FCT/Unesp – Presidente Prudente
Departamento de Matemática e Computação

Análise de Algoritmos de Ordenação Parte 3

Prof. Danilo Medeiros Eler
danilo.eler@unesp.br

Apresentação adaptada (ver referências)

Departamento de Matemática e Computação
Faculdade de Ciências e Tecnologia
Unesp – Universidade Estadual Paulista
Presidente Prudente/SP, Brasil

DEPARTAMENTO DE MATEMÁTICA E COMPUTAÇÃO

Ordenação por Seleção

Ordenação por Seleção

- Idéia básica: os elementos são selecionados e dispostos em suas posições corretas
  - Seleção direta (ou simples), ou classificação de deslocamento descendente
  - Heap-sort, ou método do monte

Seleção Direta

- Método
  1. Selecionar o elemento que apresenta o menor valor
  2. Trocar o elemento de lugar com o primeiro elemento da seqüência, x[0]
  3. Repetir as operações 1 e 2, envolvendo agora apenas os n-1 elementos restantes, depois os n-2 elementos, etc., até restar somente um elemento, o maior deles

Seleção Direta

x = 44 , 55 , 12 , 42 , 94 , 18 , 06 , 67

(vetor original)	44	55	12	42	94	18	06	67
passo 1 (06)	06	55	12	42	94	18	44	67
passo 2 (12)	06	12	55	42	94	18	44	67
passo 3 (18)	06	12	18	42	94	55	44	67
passo 4 (42)	06	12	18	42	94	55	44	67
passo 5 (44)	06	12	18	42	44	55	94	67
passo 6 (55)	06	12	18	42	44	55	94	67
passo 7 (67)	06	12	18	42	44	55	67	94

Seleção Direta

```
void selecao(int x[], int n) {
    int i, j, menor, index;
    for (i = 0; i < n-1; i++) {
        menor = x[ i ];
        index = i;
        for (j = i+1; j < n; j++) {
            if (x[ j ] < menor) {
                menor = x[ j ];
                index = j;
            }
        }
        x[ index ] = x[ i ];
        x[ i ] = menor;
    }
}
```

Seleção Direta

- No primeiro passo ocorrem n - 1 comparações, no segundo passo n - 2, e assim por diante
  - Logo, no total, tem-se (n - 1) + (n -2) + ... + 1 = n * (n -1)/2 comparações: $\Theta(n^2)$
- Não existe melhoria se a entrada está completamente ordenada ou desordenada
- É melhor que o Bubble-sort
- É útil apenas quando n é pequeno

Melhor Caso e Pior Caso são iguais: $\Theta(n^2)$

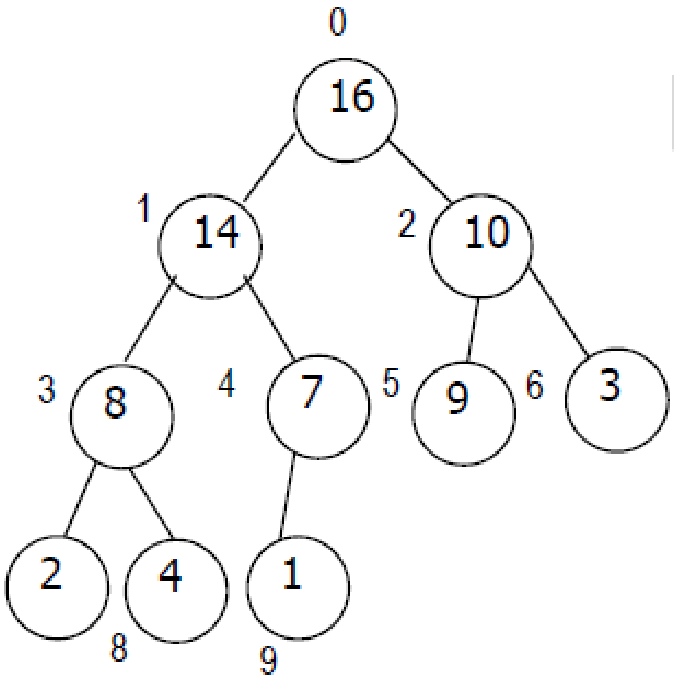
Heap-Sort

Heap-Sort

- Utiliza uma estrutura de dados - um heap – para ordenar os elementos
  - Atenção: a palavra *heap* é utilizada atualmente em algumas linguagens de programação para se referir ao "espaço de armazenamento de lixo coletado"

Heap-Sort

- Um heap é um vetor que implementa (representa) uma árvore binária <u>quase completa</u>



0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

Filhos do nó k:
- filho esquerdo = 2k + 1
- filho direito = 2k + 2

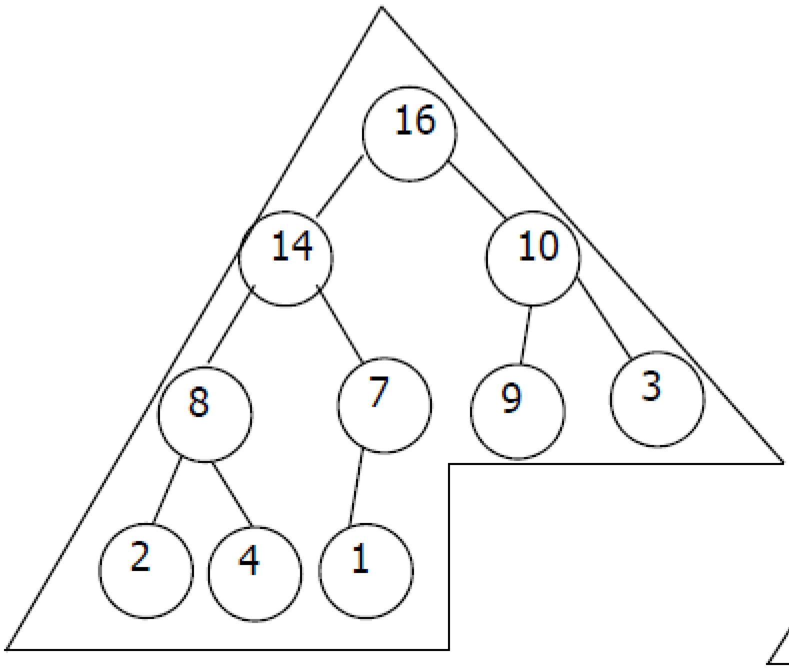
Pai do nó k: (k-1)/2

Folhas de n/2 em diante

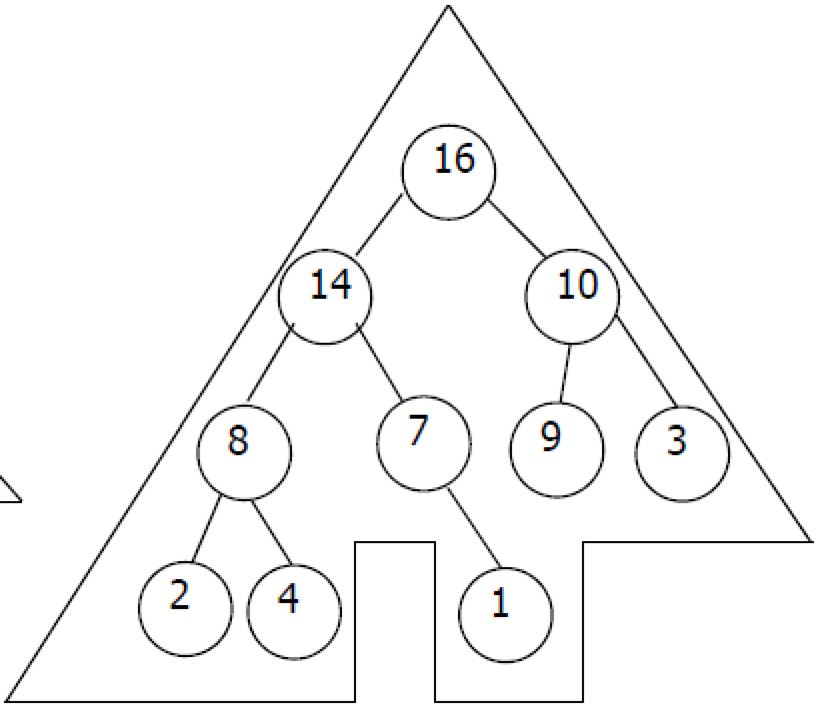
Heap-Sort

- Um heap observa conceitos de ordem e de forma
  - Ordem: o item de qualquer nó deve satisfazer uma relação de ordem com os itens dos nós filhos
    - **Heap máximo** (ou descendente): pai >= filhos, sendo que a raiz é o maior elemento
      - *Propriedade de heap máximo*
    - Heap mínimo (ou heap ascendente): pai <= filhos, sendo que a raiz é o menor elemento
      - *Propriedade de heap mínimo*
  - Forma: a árvore binária tem seus nós-folha, no máximo, em dois níveis, sendo que as folhas devem estar o mais à esquerda possível

Heap-Sort



É um heap máximo



Não é um heap máximo

Heap-Sort

- Assume-se que:
  - A raiz está sempre na posição 0 do vetor
  - comprimento(vetor) indica o número de elementos do vetor
  - tamanho_do_heap(vetor) indica o número de elementos no heap armazenado dentro do vetor
    - Ou seja, embora A[1..comprimento(A)] contenha números válidos, nenhum elemento além de A[tamanho_do_heap(A)] é um elemento do heap, sendo que tamanho_do_heap(A)<=comprimento(A)

Heap-Sort

- A idéia para ordenar usando um heap é:
  - Construir um heap máximo
  - Trocar a raiz – o maior elemento – com o elemento da última posição do vetor
  - Diminuir o tamanho do heap em 1
  - Rearranjar o heap máximo, se necessário
  - Repetir o processo n-1 vezes

Heap-Sort

- O processo continua até todos os elementos terem sido incluídos no vetor de forma ordenada

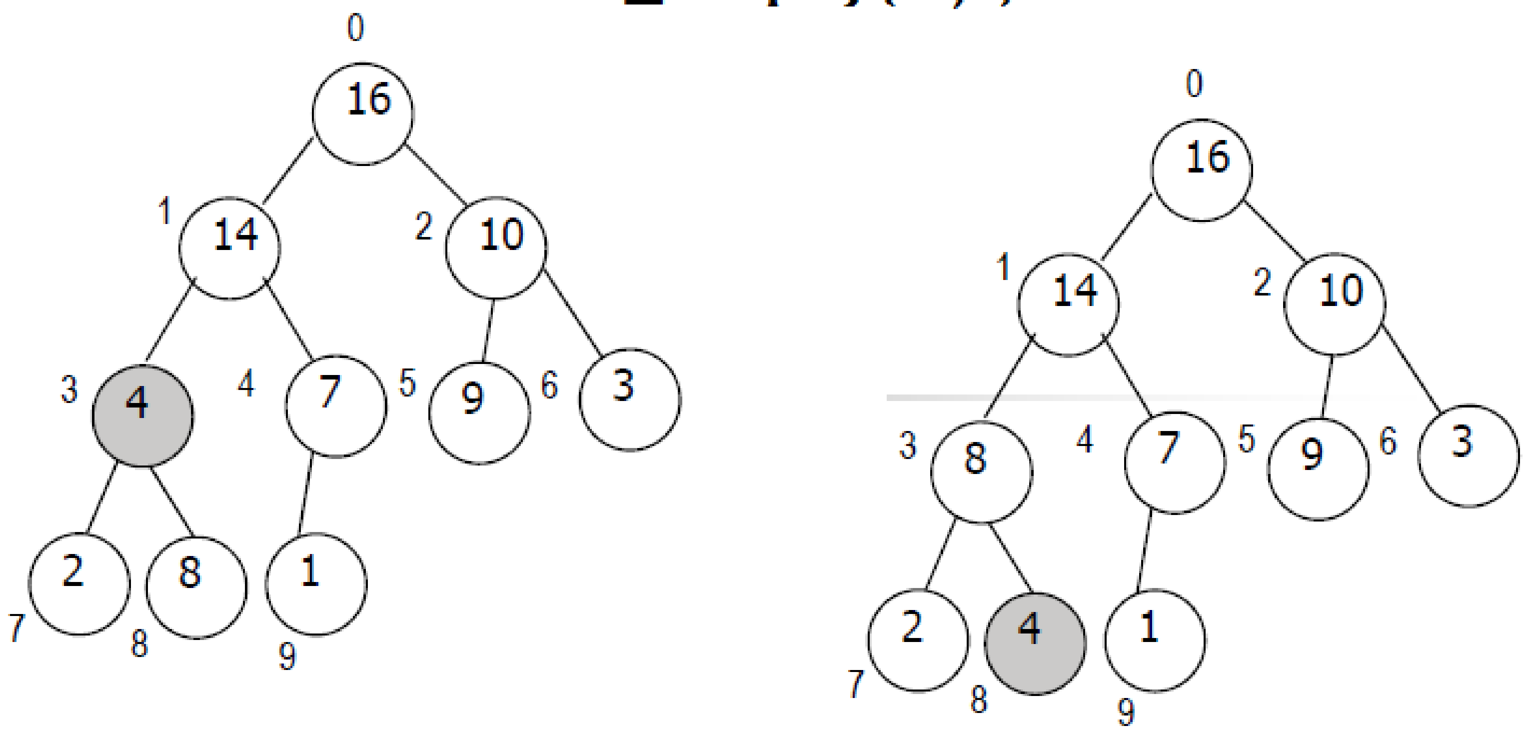
- É necessário:
  - Saber construir um heap a partir de um vetor qualquer
    - Procedimento *build_max_heap*
  - Saber como rearranjar o heap, i.e., manter a propriedade de heap máximo
    - Procedimento *max_heapify*

Heap-Sort

- Procedimento **max_heapify**: manutenação da propriedade de heap máximo
  - Recebe como entrada um vetor A e um índice i
  - Assume que as árvores binárias com raízes nos filhos esquerdo e direito de i são heap máximos, mas que A[i] pode ser menor que seus filhos, violando a propriedade de heap máximo
  - A função do procedimento max_heapify é deixar A[i] "escorregar" para a posição correta, de tal forma que a subárvore com raiz em i torne-se um heap máximo

Heap-Sort

max_heapify(A,3)



Heap-Sort

- Na realidade, trabalhando-se com o vetor A

0	1	2	3	4	5	6	7	8	9
16	14	10	4	7	9	3	2	8	1

Execução recursiva de max_heapify(A,3)

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

- Lembrete: as folhas do heap começam na posição n/2+1

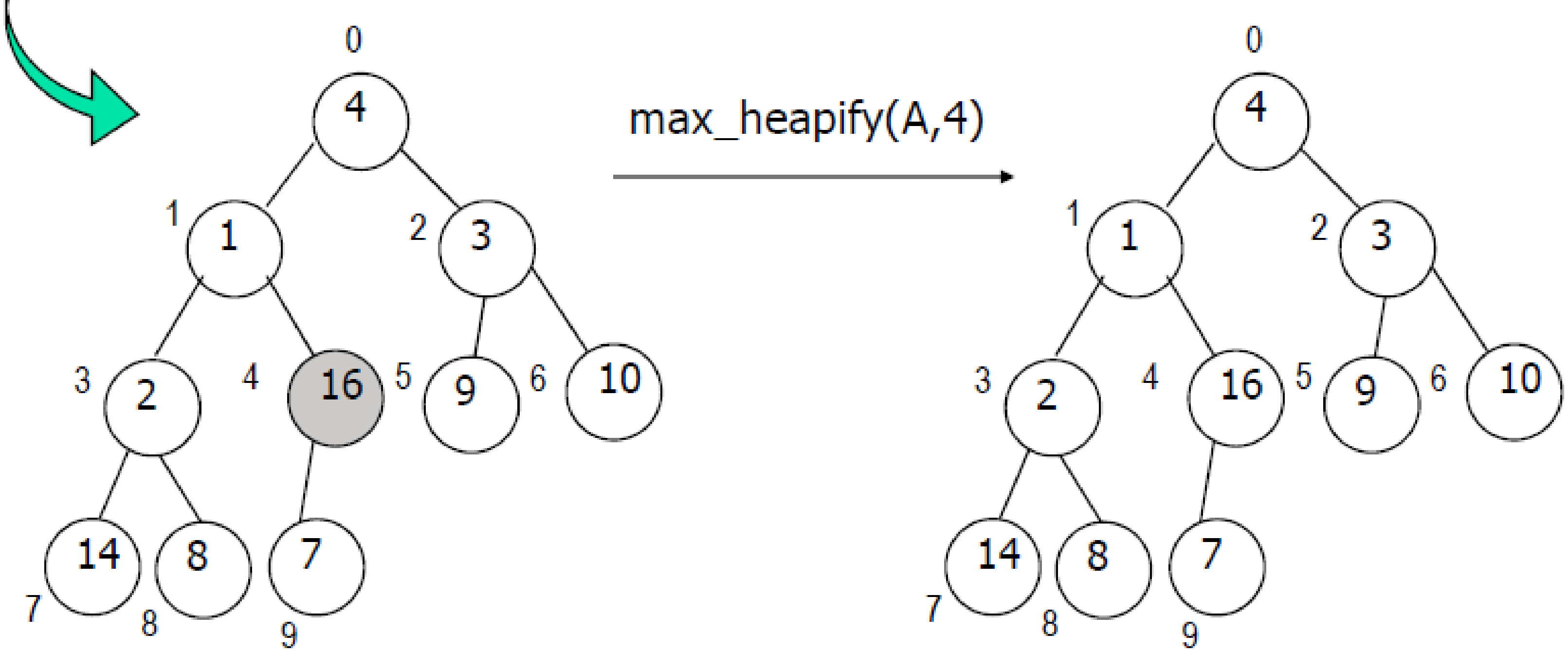
Heap-Sort

- Procedimento build_max_heap
  - Percorre de forma ascendente os primeiros n/2 nós (que não são folhas) e executa o procedimento max_heapify
  - A cada chamada do max_heapify para um nó, as duas árvores com raiz neste nó tornam-se heaps máximos
  - Ao chamar o max_heapify para a raiz, o heap máximo completo é obtido

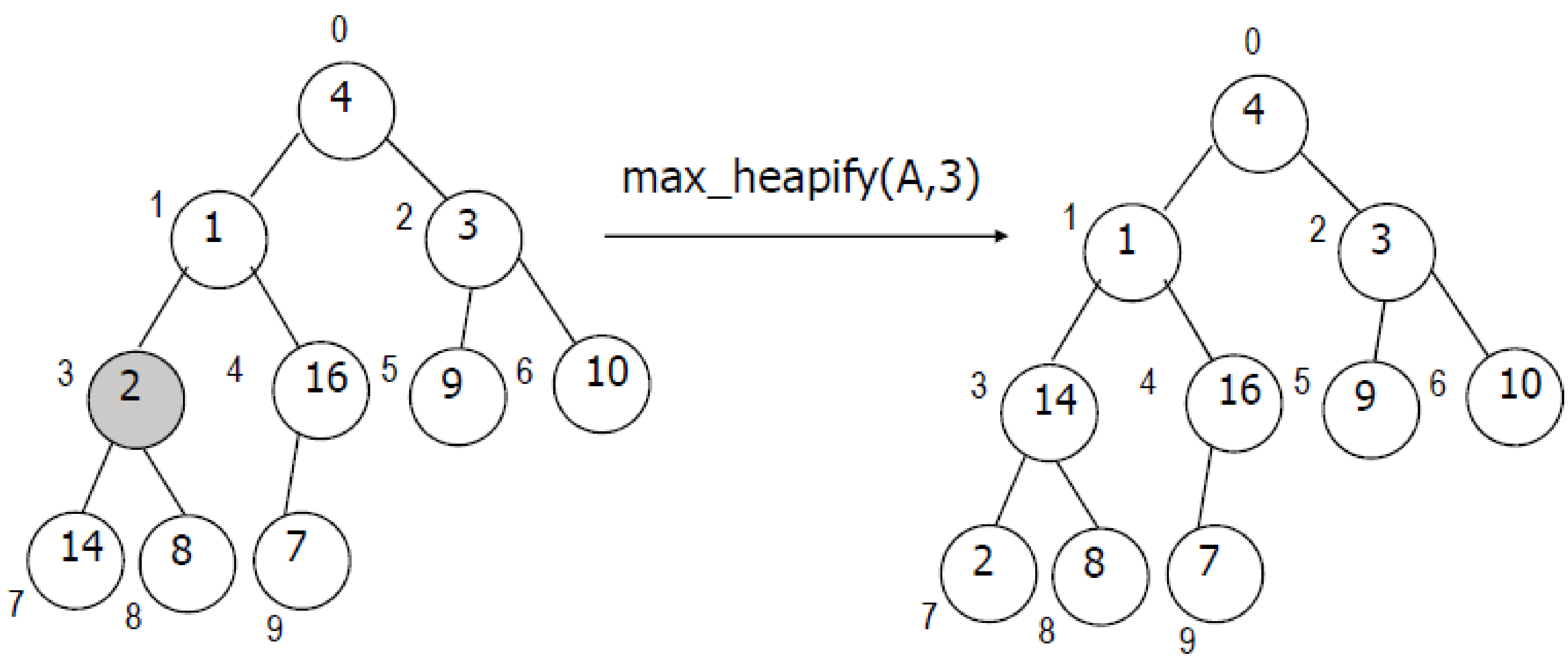
Heap-Sort – Build Max Heap

0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7

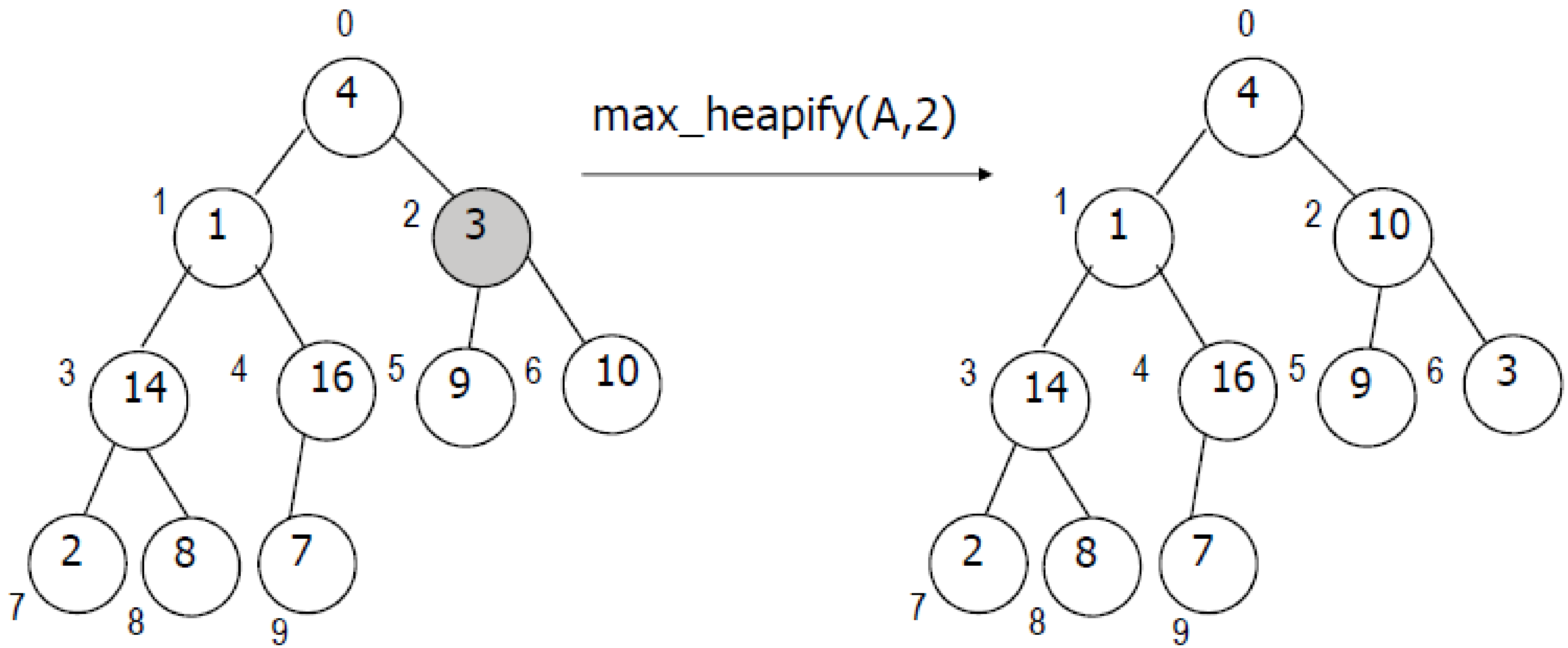
n/2=4



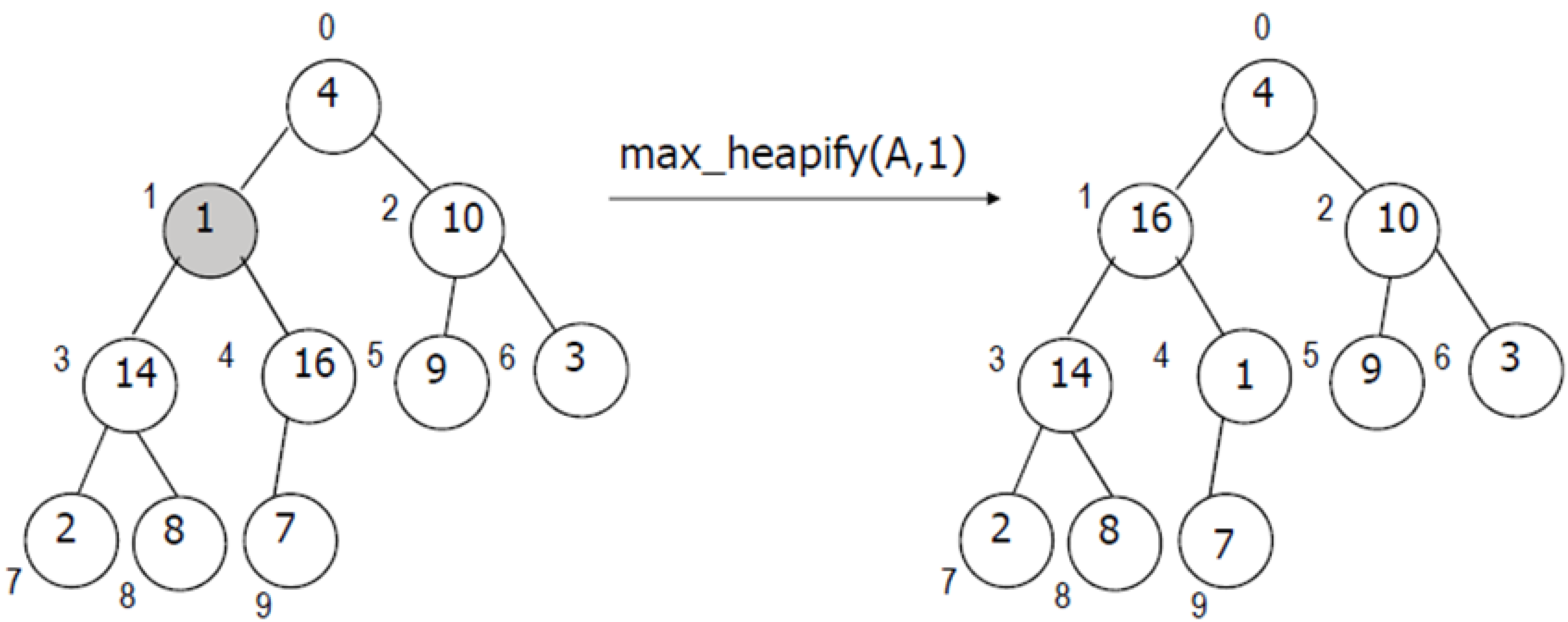
Heap-Sort – Build Max Heap



Heap-Sort – Build Max Heap

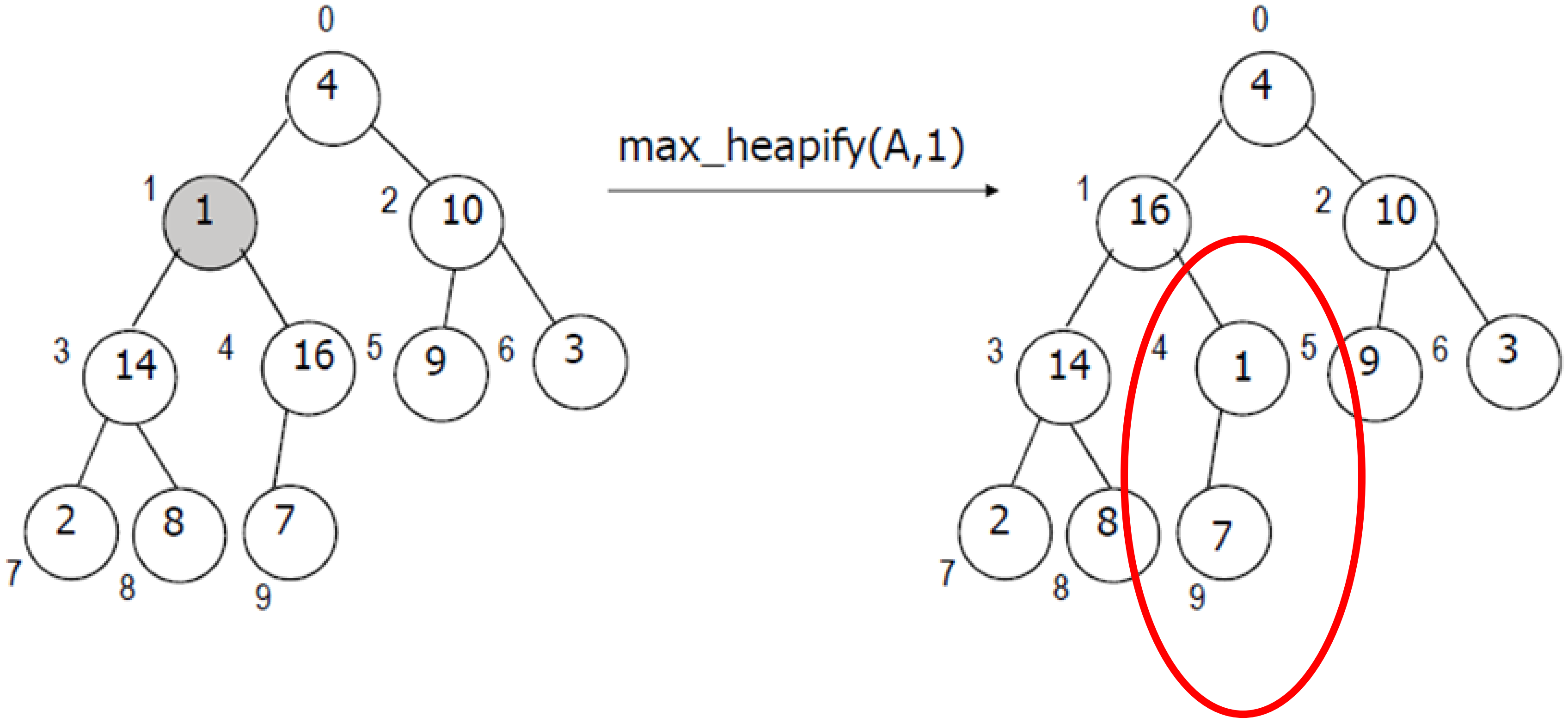


Heap-Sort – Build Max Heap

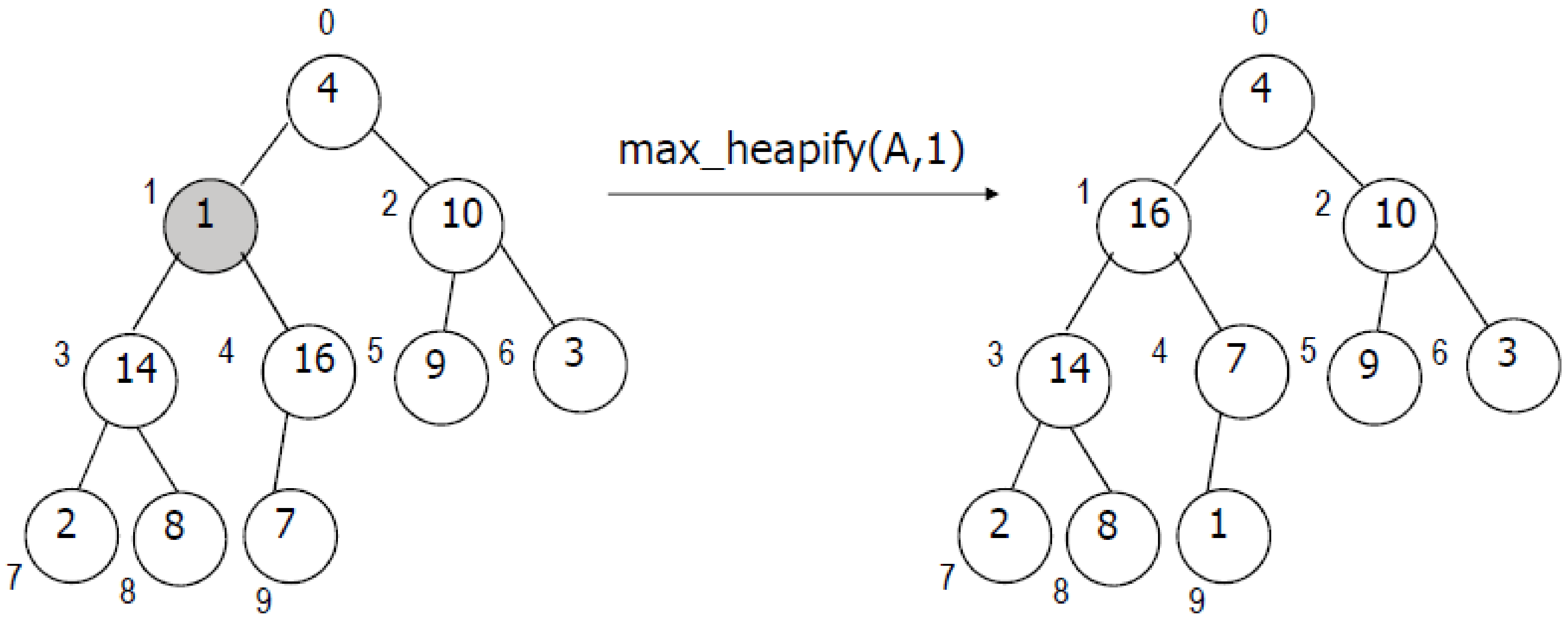


DEPARTAMENTO DE MATEMÁTICA E COMPUTAÇÃO

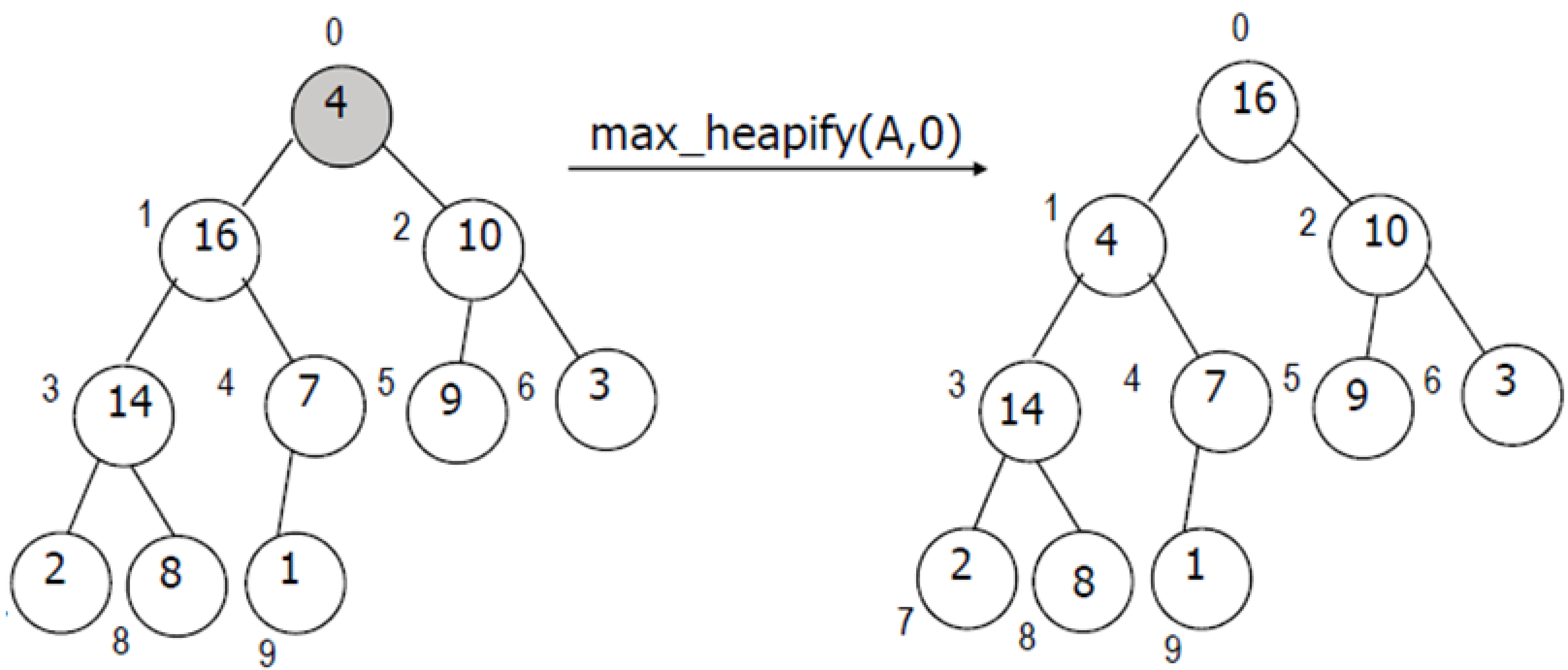
Heap-Sort – Build Max Heap



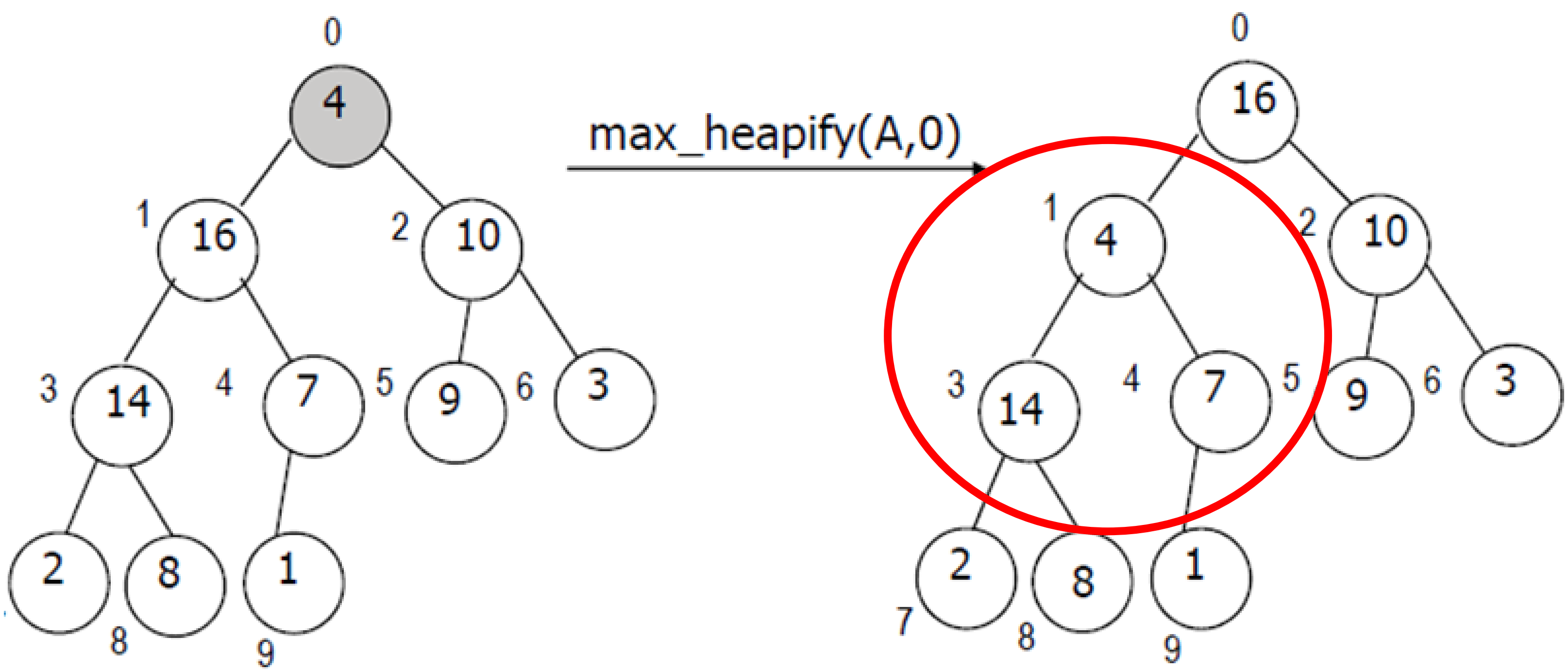
Heap-Sort – Build Max Heap



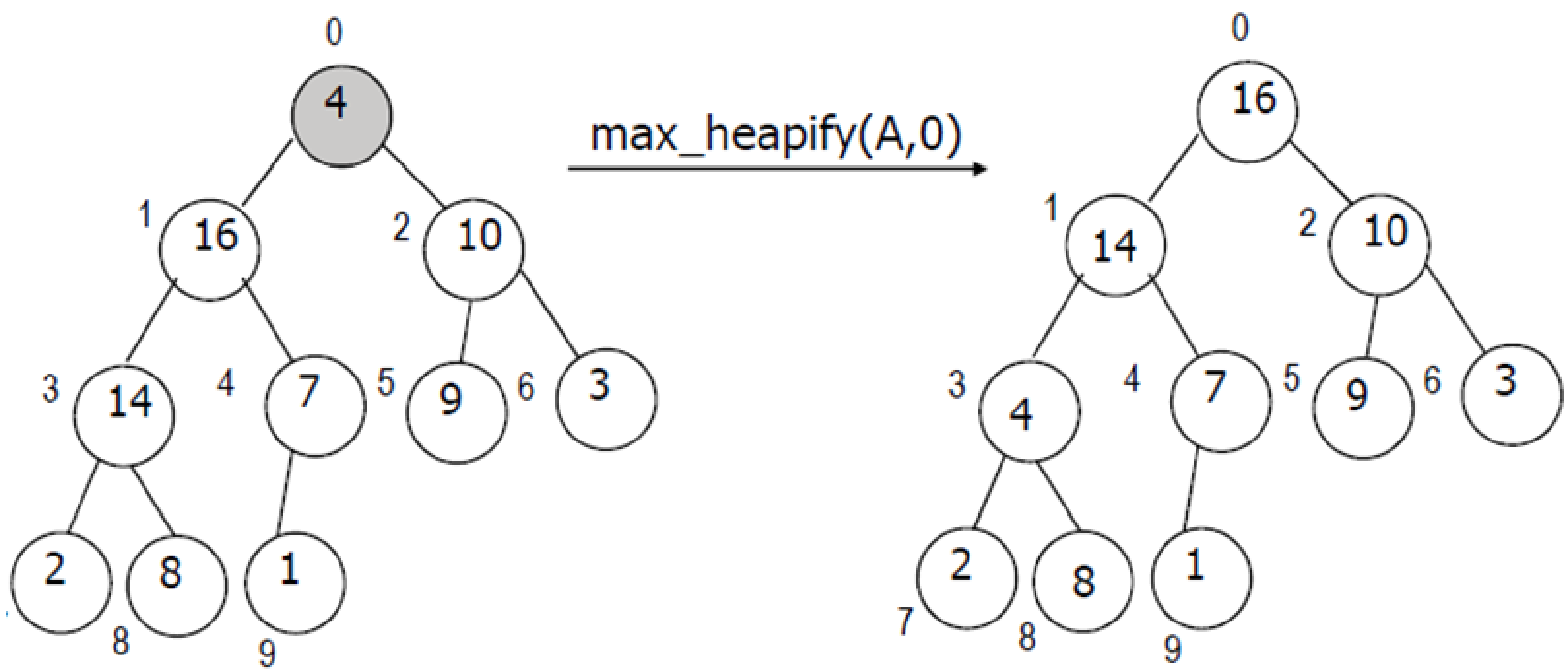
Heap-Sort – Build Max Heap



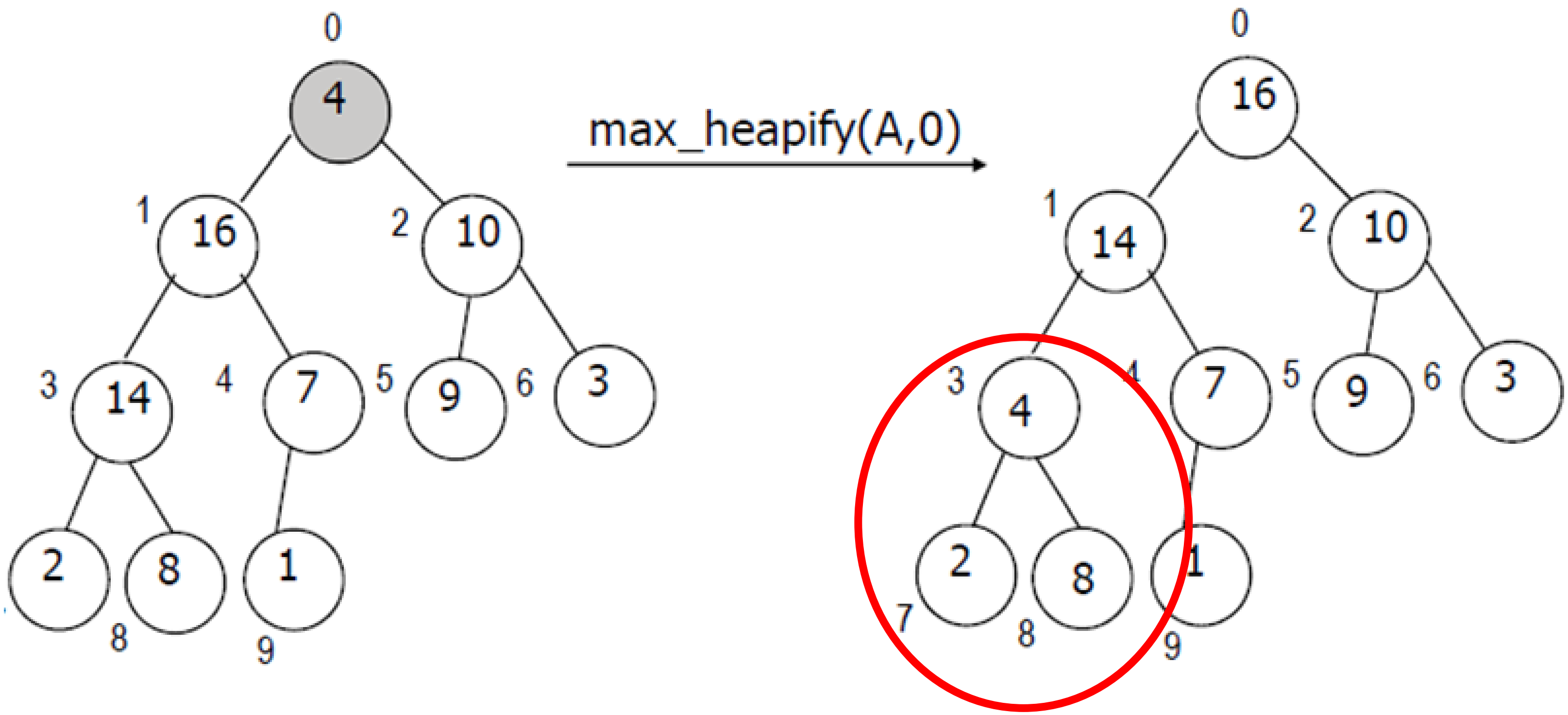
Heap-Sort – Build Max Heap



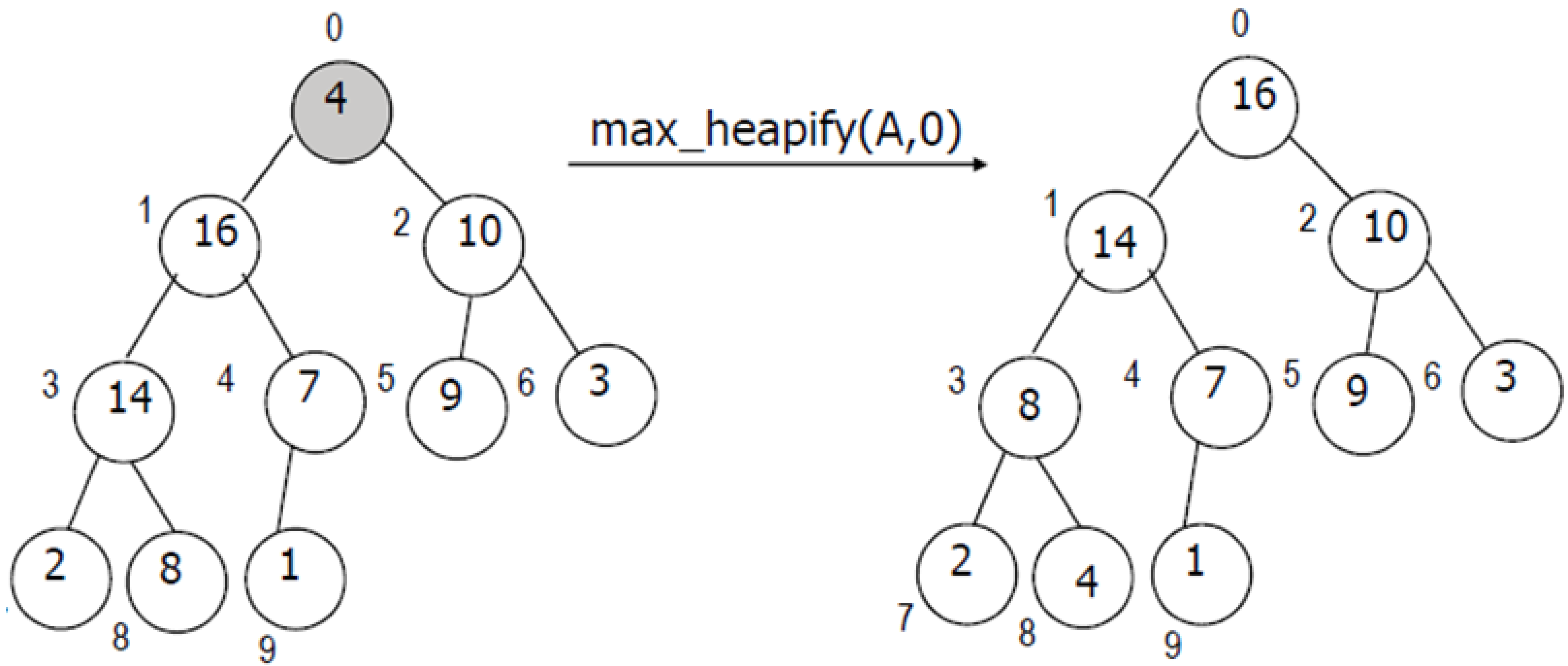
Heap-Sort – Build Max Heap



Heap-Sort – Build Max Heap

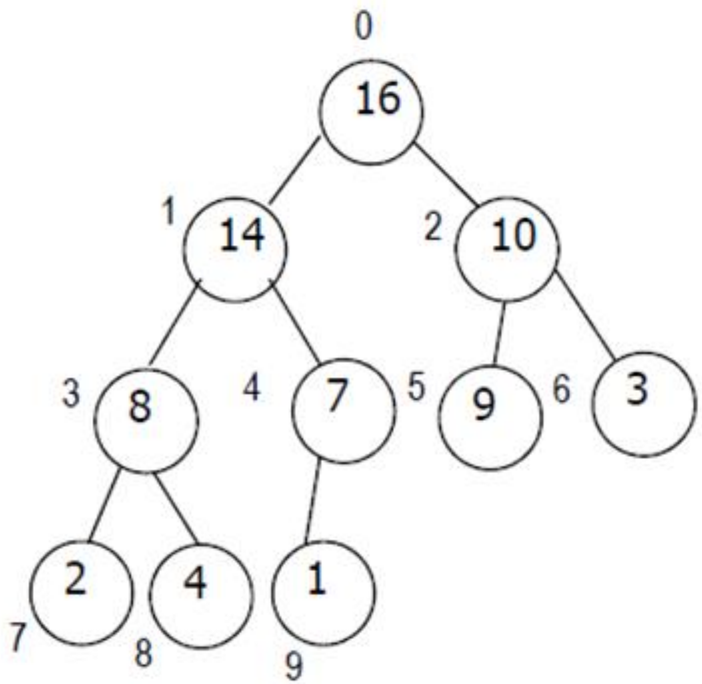


Heap-Sort – Build Max Heap



Heap-Sort

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

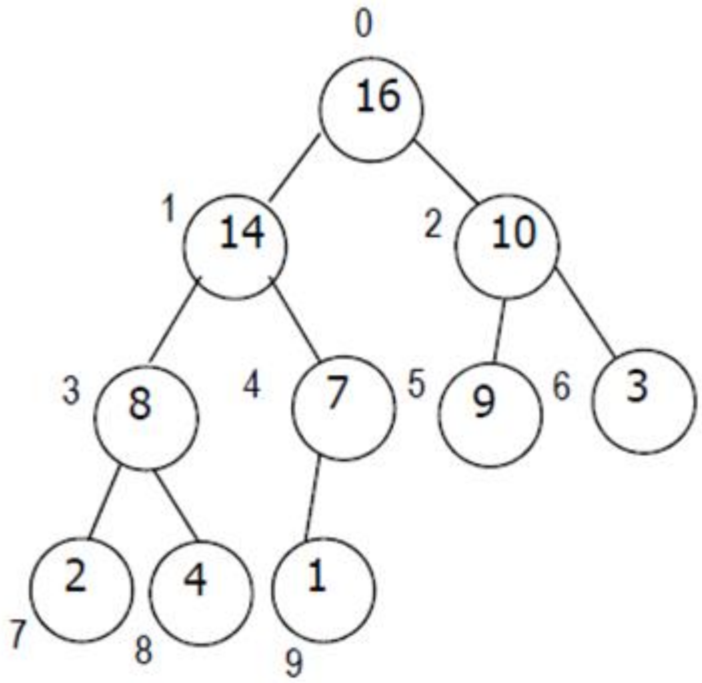


Heap-Sort

- Procedimento Heap-Sort
  - 1- Construir um heap máximo (via **build_max_heap**)
  - 2 – Trocar primeiro elemento com o último
  - 3 – Diminuir tamanho do heap
  - 4 – Rearranjar o heap máximo (aplicar o **max_heapify** para o elemento da posição 1)
  - Repetir os passos 2 a 4 até de N até 1 (nesse caso, N controla o tamanho do heap)

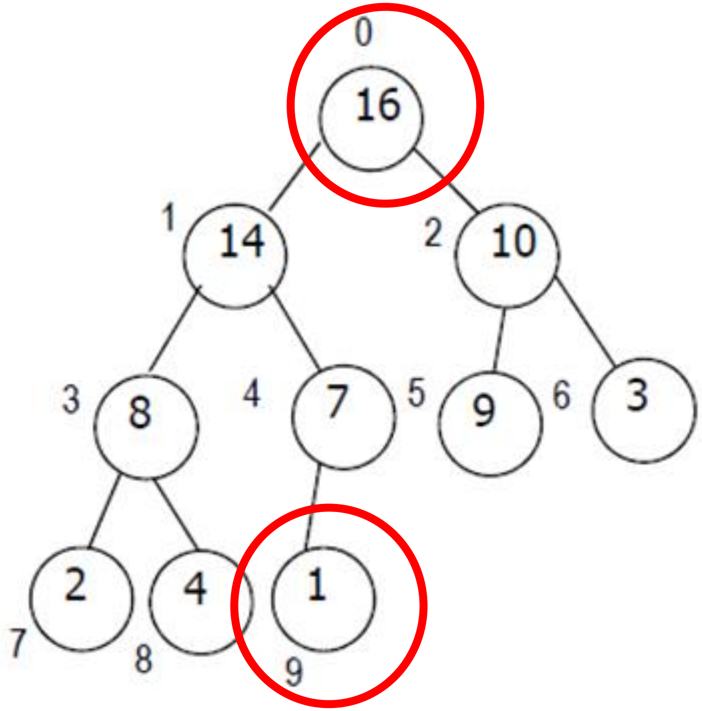
Heap-Sort

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1



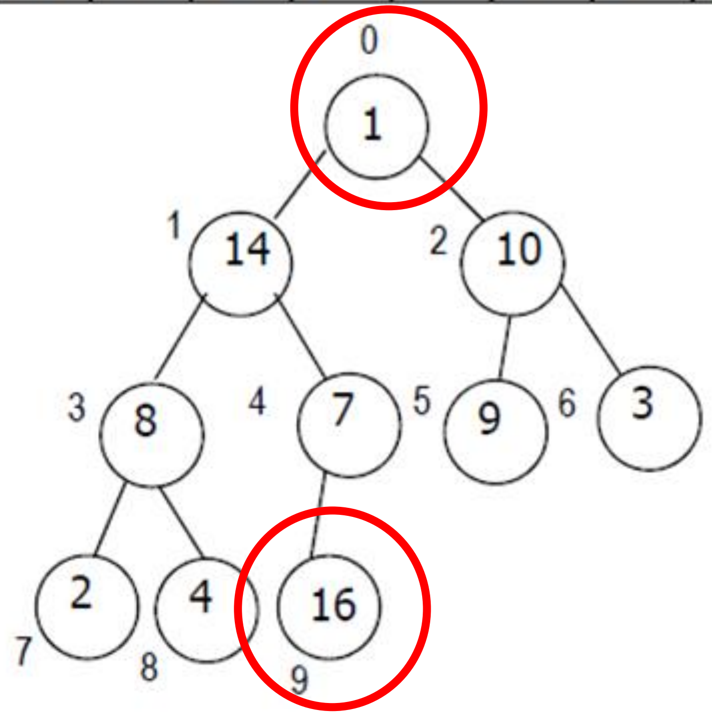
Heap-Sort

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1



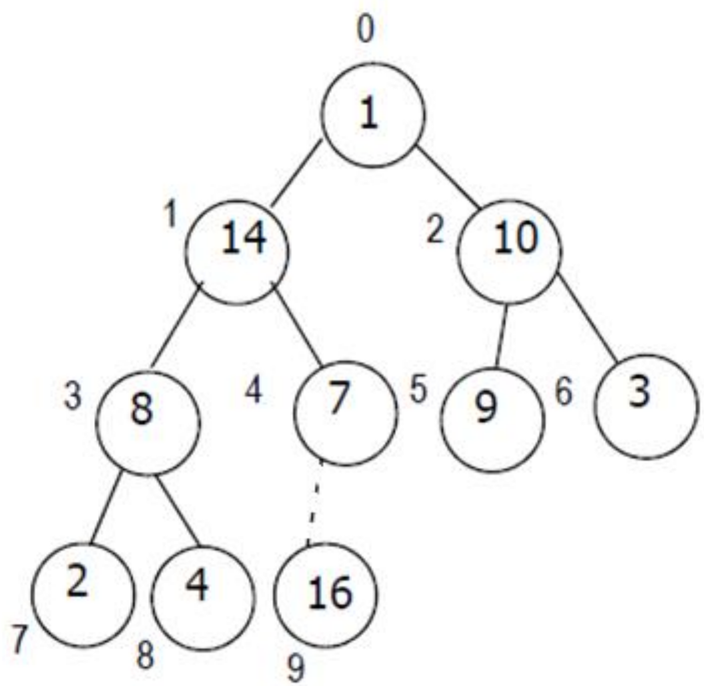
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	14	10	8	7	9	3	2	4	16



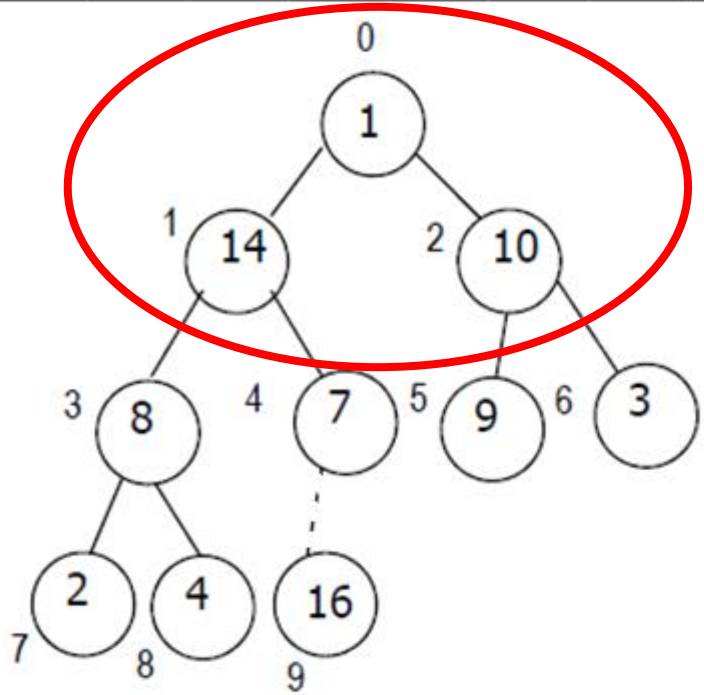
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	14	10	8	7	9	3	2	4	16



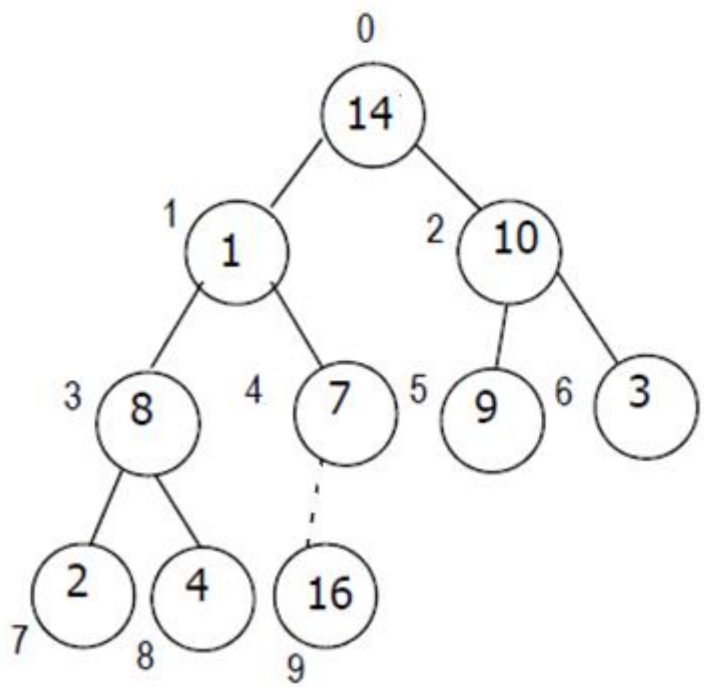
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	14	10	8	7	9	3	2	4	16



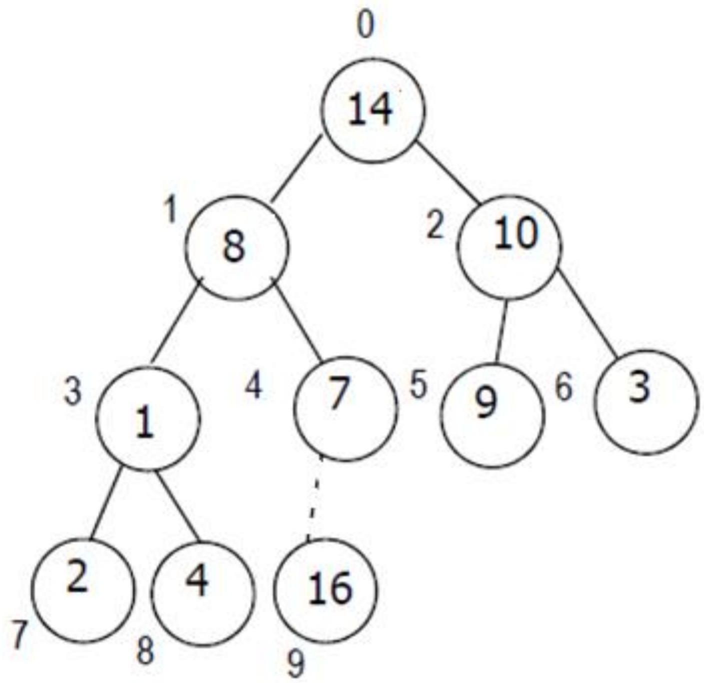
Heap-Sort

0	1	2	3	4	5	6	7	8	9
14	1	10	8	7	9	3	2	4	16



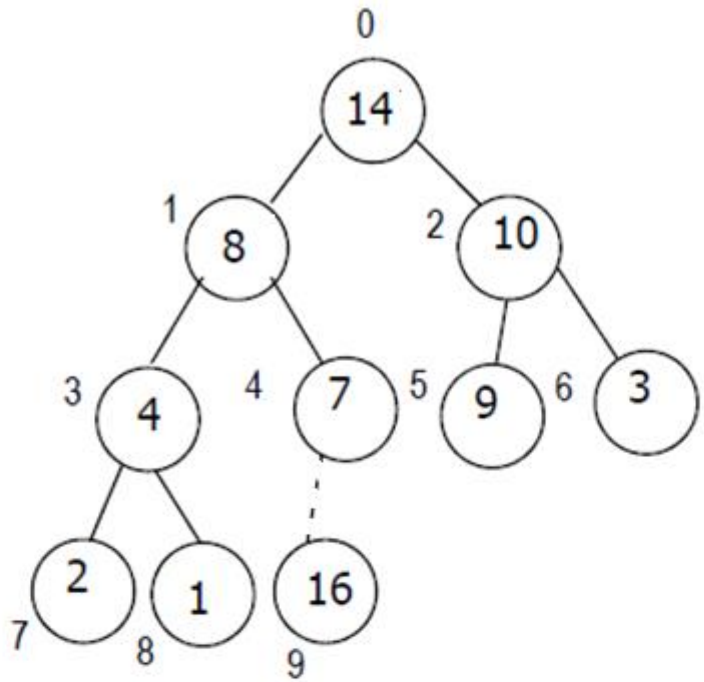
Heap-Sort

0	1	2	3	4	5	6	7	8	9
14	8	10	1	7	9	3	2	4	16



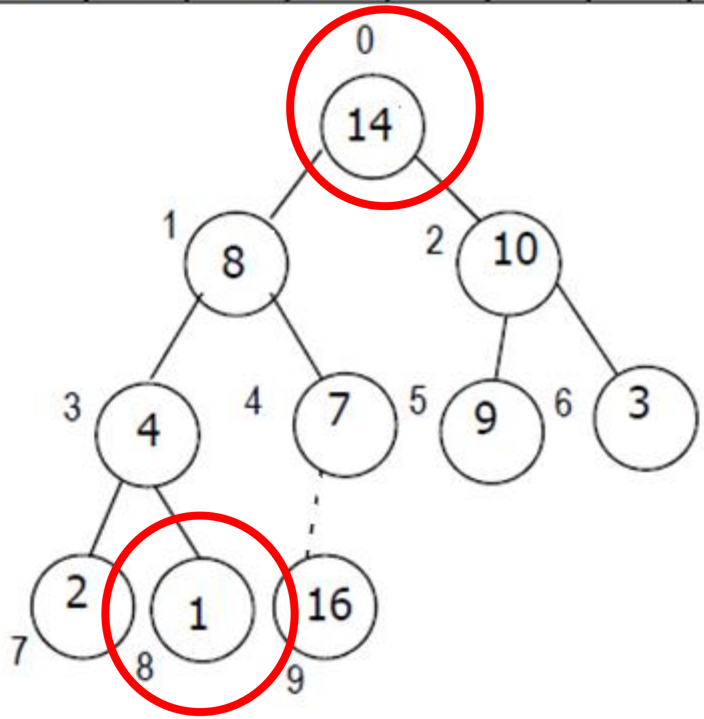
Heap-Sort

0	1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	1	16



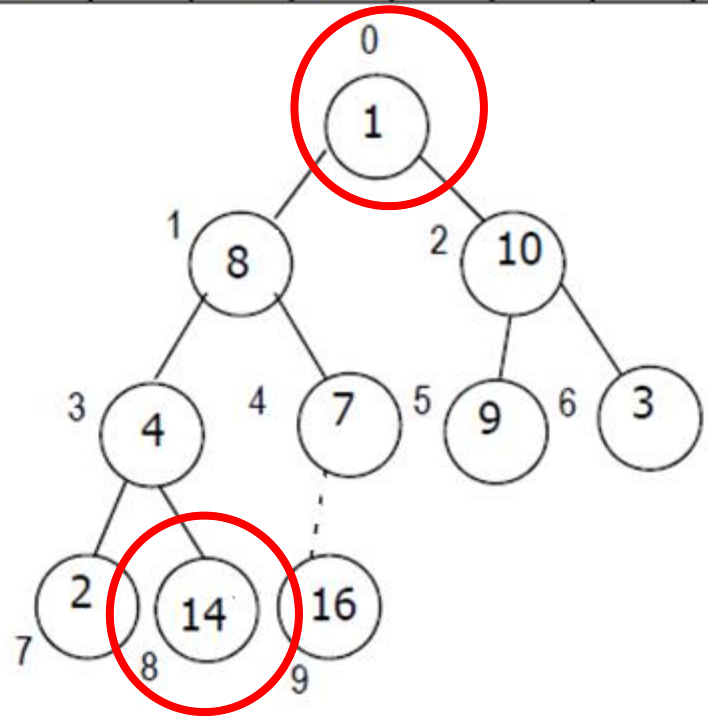
Heap-Sort

0	1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	1	16



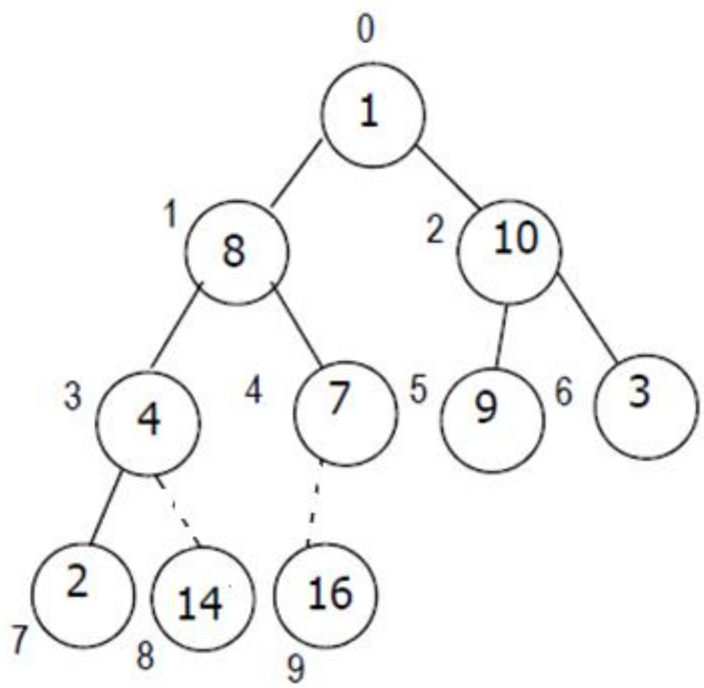
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	8	10	4	7	9	3	2	14	16



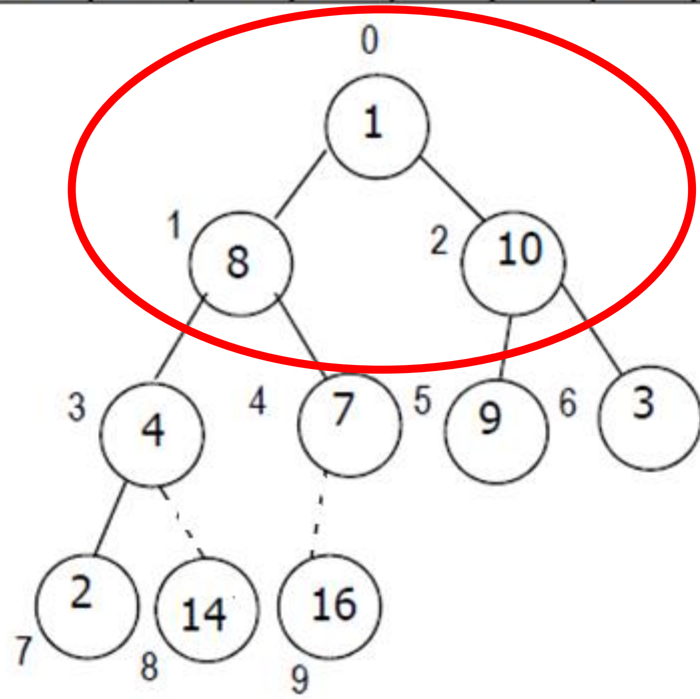
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	8	10	4	7	9	3	2	14	16



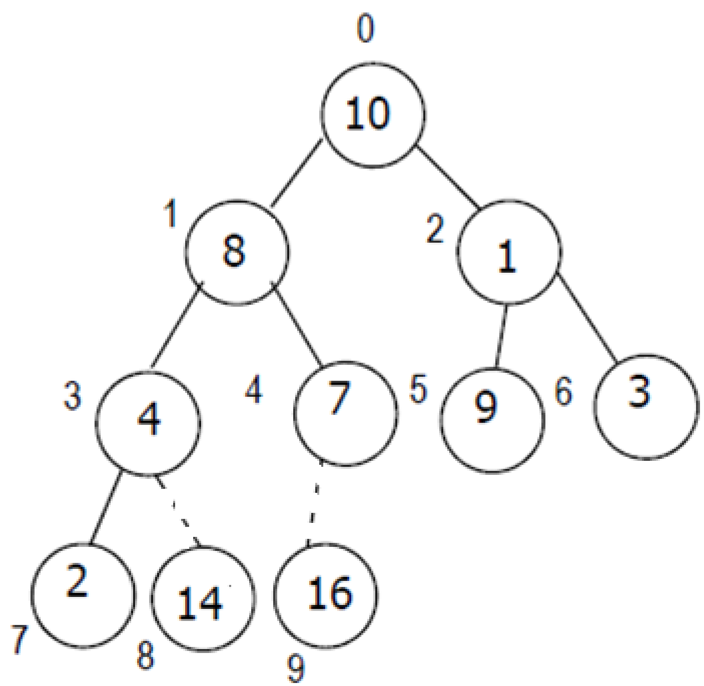
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	8	10	4	7	9	3	2	14	16



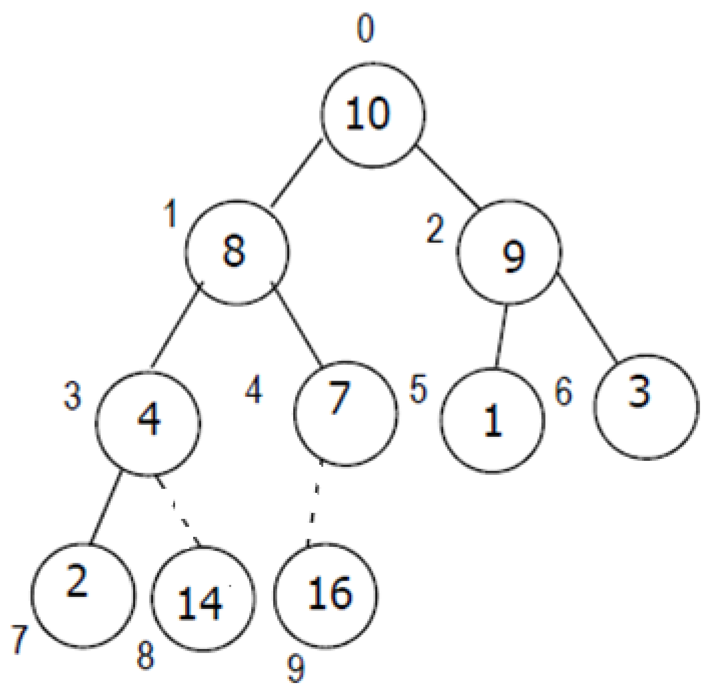
Heap-Sort

0	1	2	3	4	5	6	7	8	9
10	8	1	4	7	9	3	2	14	16



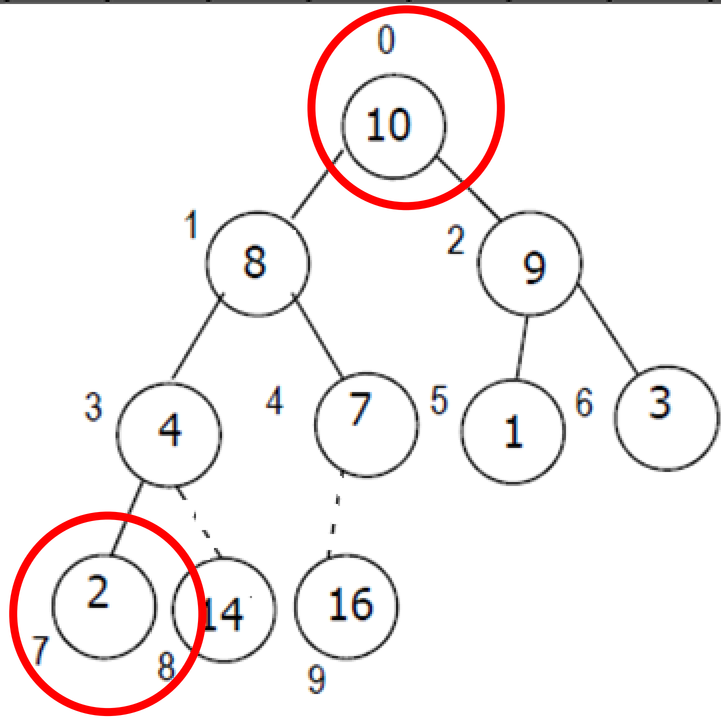
Heap-Sort

0	1	2	3	4	5	6	7	8	9
10	8	9	4	7	1	3	2	14	16



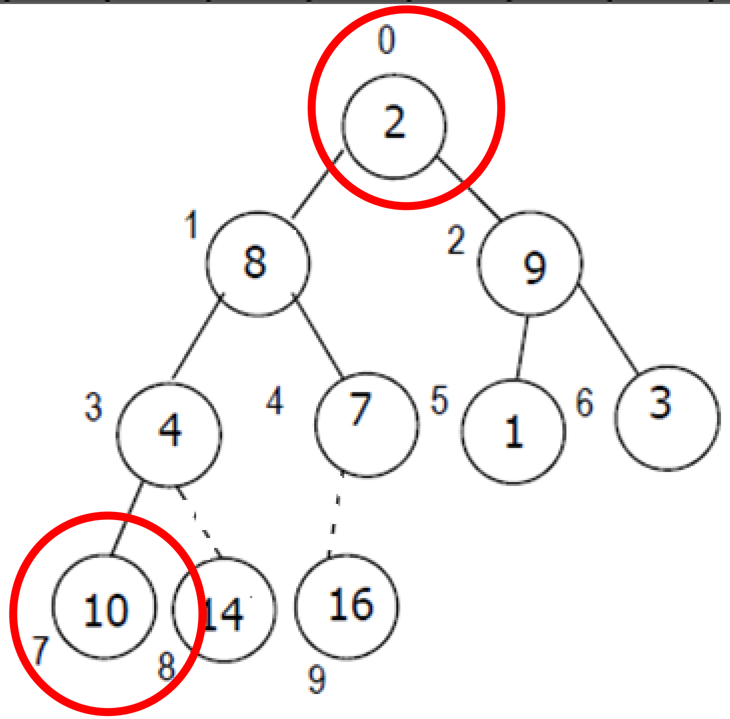
Heap-Sort

0	1	2	3	4	5	6	7	8	9
10	8	9	4	7	1	3	2	14	16



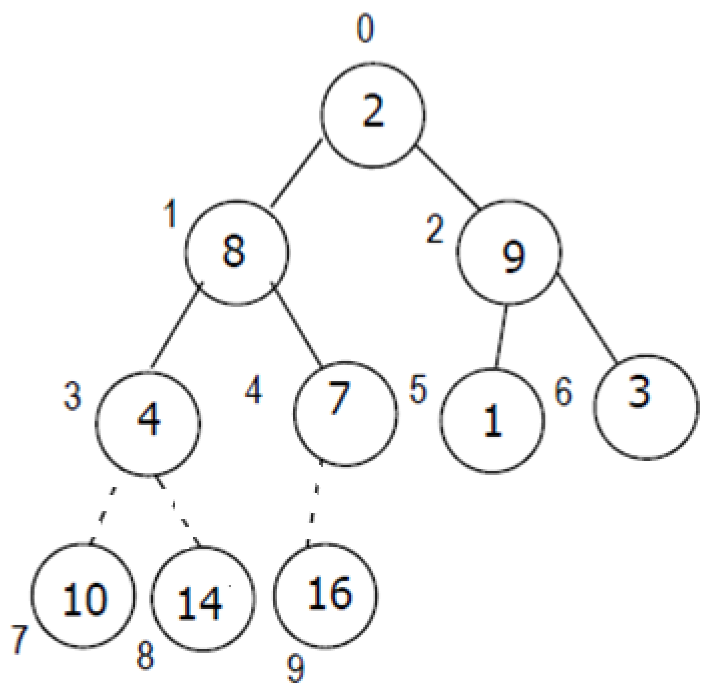
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	8	9	4	7	1	3	10	14	16



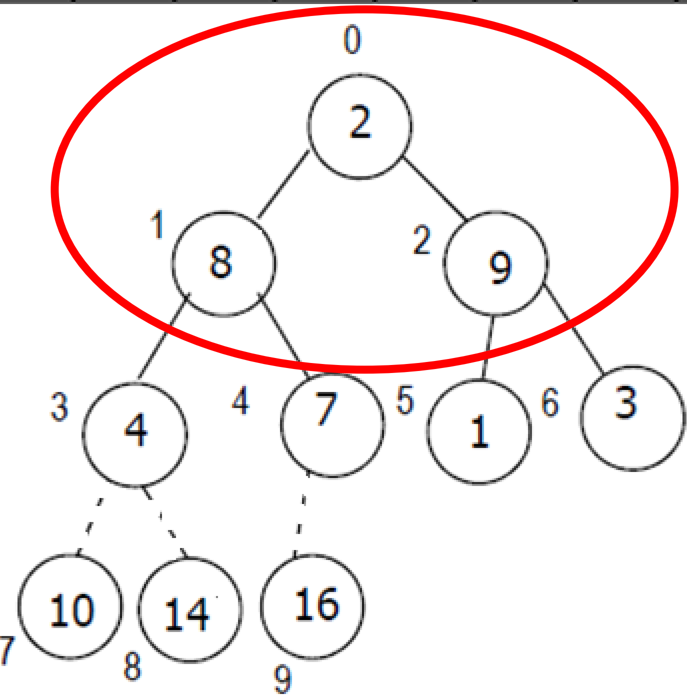
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	8	9	4	7	1	3	10	14	16



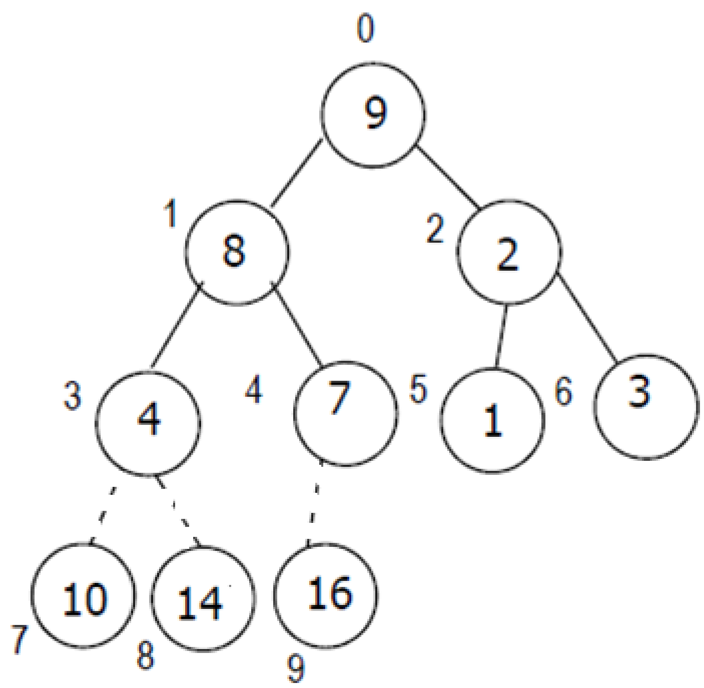
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	8	9	4	7	1	3	10	14	16



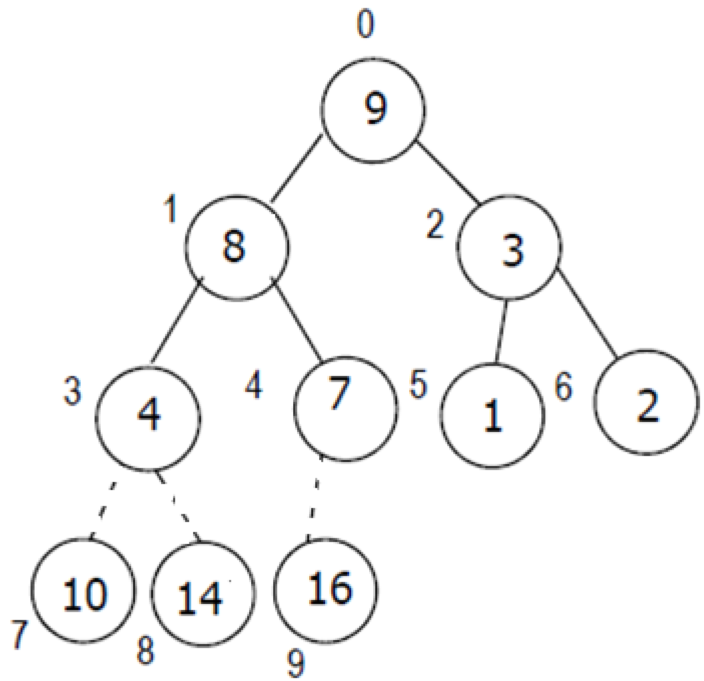
Heap-Sort

0	1	2	3	4	5	6	7	8	9
9	8	2	4	7	1	3	10	14	16



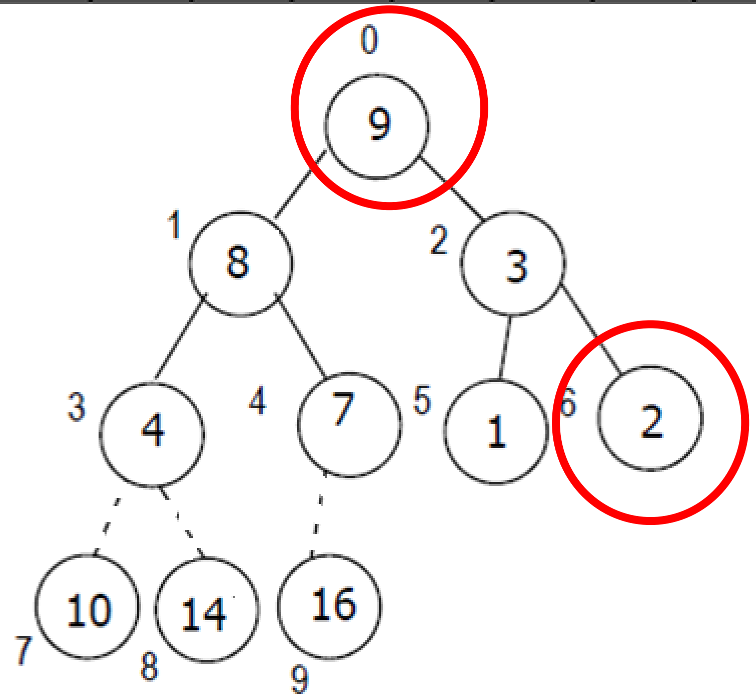
Heap-Sort

0	1	2	3	4	5	6	7	8	9
9	8	3	4	7	1	2	10	14	16



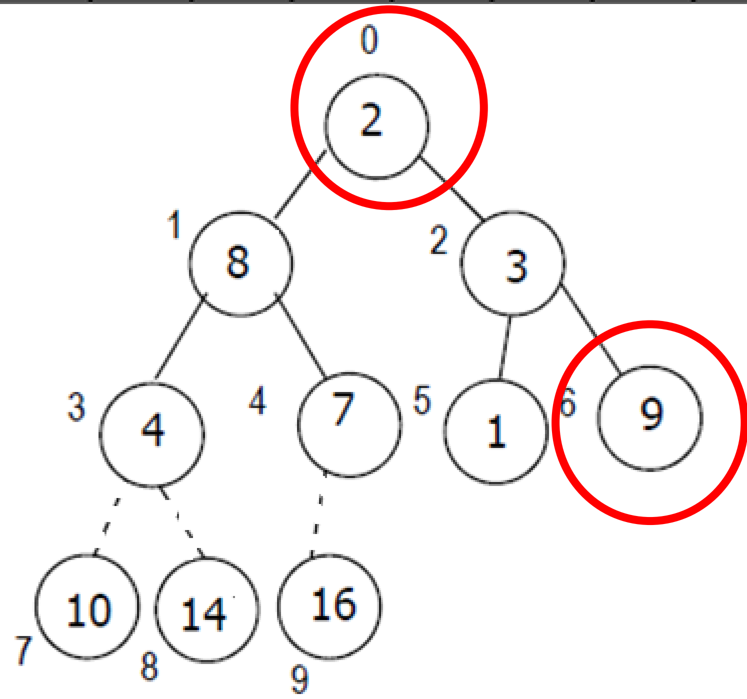
Heap-Sort

0	1	2	3	4	5	6	7	8	9
9	8	3	4	7	1	2	10	14	16



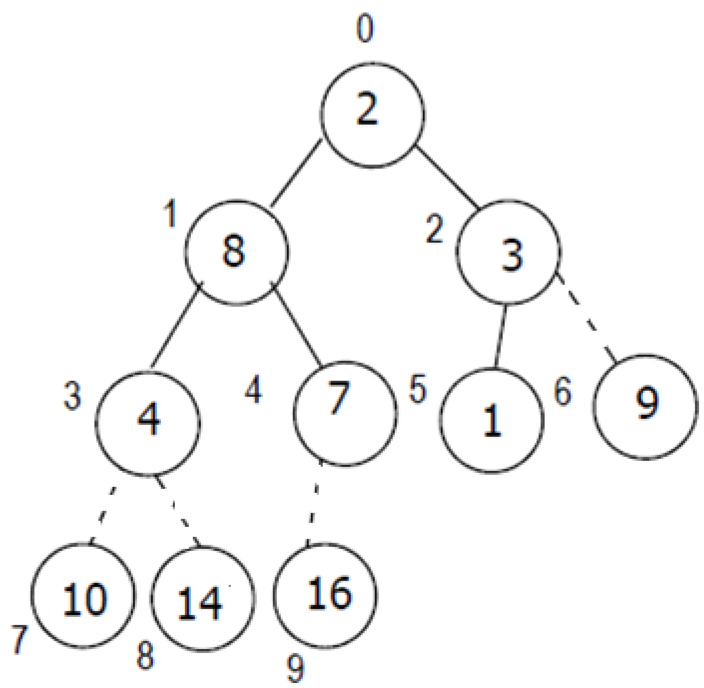
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	8	3	4	7	1	9	10	14	16



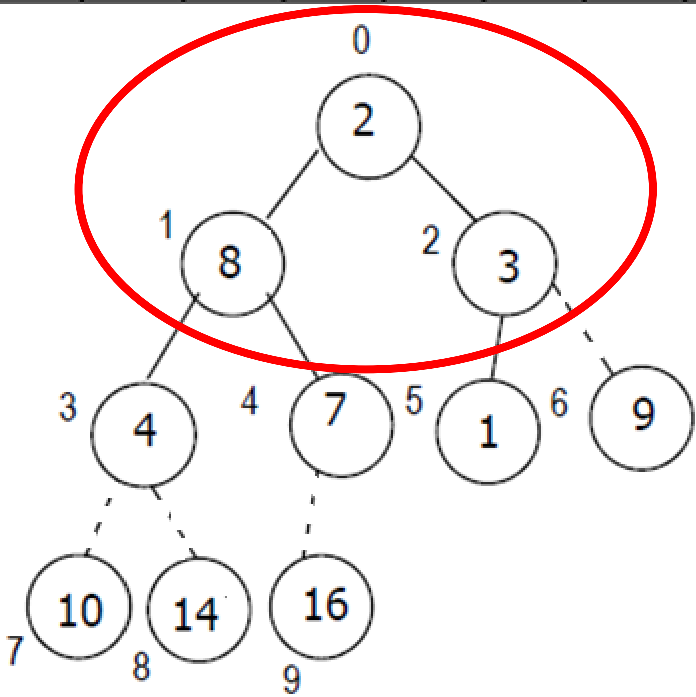
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	8	3	4	7	1	9	10	14	16



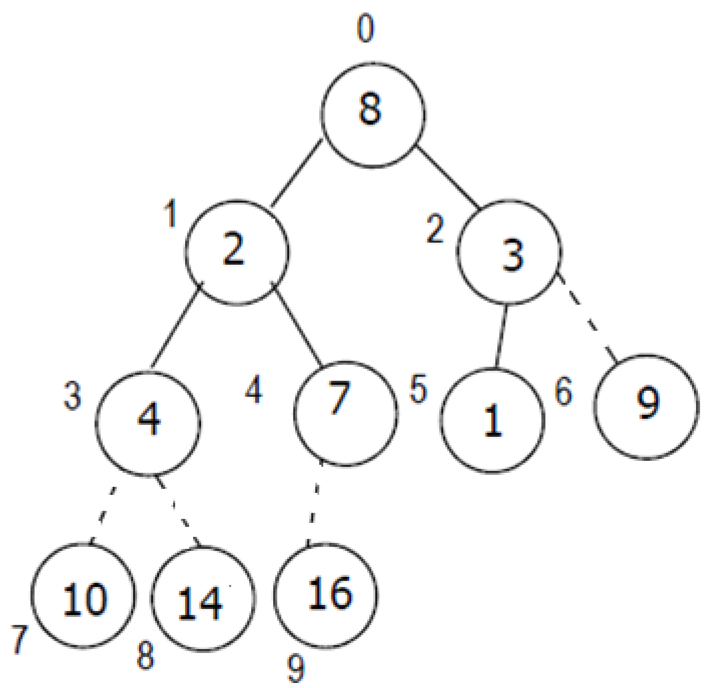
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	8	3	4	7	1	9	10	14	16



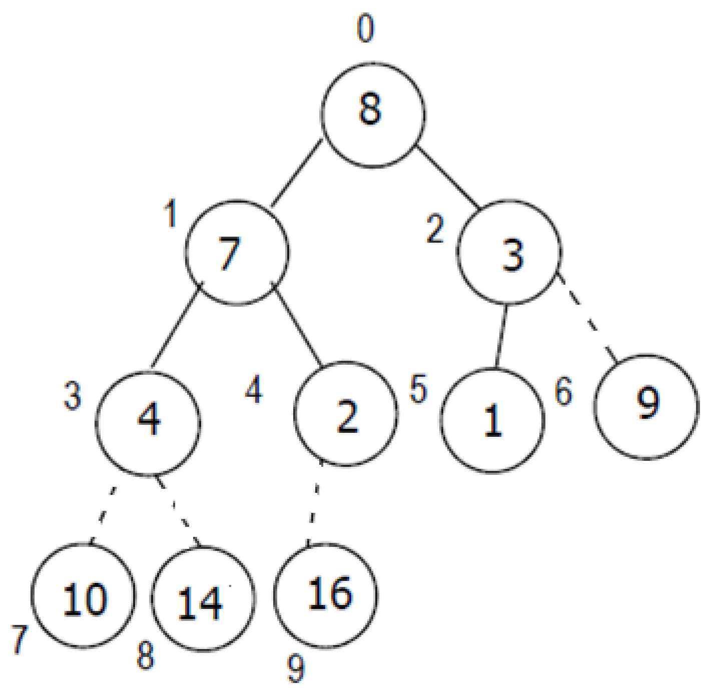
Heap-Sort

0	1	2	3	4	5	6	7	8	9
8	2	3	4	7	1	9	10	14	16



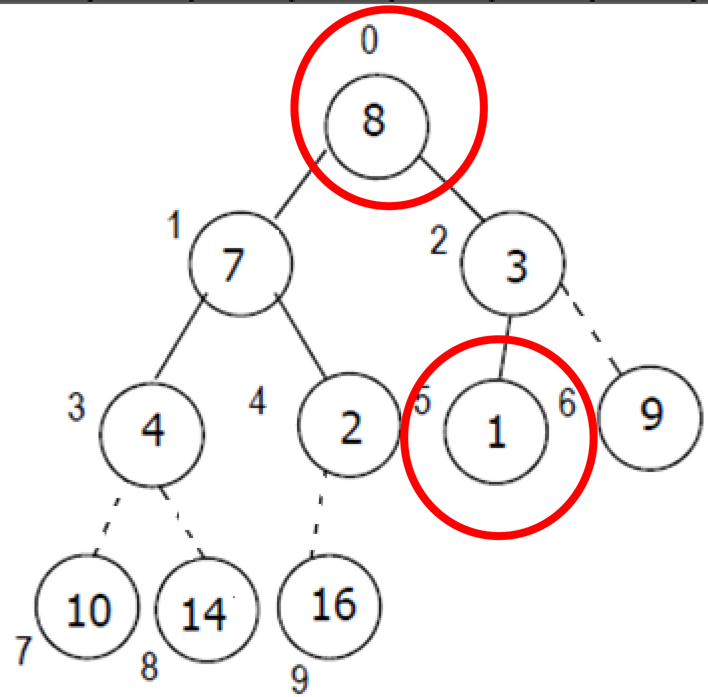
Heap-Sort

0	1	2	3	4	5	6	7	8	9
8	7	3	4	2	1	9	10	14	16



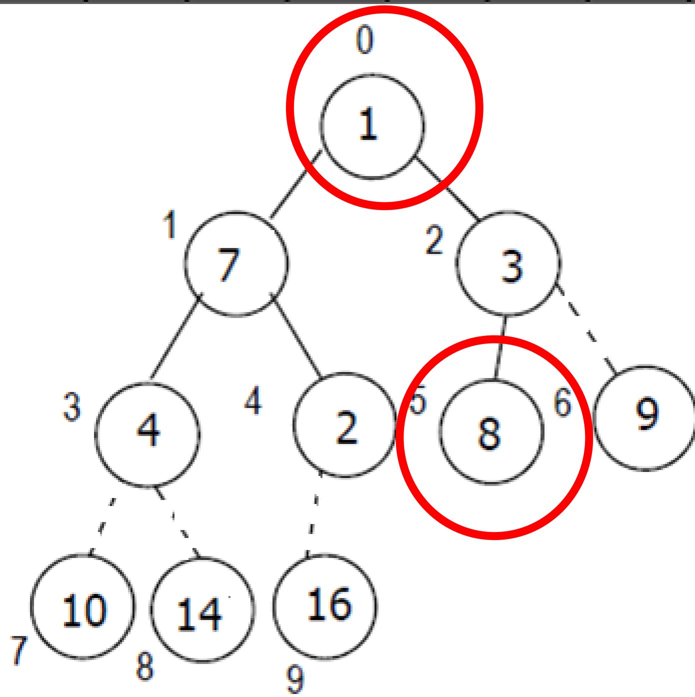
Heap-Sort

0	1	2	3	4	5	6	7	8	9
8	7	3	4	2	1	9	10	14	16



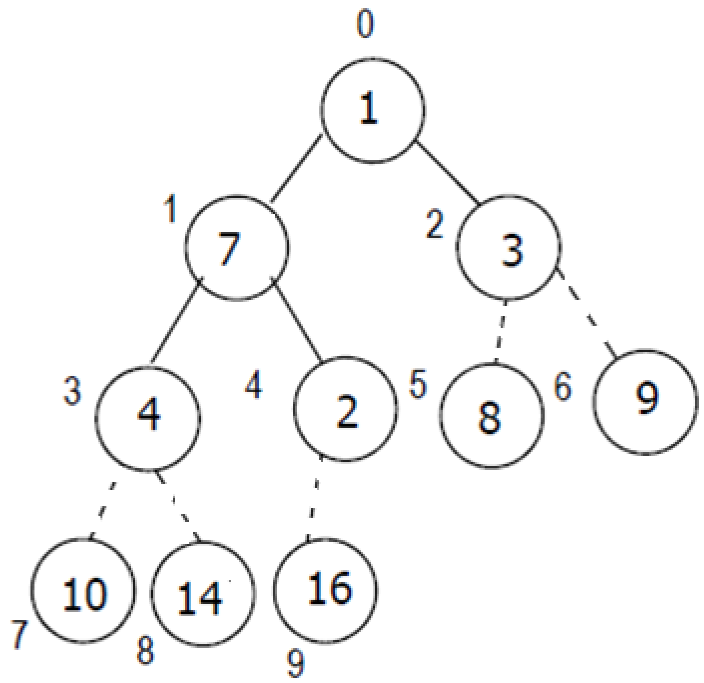
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	7	3	4	2	8	9	10	14	16



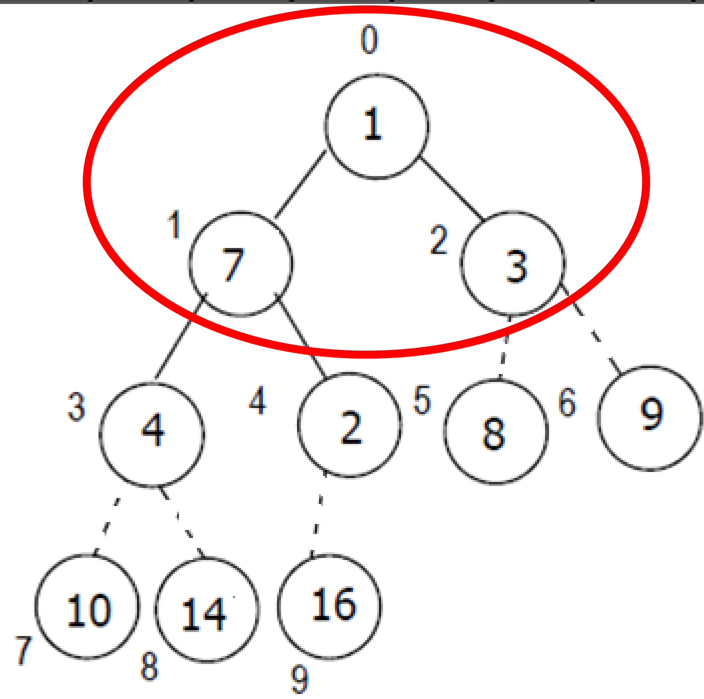
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	7	3	4	2	8	9	10	14	16



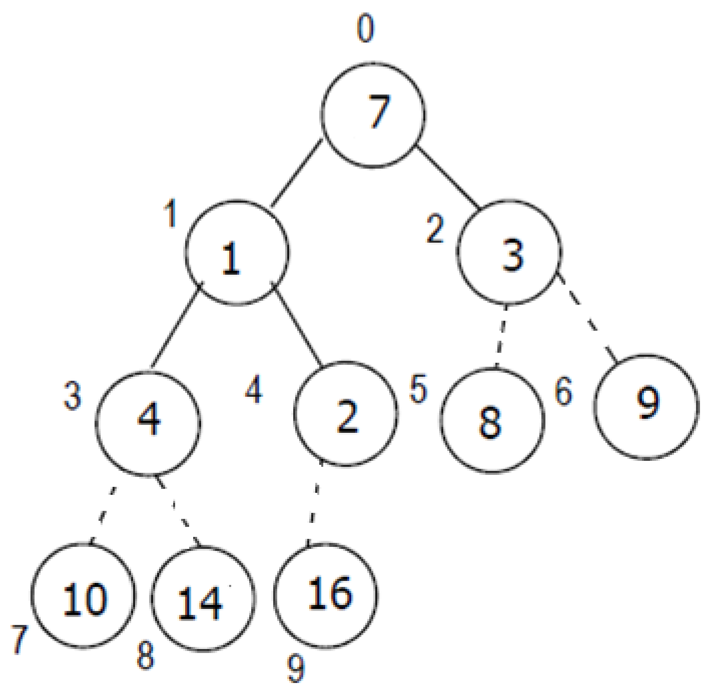
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	7	3	4	2	8	9	10	14	16



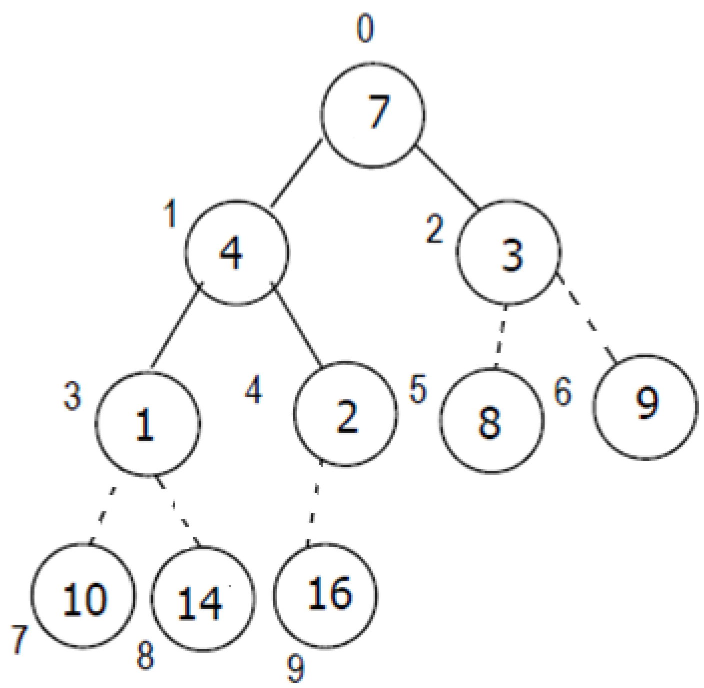
Heap-Sort

0	1	2	3	4	5	6	7	8	9
7	1	3	4	2	8	9	10	14	16



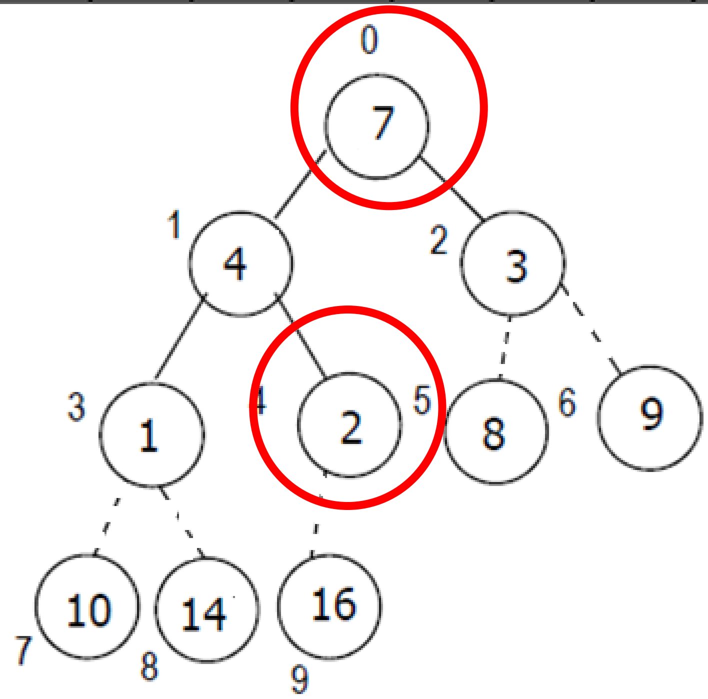
Heap-Sort

0	1	2	3	4	5	6	7	8	9
7	4	3	1	2	8	9	10	14	16



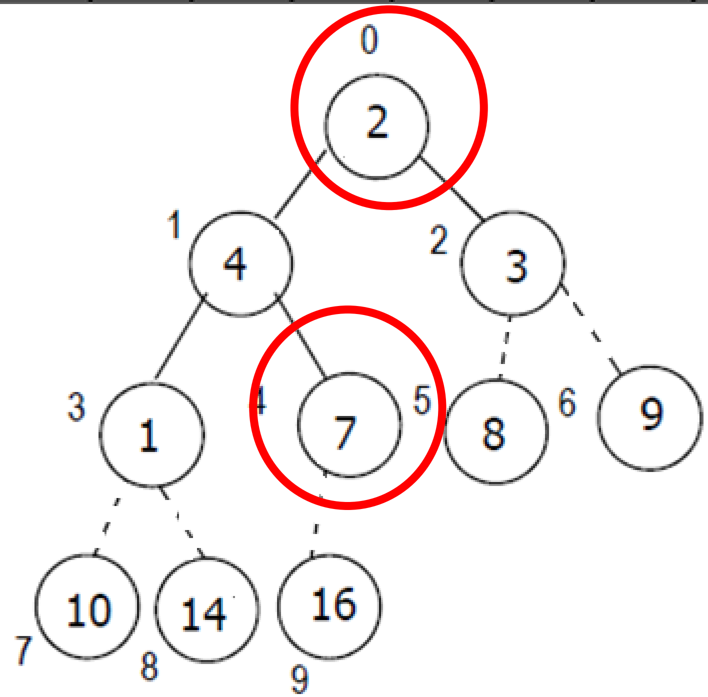
Heap-Sort

0	1	2	3	4	5	6	7	8	9
7	4	3	1	2	8	9	10	14	16



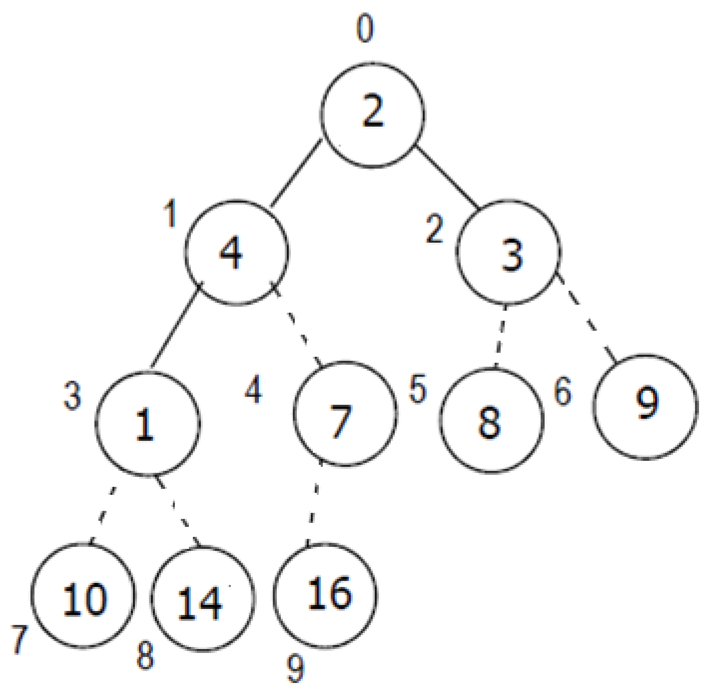
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	4	3	1	7	8	9	10	14	16



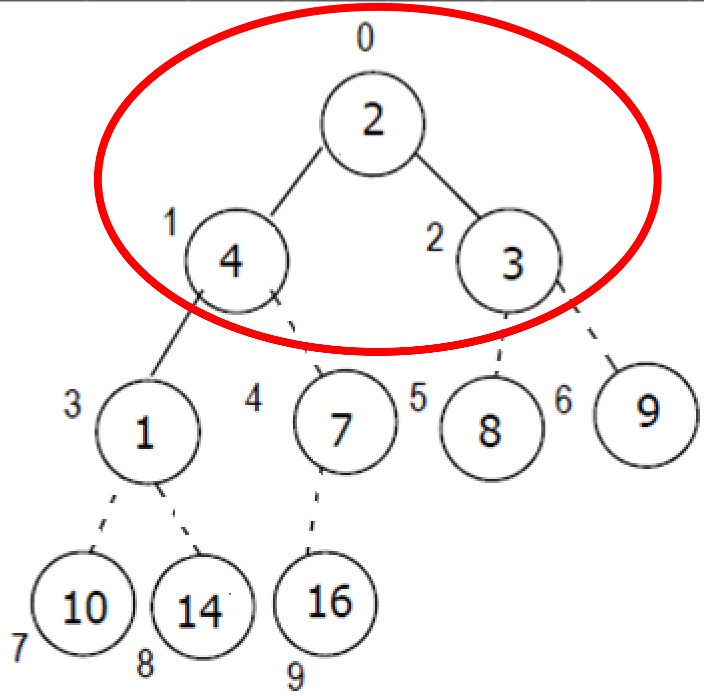
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	4	3	1	7	8	9	10	14	16



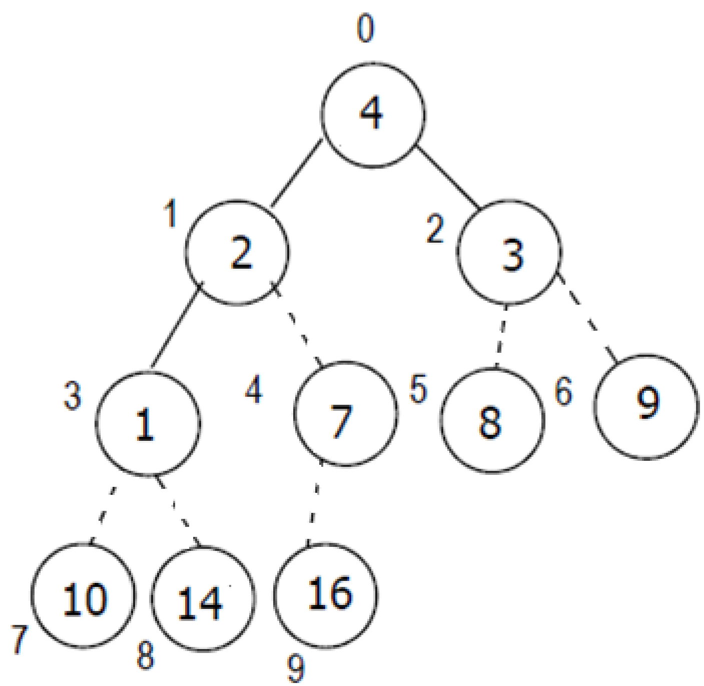
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	4	3	1	7	8	9	10	14	16



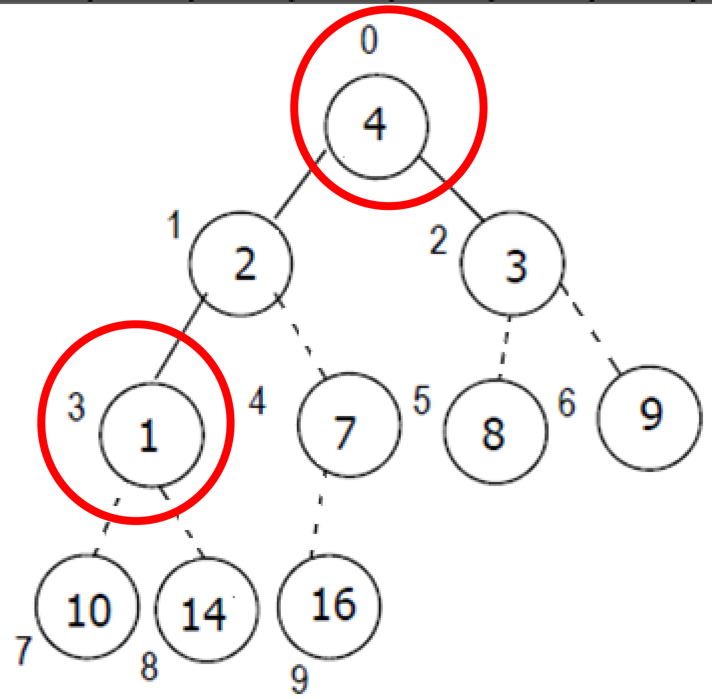
Heap-Sort

0	1	2	3	4	5	6	7	8	9
4	2	3	1	7	8	9	10	14	16



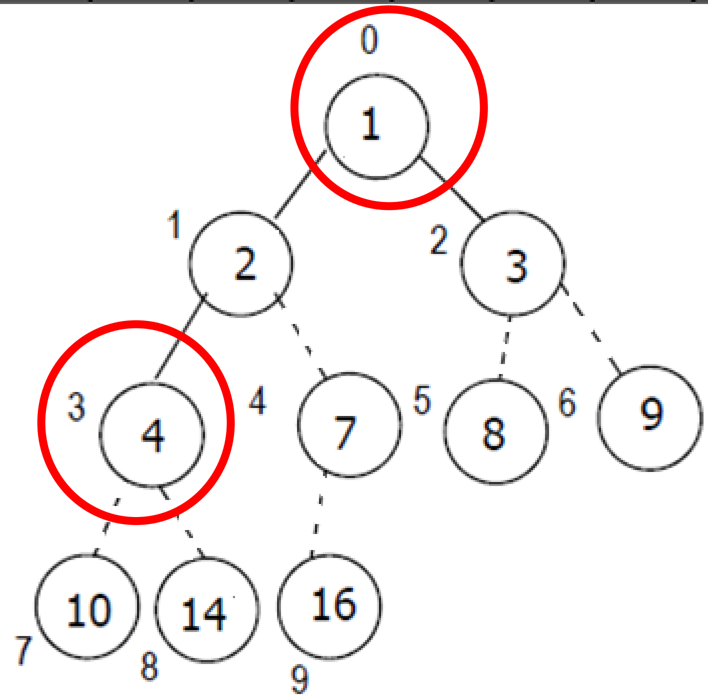
Heap-Sort

0	1	2	3	4	5	6	7	8	9
4	2	3	1	7	8	9	10	14	16



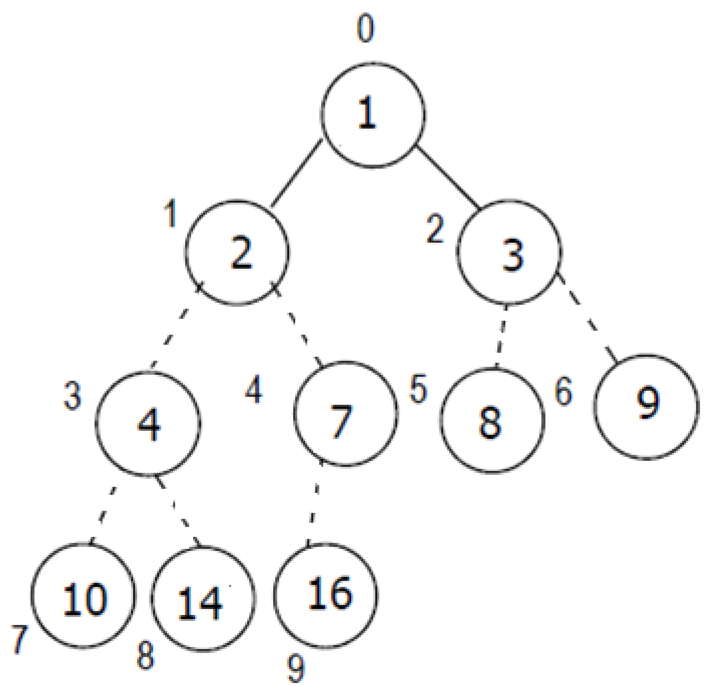
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



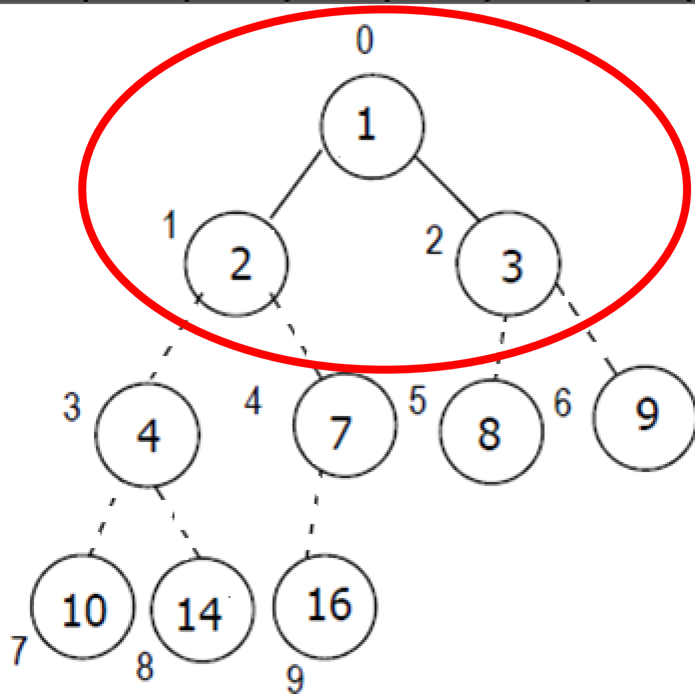
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



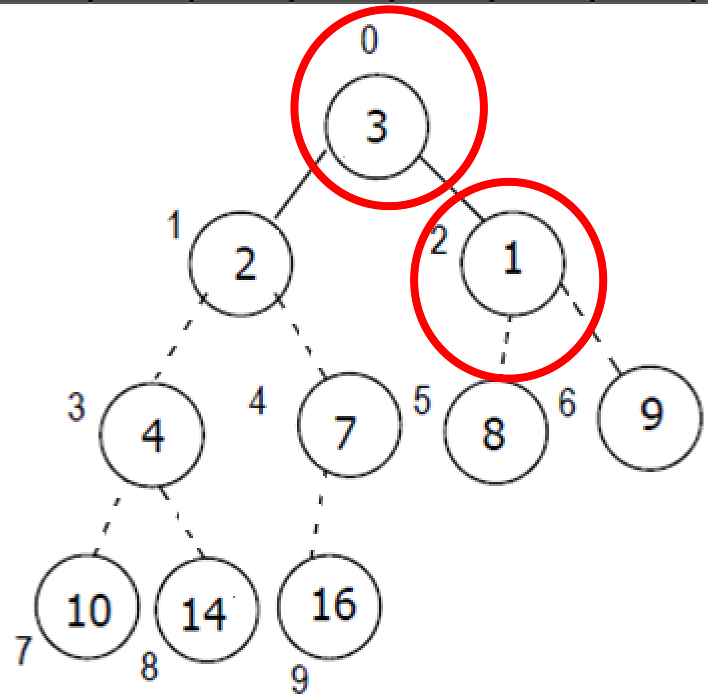
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



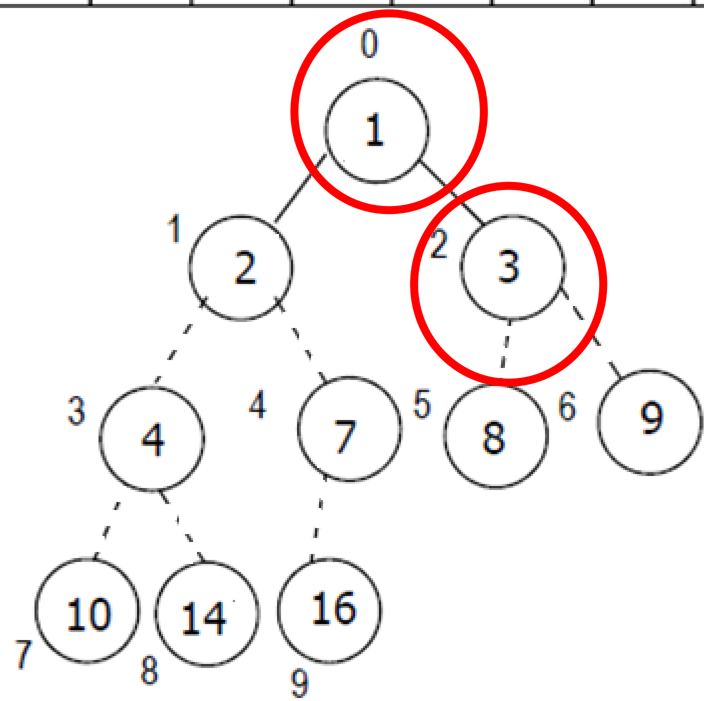
Heap-Sort

0	1	2	3	4	5	6	7	8	9
3	2	1	4	7	8	9	10	14	16



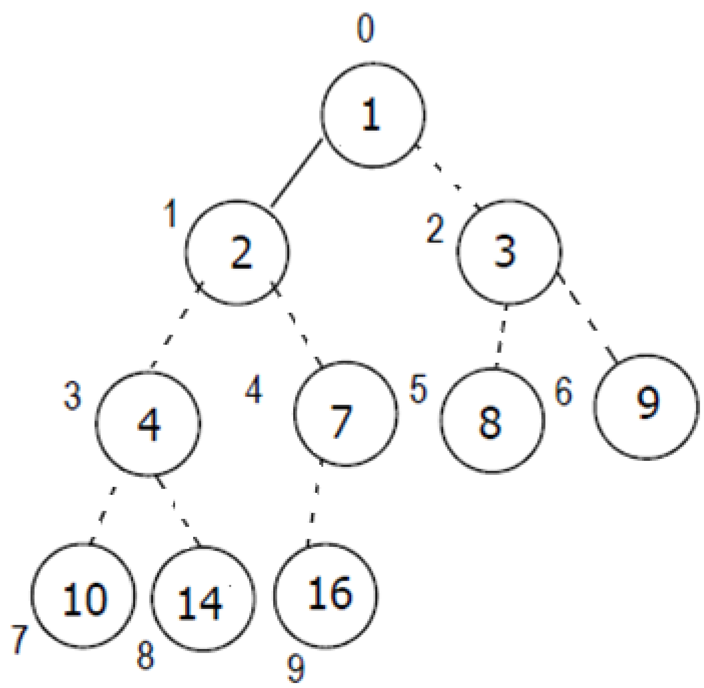
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



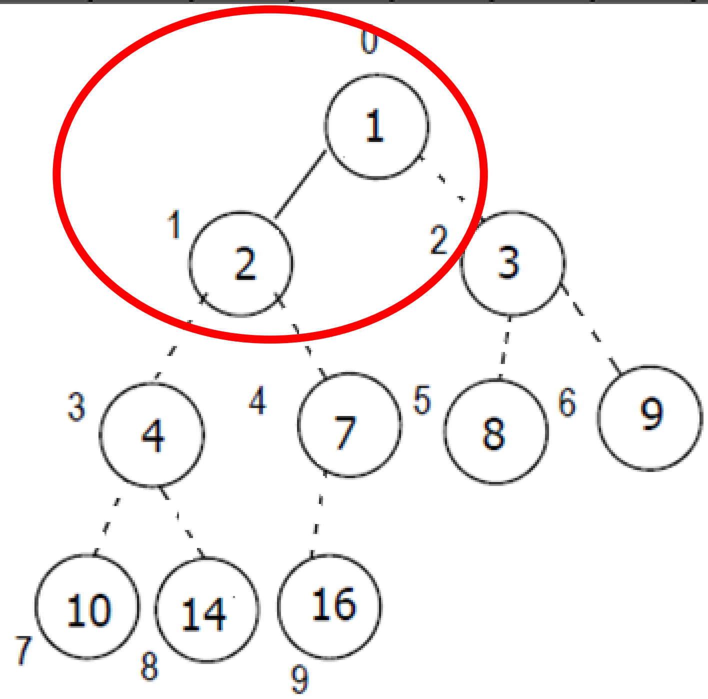
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



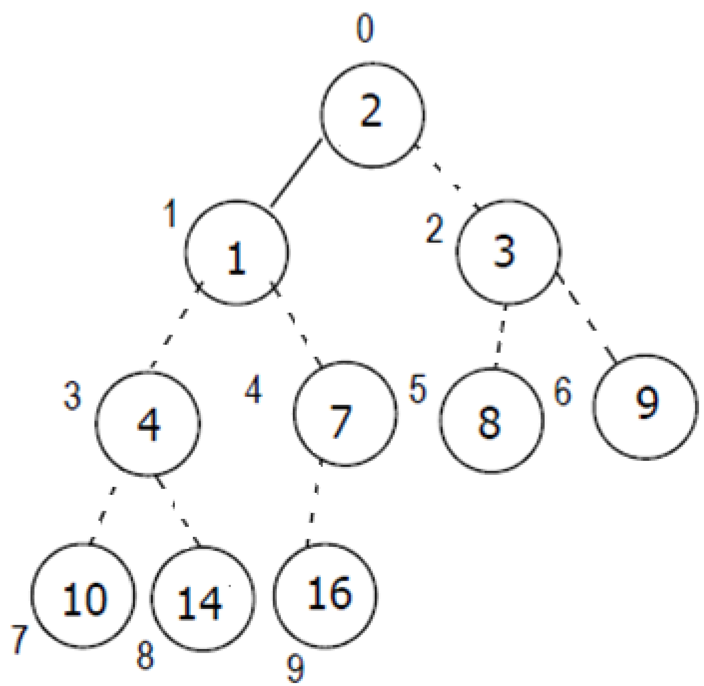
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



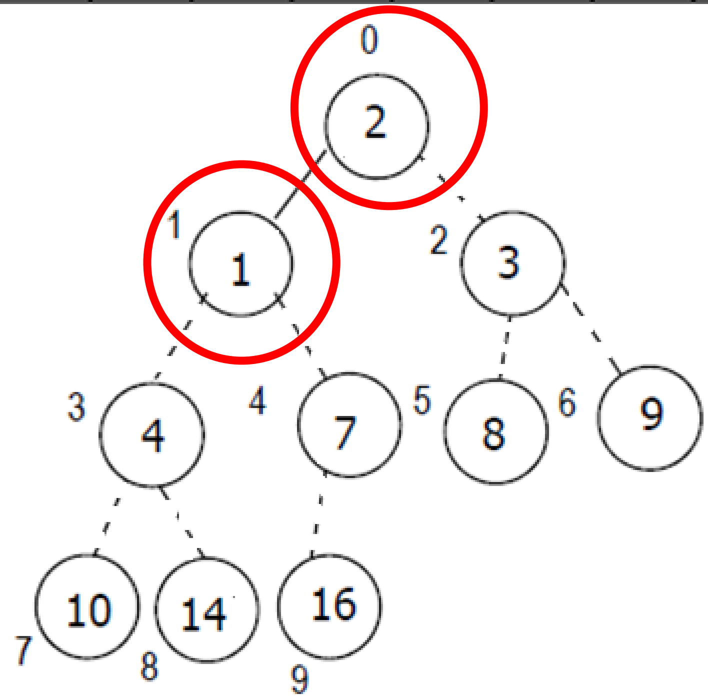
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	1	3	4	7	8	9	10	14	16



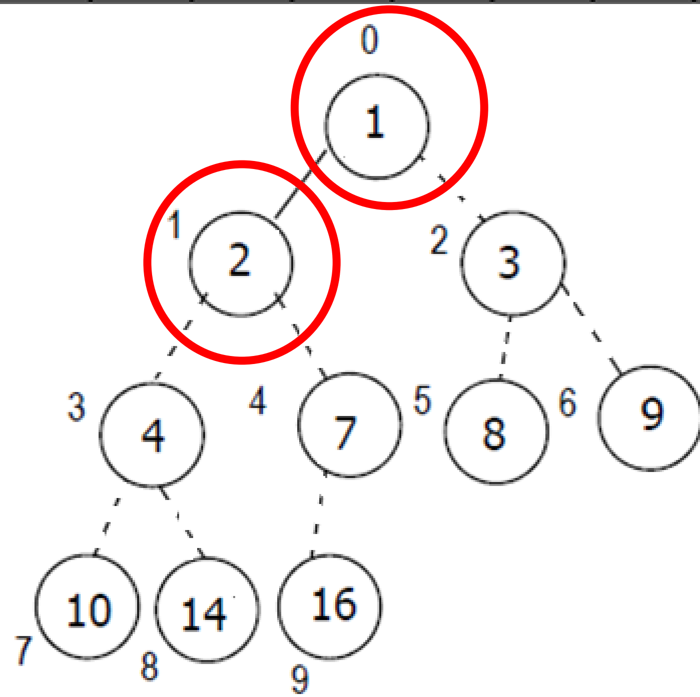
Heap-Sort

0	1	2	3	4	5	6	7	8	9
2	1	3	4	7	8	9	10	14	16



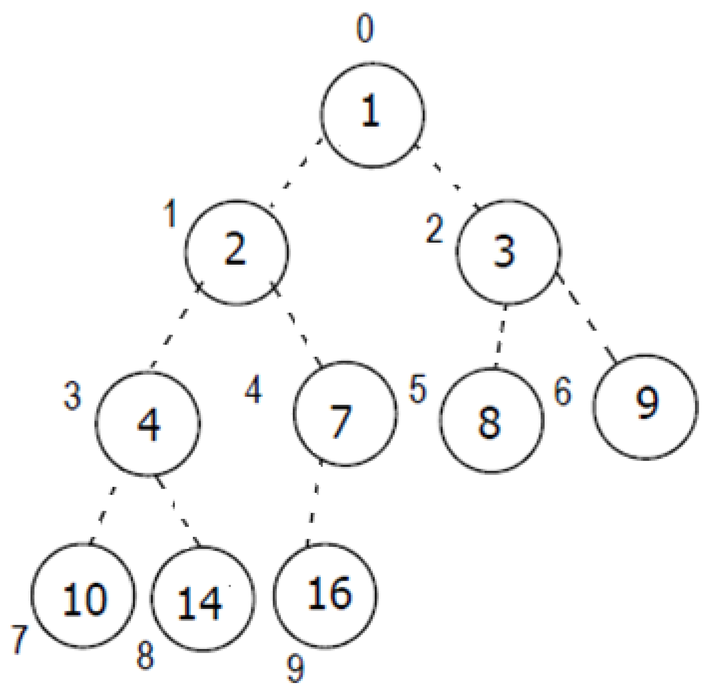
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



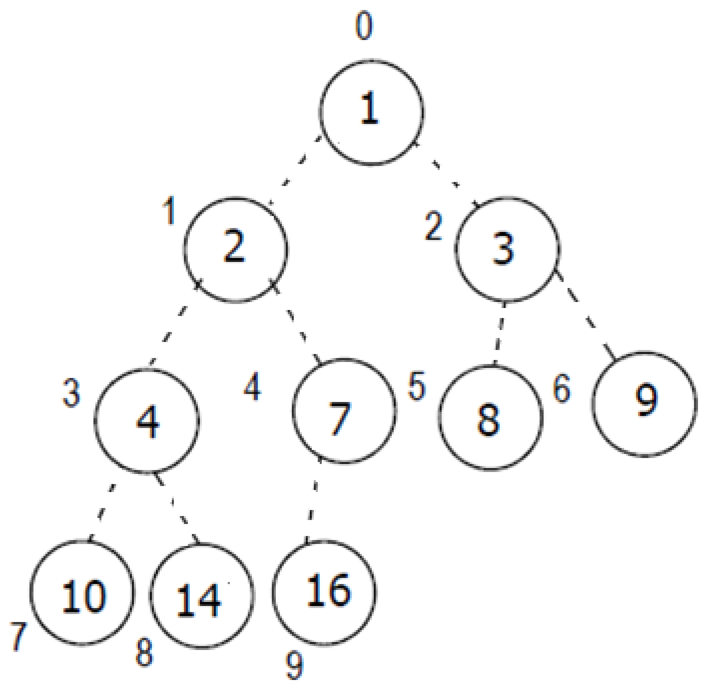
Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



Heap-Sort

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16



Heap-Sort

- O método Heap-Sort tem complexidade $\Theta(n\log n)$
  - É eficiente mesmo quando o vetor já está ordenado
  - Faz n-1 chamadas à função max_heapify, que tem custo O(log n)
  - build_max_heap é O(n)

Ordenação por Intercalação

Ordenação por Intercalação

- Também chamado merge-sort

- Idéia básica: dividir para conquistar
  - Um vetor v é dividido em duas partes, recursivamente
  - Cada metade é ordenada e ambas são intercaladas formando o vetor ordenado
  - Usa um vetor auxiliar para intercalar

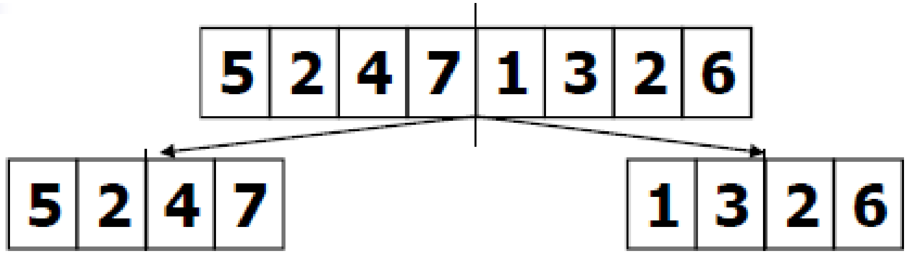
Merge-Sort

5	2	4	7	1	3	2	6

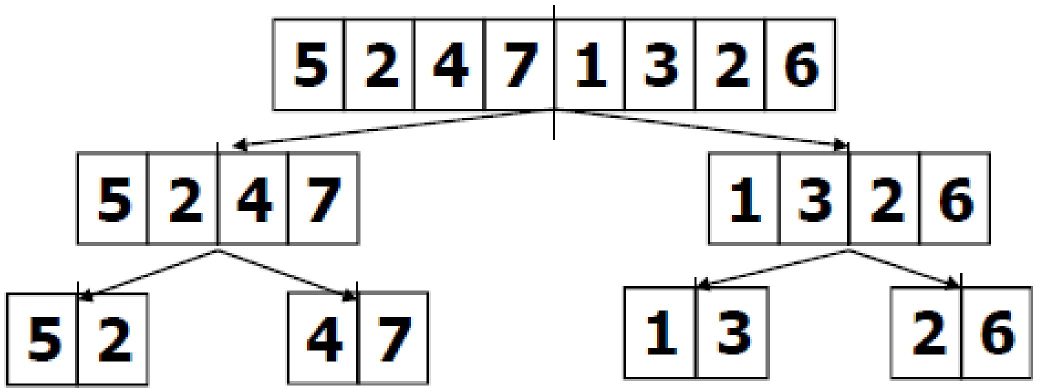
unesp

DEPARTAMENTO DE MATEMÁTICA E COMPUTAÇÃO

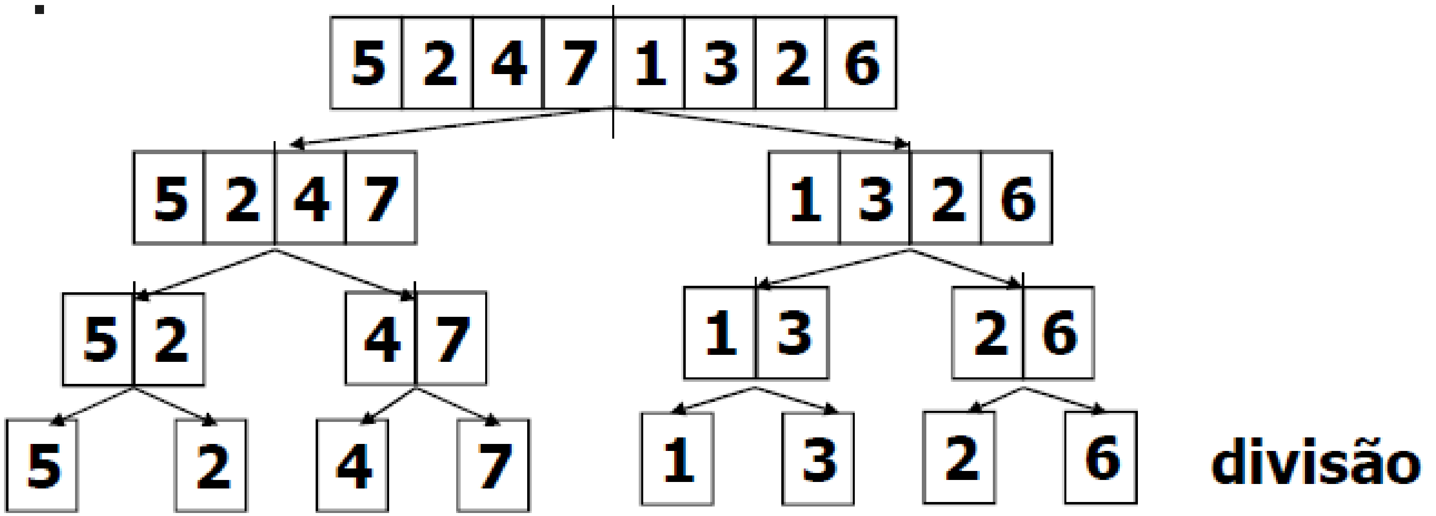
Merge-Sort



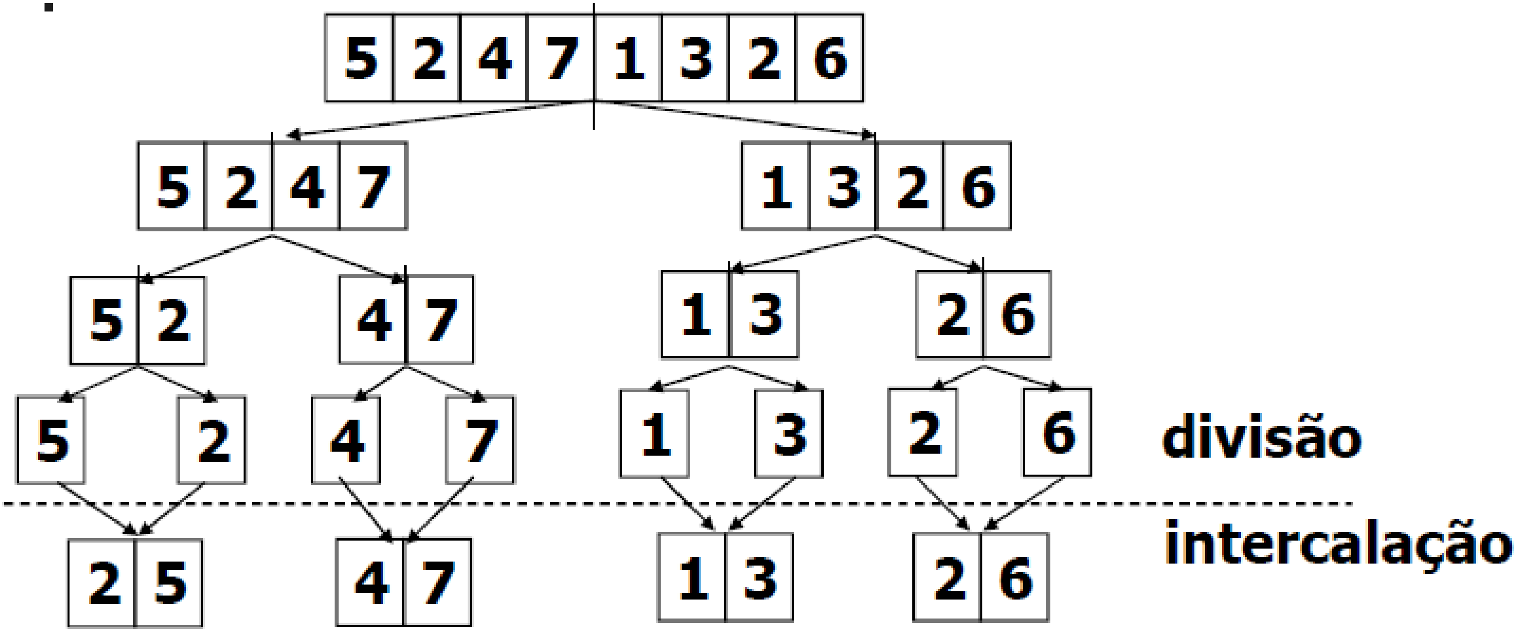
Merge-Sort



Merge-Sort

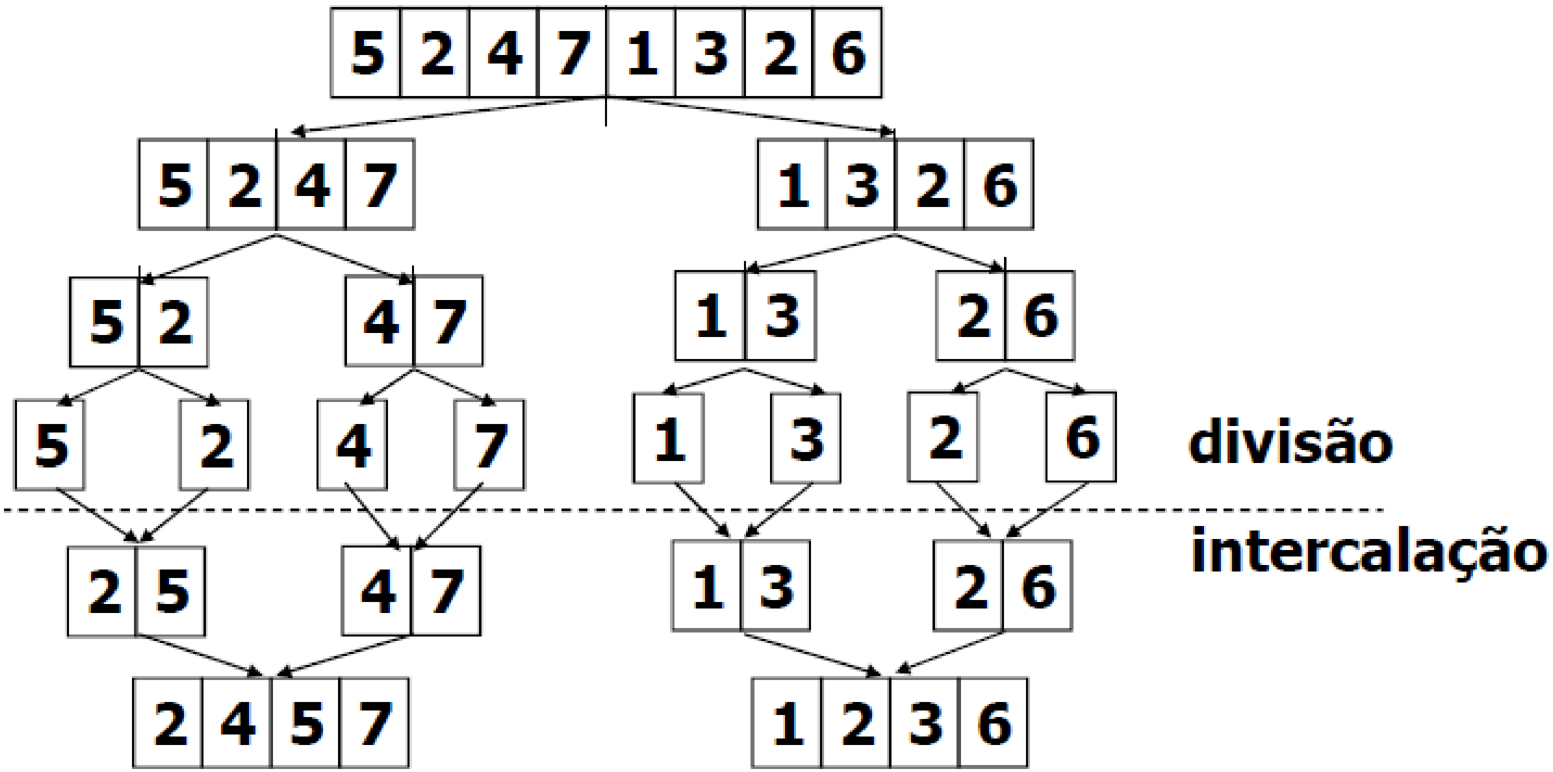


Merge-Sort



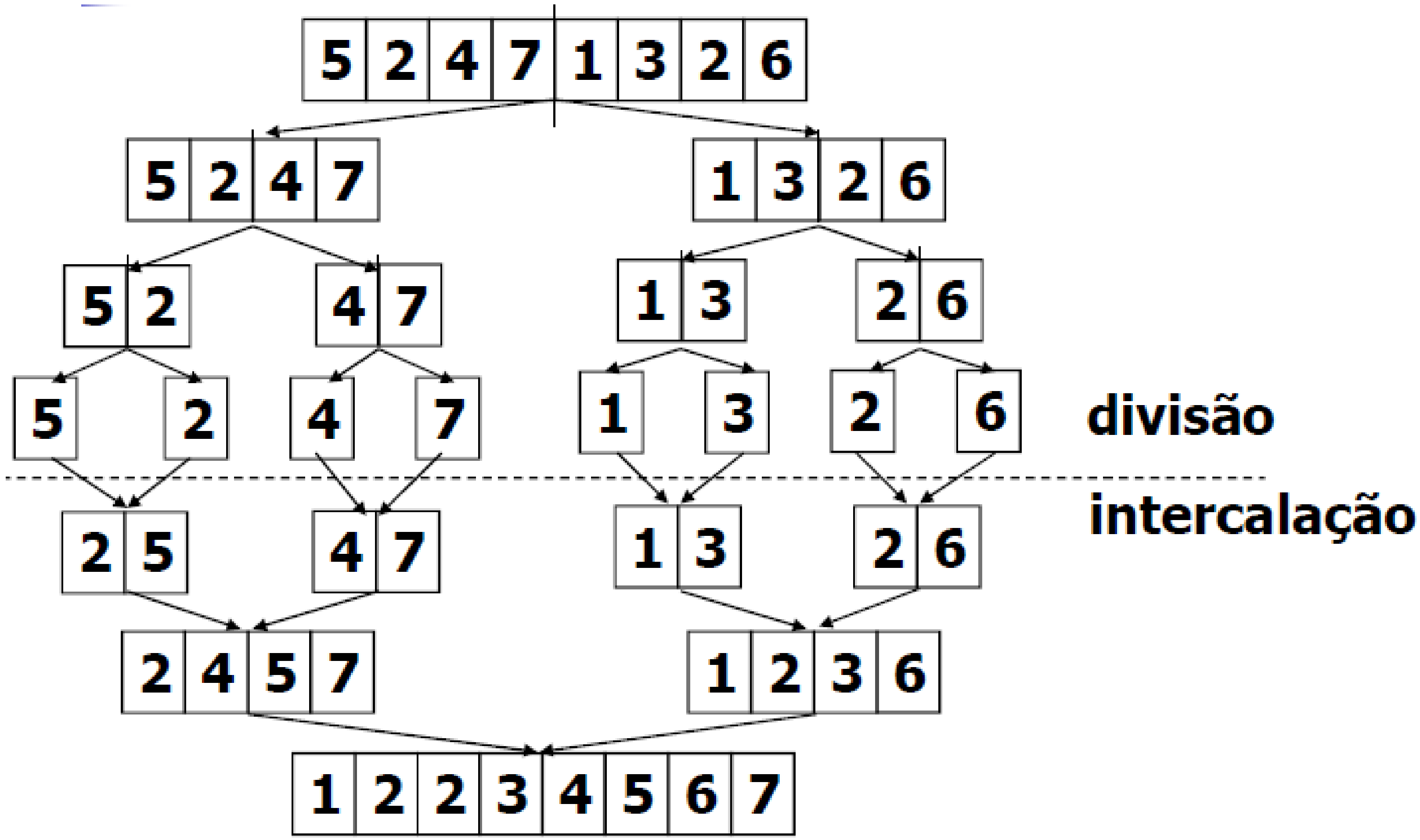
5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
divisão
intercalação
2 5
4 7
1 3
2 6

Merge-Sort



5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
divisão
intercalação
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6

Merge-Sort


5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
divisão
intercalação
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

Merge-Sort

```
void MergeSort(int *A, int e, int d)
{
  int q;

  if (e < d)
  {
    q = floor((e+d)/2); // Determina a metade do vetor
    MergeSort(A,e,q);   // Primeira metade
    MergeSort(A,q+1,d); // Segunda metade
    Merge(A,e,q,d);     // Combina as metades já ordenadas
  }
}
```

- Qual a complexidade?

Merge-Sort

- Complexidade
- Se o particionamento gerar dois sub-conjuntos de tamanho *n/2* temos a recorrência $T(n) = 2T(n/2) + n$
- Caso 2 do teorema mestre
  - $f(n) = \Theta(n)$

$$T(n) = \Theta(n \log n)$$

Comparação Entre os Métodos

Comparação Entre os Métodos

- Ordem aleatória dos elementos
  - O mais rápido recebe valor 1 e o restante é recalculado em função disso

	500	5.000	10.000	30.000
Inserção	11,3	87	161	–
Seleção	16,2	124	228	–
Shellsort	1,2	1,6	1,7	2
Quicksort	1	1	1	1
Heapsort	1,5	1,6	1,6	1,6

Ziviani, 2007

Comparação Entre os Métodos

- Ordem ascendente dos elementos (já ordenado)

	500	5.000	10.000	30.000
Inserção	1	1	1	1
Seleção	128	1.524	3.066	–
Shellsort	3,9	6,8	7,3	8,1
Quicksort	4,1	6,3	6,8	7,1
Heapsort	12,2	20,8	22,4	24,6

Ziviani, 2007

Comparação Entre os Métodos

- Ordem descendente dos elementos

	500	5.000	10.000	30.000
Inserção	40,3	305	575	–
Seleção	29,3	221	417	–
Shellsort	1,5	1,5	1,6	1,6
Quicksort	1	1	1	1
Heapsort	2,5	2,7	2,7	2,9

Ziviani, 2007

DEPARTAMENTO DE MATEMÁTICA E COMPUTAÇÃO

Visualização dos Algoritmos

- http://sorting.at/
  - Visitado em 27/03/2016

Considerações Finais

- Quick-sort é o mais rápido para todos os arranjos com elementos aleatórios
- Heap-sort e quick-sort tem uma diferença constante, sendo o heap-sort mais lento
- Para arranjos pequenos, shell-sort é melhor do que o heap-sort

Considerações Finais

- O método da inserção direta é mais rápido para arranjos ordenados
- O método da inserção direta é melhor do que o método da seleção direta para arranjos com elementos aleatórios
- Shell-sort e quick-sort são sensíveis em relação as ordenações ascendentes e descendentes
- Heap-sort praticamente não é sensível em relação às ordenações ascendentes e descendentes

Adaptado de

Métodos de Ordenação

SCC-201 Introdução à Ciência da Computação II

Rosane Minghim

2010/2011

Baseado no material dos Professores Rudinei Goularte e Thiago Pardo

Referências Bibliográficas

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; (2002). Algoritmos –Teoria e Prática. Tradução da 2ª edição americana. Rio de Janeiro. Editora Campus
- TAMASSIA, ROBERTO; GOODRICH, MICHAEL T. (2004). Projeto de Algoritmos -Fundamentos, Análise e Exemplos da Internet
- ZIVIANI, N. (2007). Projeto e Algoritmos com implementações em Java e C++. São Paulo. Editora Thomson