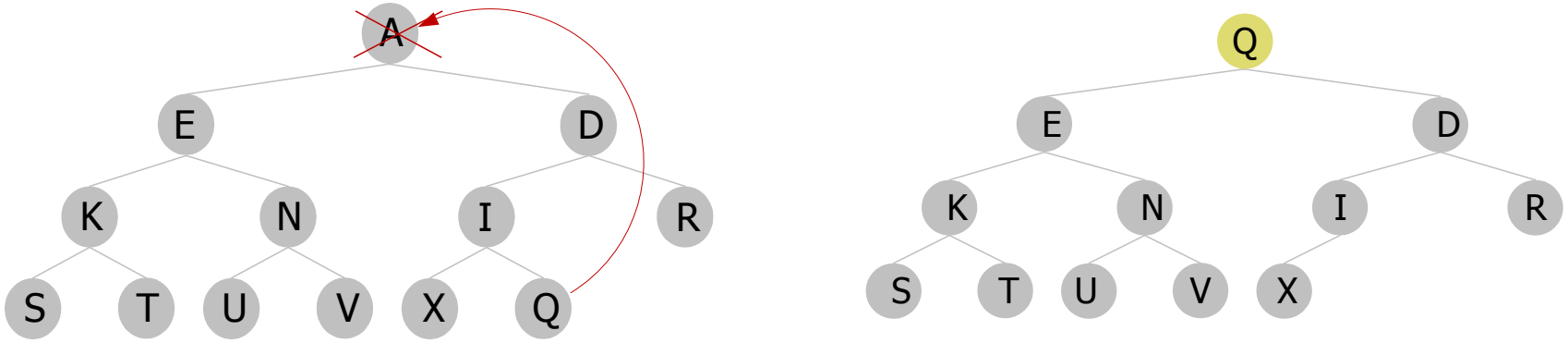
Fim Percolate_up



Heap – Remoção da Raiz

Para remover um elemento da raiz usa-se um princípio similar ao da inserção:

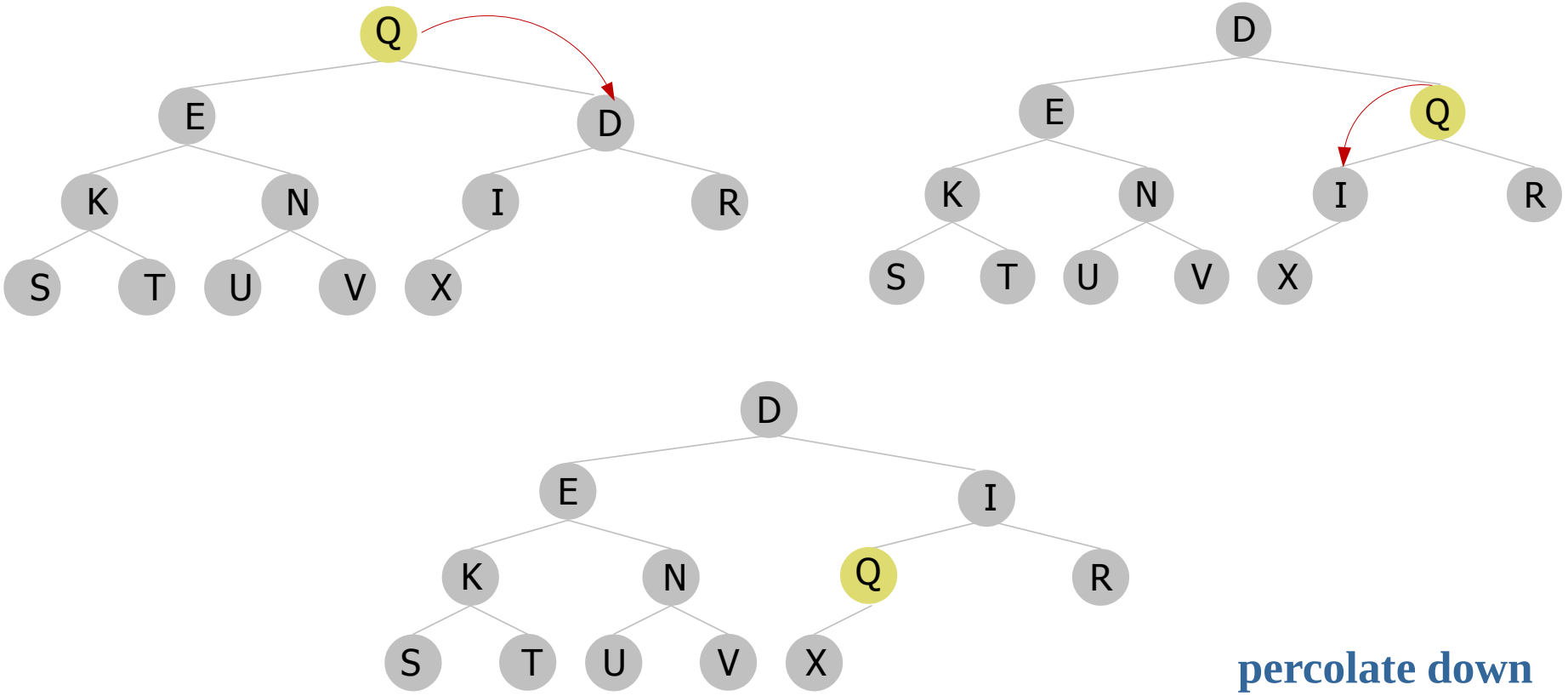
- para respeitar a propriedade 1. quando se remove a raiz, coloca-se lá o elemento que está o mais à direita no último nível da árvore



- seguidamente, procura-se respeitar a propriedade 2. fazendo com que a árvore passe a estar ordenada

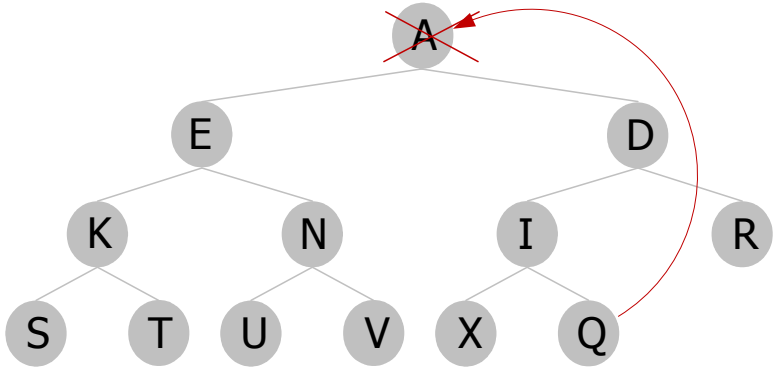
Heap – Remoção

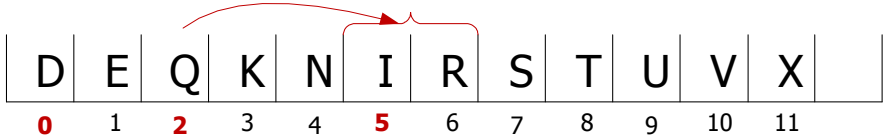
- verifica-se se o elemento da raiz tem menor prioridade do que os seus filhos (pelo menos de um deles), e caso isso se verifique, **troca-se pelo filho de maior prioridade**
- repete-se sucessivamente o procedimento até não ser necessário descer mais o elemento (até ele estar ordenado)

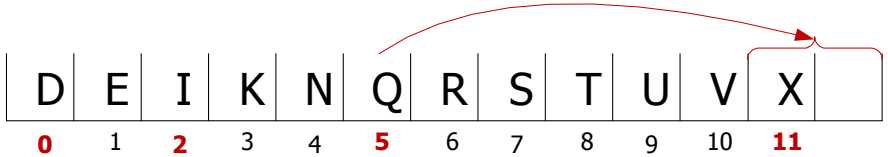


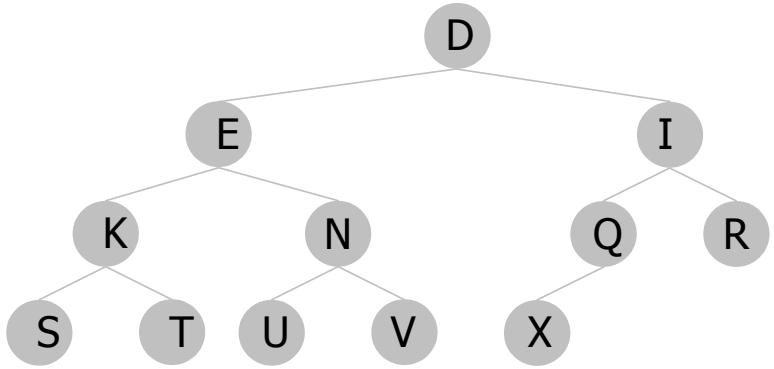
percolate down

"Percolate down"









Pseudo-código "*Percolate down*"

```
Percolate_down  ( )
    tam = vector.size()
    i = 0
    iesq = 2 × i + 1
     idir = 2 × i + 2
    Enqto ( idir < tam  ∧  (vector[i] < vector[iesq] ∨ vector[i] < vector[idir]) )
          Se (vector[iesq] < vector[idir] )
              imaior = idir
          Else
              imaior = iesq

          vector[i] ⇆ vector [imaior]        //troca o elto pelo filho com maior prioridade

           i = imaior
           iesq = 2× i + 1
            idir = 2× i + 2
      FEnqto
Fim Percolate_down
```

Implementação – Classe Heap

```
template <class T>
class heap
{
   private:
        vector<T> hv;

        void percolate_up();
        void percolate_down(int i);

   public:
        heap();
        heap(const vector<T> &v);
        heap(const heap &h);
        ~heap() ;

        void push(const T &el);
        T pop();
        T top() const;

        bool empty() const;
        void heapsort (const vector<T> &v) ;
        void write(ostream &out) const;
};
```

Implementação – Classe Heap

```
template <class T>
void heap<T>::push(const  T& el) {
        hv.push_back(el);
        percolate_up();
}

template <class T>
T heap<T>::pop() {
        T elem = hv.front() ;
        hv[0] = hv.back();
        hv.pop_back();
        percolate_down(0);
        return elem ;
}
```

Complexidade

As operações de manipulação numa Heap têm,

➥ no máximo, **complexidade logarítmica**

no máximo, o número de comparações realizadas, são as mesmas que a altura da árvore

Para além disso, o acesso ao maior elemento é imediato

Portanto uma Heap constitui uma implementação ideal para uma Fila de Prioridades

	Heap	**Array**	**Array ordenado**	**Lista Ligada**
push	O(log n)	O(1)	O(n)	O(1)
pop	O(log n)	O(n)	O(1)	O(n)
front	O(1)	O(n)	O(1)	O(n)

Algoritmo de Ordenação HeapSort

O mecanismo da Heap permite ordenar estruturas sequenciais:

Procedimento óbvio para ordenar um vetor:

- inserem-se (push) todos os elementos do vetor numa Heap
- em seguida, fazem-se sucessivas extrações do máximo (pop), da Heap de novo para o vetor
  - ↪ Isto faz com que os elementos sejam inseridos no vetor de forma ordenada

Algoritmo HeapSort

```
template <class T>
void heap<T>::heapsort (vector <T> &v)
{
   for (int i = 0 ; i < v.size() ; i++)
      push(v[i]) ;

   for (int i = 0 ; i < v.size() ; i++)
       v[i] = pop() ;
}
```

1| 28 | 27 | 5 | 26 | 26

28| 26 | 27 | 1 | 5 | 26

28| 27 | 26 | 26 | 5 | 1

Algoritmo de Ordenação HeapSort

O algoritmo HeapSort em termos de complexidade é um algoritmo concorrente relativamente aos algoritmos estudados

Algoritmo	Pior caso	Melhor Caso
Selection	$O(n^2)$	O(1)
Bubble	$O(n^2)$	O(n)
Insertion	$O(n^2)$	O(n)
MergeSort	O(n log n)	O(n log n)
QuickSort	$O(n^2)$	O(n log n)
HeapSort	O(n log n)	O(n log n)

Make heap da STL

```
int v[] = { 2, 15, 7, 1, 14, 6, 10, 8, 3, 9, 13, 5, 16, 4, 11 };

int main()
{
   vector <int> hv(begin(v),end(v));

   cout << "Make heap da STL " << endl ;
   make_heap(hv.begin(), hv.end()); // STL library equivalent

   for (auto el : hv) cout << el << " " ;
   cout << endl;
}
```

```
> Make heap da STL
> 16 15 14 13 11 10 9 8 7 6 5 4 3 2 1
>
```