UNIFEI

O algoritmo Heap Sort, seu funcionamento e desempenho

1º Trabalho Prático- GRUPO 7

Carlos Henrique Reis - 30415
Erick Thomas Alves da Silva - 32068
Fabio Rocha da Silva - 31171
Flavio Wander de A. Batista - 30198
João Paulo M. Oliveira Silva - 24482

Objetivos

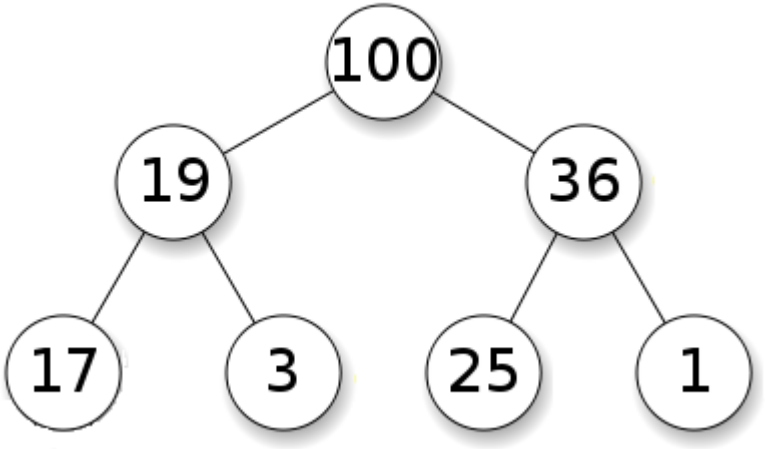
- Implementar o algoritmo de ordenação Heap Sort, analisa-lo e comparar seu desempenho e funcionamento com o algoritmo Selection Sort.

Cronograma

- Heap Sort : Conceito, Algoritmo, Complexidade
- Selection Sort
- Implementação
- Apresentação dos resultados de desempenho
- Conclusão.

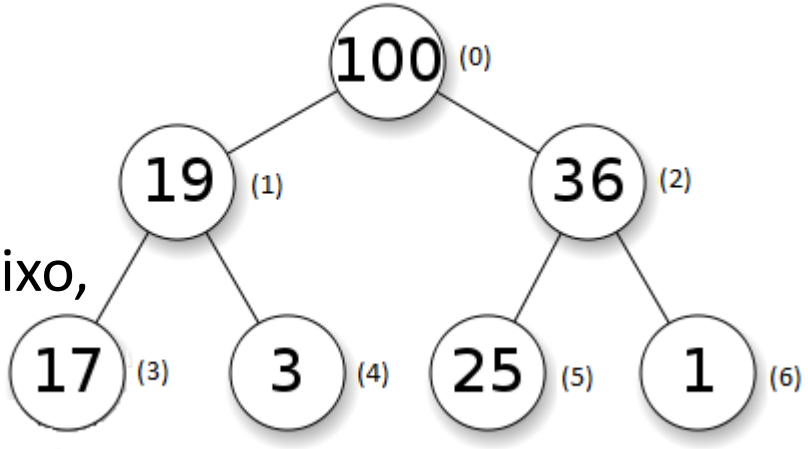
Estrutura Heap

- Estrutura semelhante a uma árvore
- Vários modelos
  - Heap Binário
  - Heap n-ário
- Tipos de Heap
  - Heaps de máximo
  - Heaps de mínimo



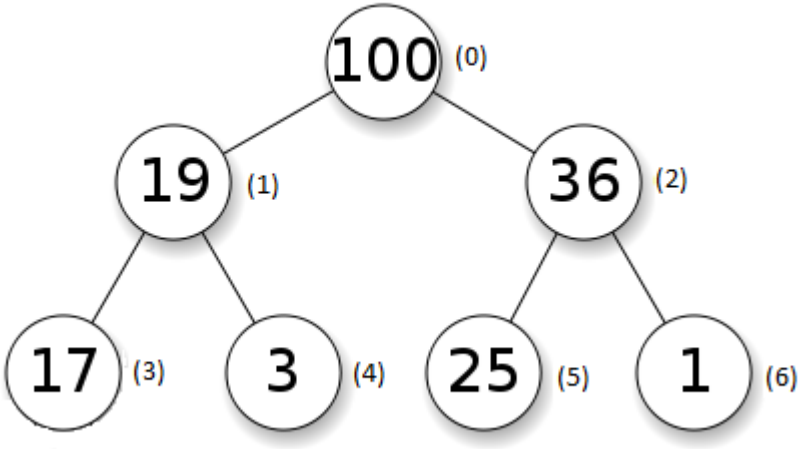
Estrutura Heap

- Regras da estrutura:
  - Pais sempre maiores que filhos
  - Árvore completa, exceto pela última linha, na parte da direita
- Casas são numeradas:
  - Linha por linha, de cima para baixo, pois sua representação clássica é um vetor



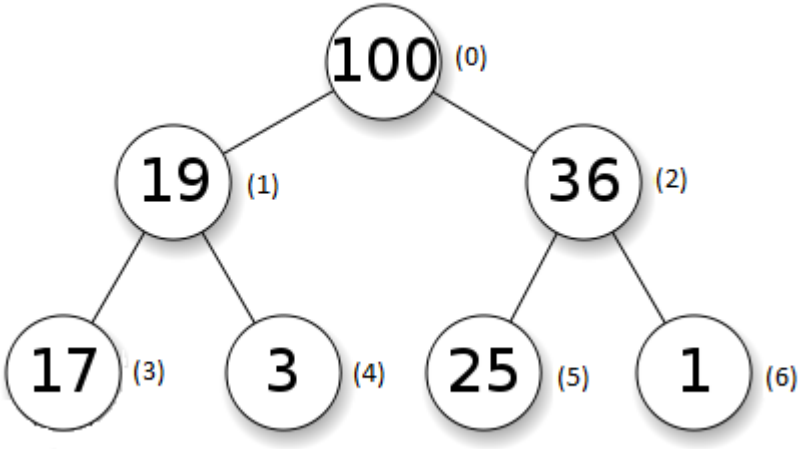
Estrutura Heap

- Pais e filhos:
  - V[n] = Pai
    - V[2n+1] = Filho da esquerda
    - V[2n+2] = Filho da direita



Estrutura Heap

- Operações comuns com Heap:
  - findMax();
    - findMin();
  - insert();
  - extractMax();
    - extractMin();
  - heapify();



Heap Sort

- Algoritmo criado por John Williams em 1964
- Utiliza a abordagem proposta pelo Selection Sort
- Usa de um Heap para a escolha do elemento de maior grau

Heap Sort

- Tenta-se evitar a utilização real de uma árvore.
- Importância do Heap
- O elemento de maior valor sempre estará em V[0]
- Em caso de alteração em V[0], a alteração é rápida

Heap Sort

- Funcionamento do algoritmo:
  1. n = Tamanho do vetor
  2. Transforma o vetor (de 0 até n) em Heap
     - Caso n=0, fim do algoritmo.
  3. Troca V[0] (elemento de maior valor) com V[n]
  4. n = n-1
  5. Volta para passo 2

Heap Sort

- Transformação de vetor em heap (heapify):
  - É preciso fazer o vetor seguir as seguintes diretrizes:
    1. A raiz está em V[0]
    2. O pai de um índice n é (n-1)/2
    3. O filho esquerdo de um índice n é 2n+1
    4. O filho direito de um índice n é 2n+2

Heap Sort

- Transformação de vetor em heap (heapify):
  - Define quem são os filhos de um V[i]
  - Caso o pai seja menor que um filho, ocorre uma troca
  - Esse processo se repete até a posição do filho ser maior ou igual a posição do último vetor, ou seja, chegar no final do vetor
  - Esse processo ocorre entre todos os pais e seus filhos com o auxilio de uma variável aux.

Heap Sort

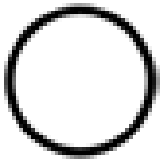
```
void heapify (int *vet, int inicio, int fim){
    int aux = vet[inicio];
    int j = 2*i+1;
    while (j<=fim){
        if (j!=fim && vet[j] < vet[j+1]){
                j++;
        }
        if (aux<vet[j]){
            vet[inicio] = vet[j];
            inicio=j;
            j = 2*i+1;
        }
        else{
            j = fim+1;
        }
    }
    vet[i] = aux;
}
```

Heap Sort

6 5 3 1 8 7 2 4

Heap Sort

6 5 3 1 8 7 2 4

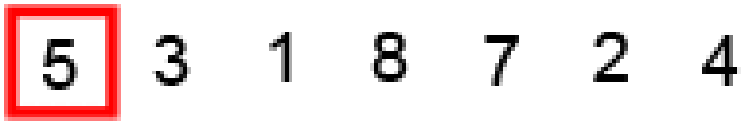


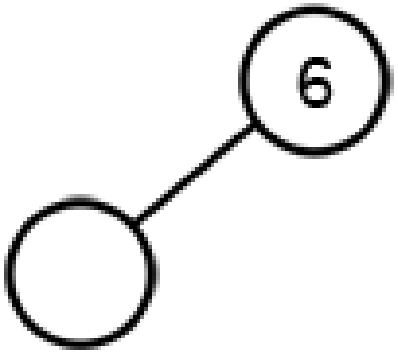
Heap Sort

5	3	1	8	7	2	4

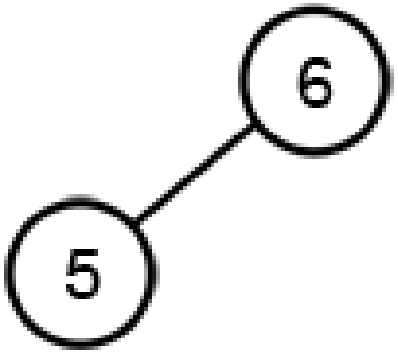
6

Heap Sort

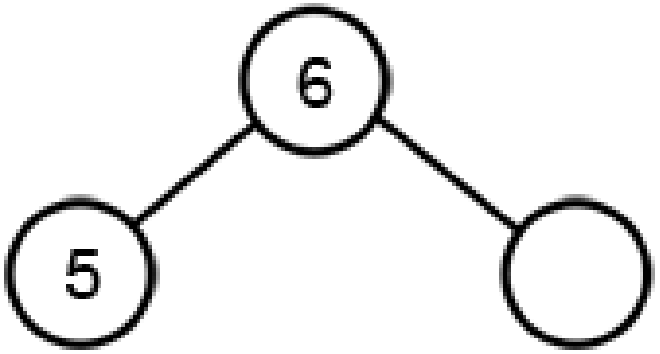




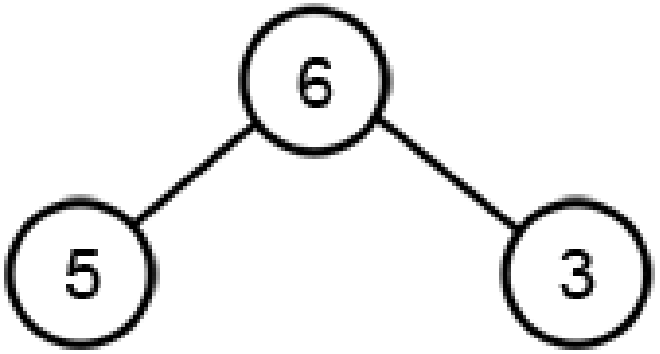
Heap Sort



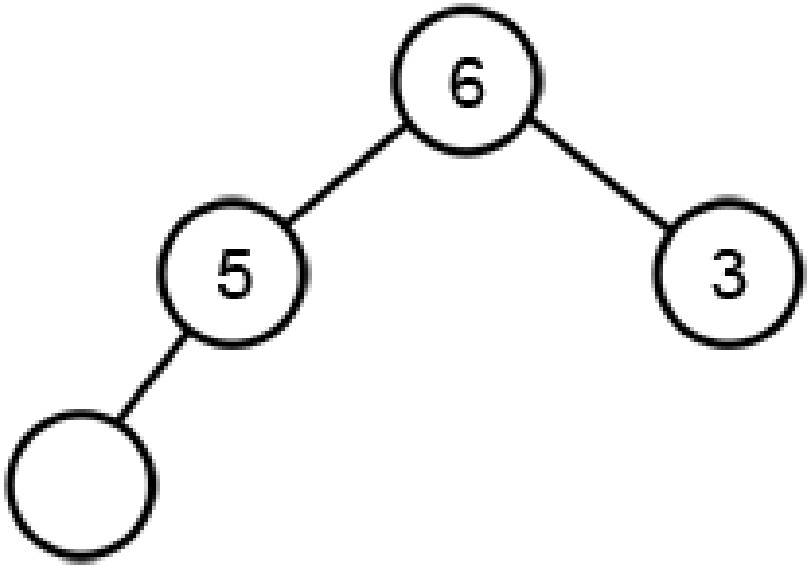
Heap Sort



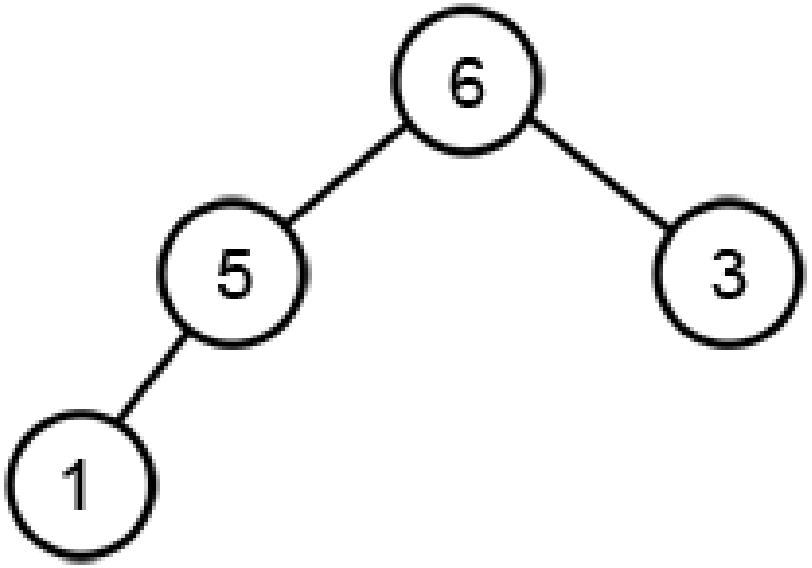
Heap Sort



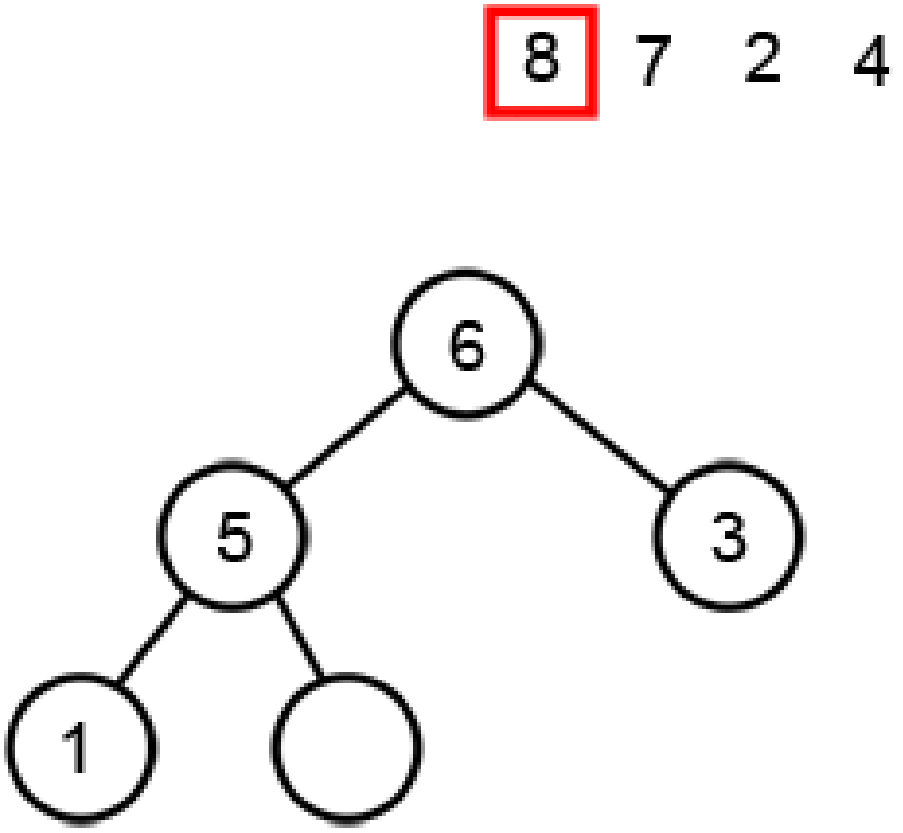
Heap Sort



Heap Sort

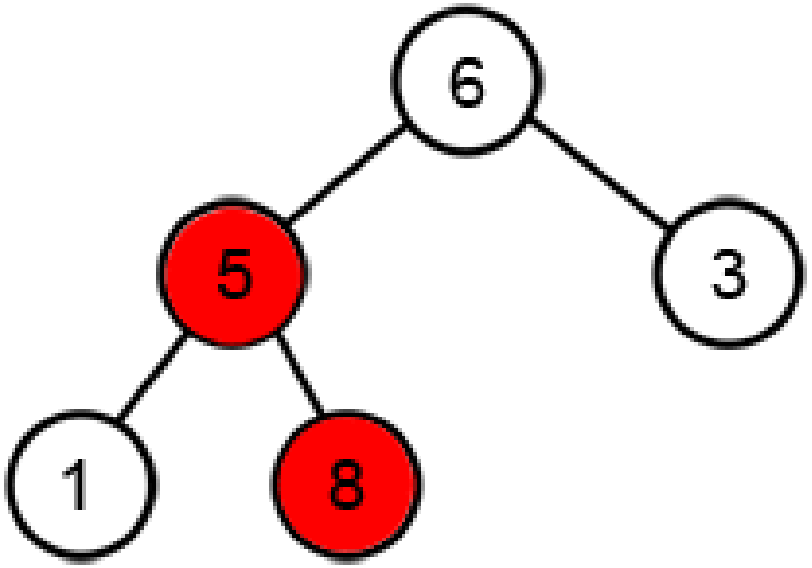


Heap Sort



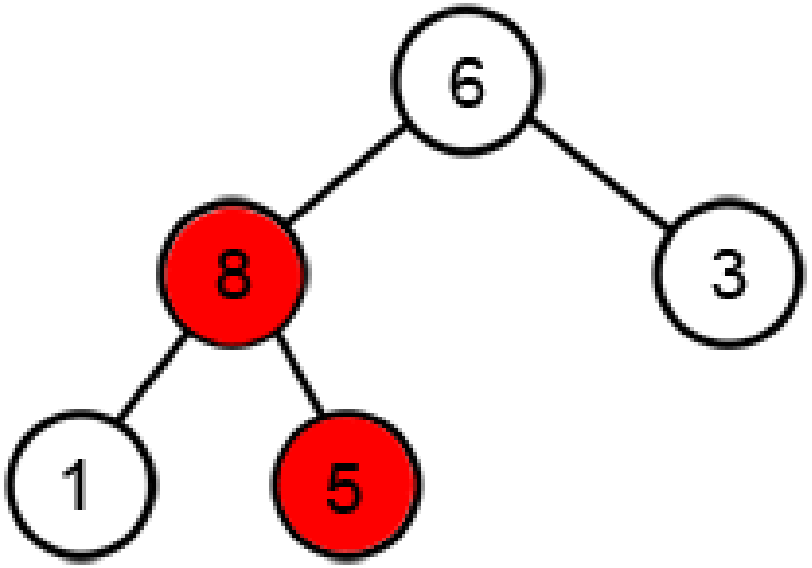
Heap Sort

7 2 4



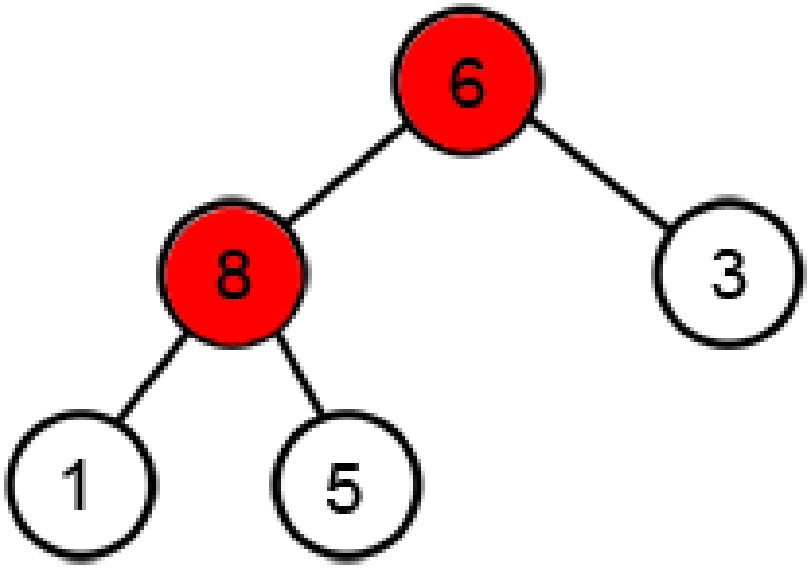
Heap Sort

7 2 4

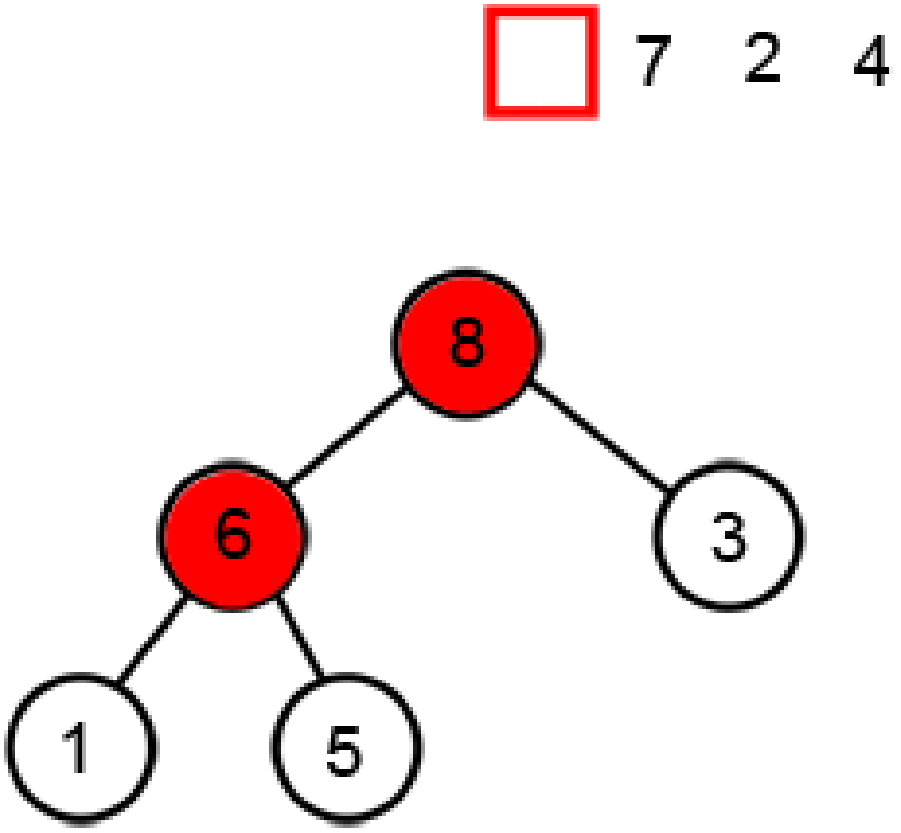


Heap Sort

7 2 4

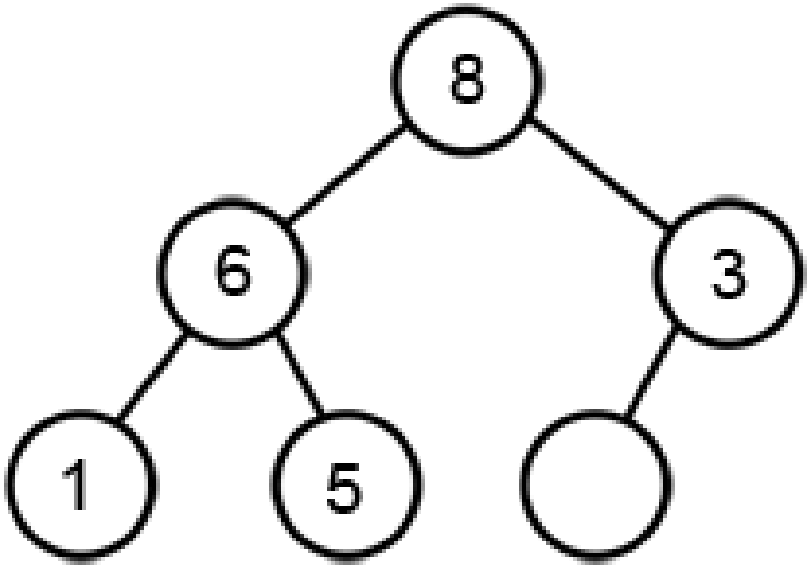


Heap Sort

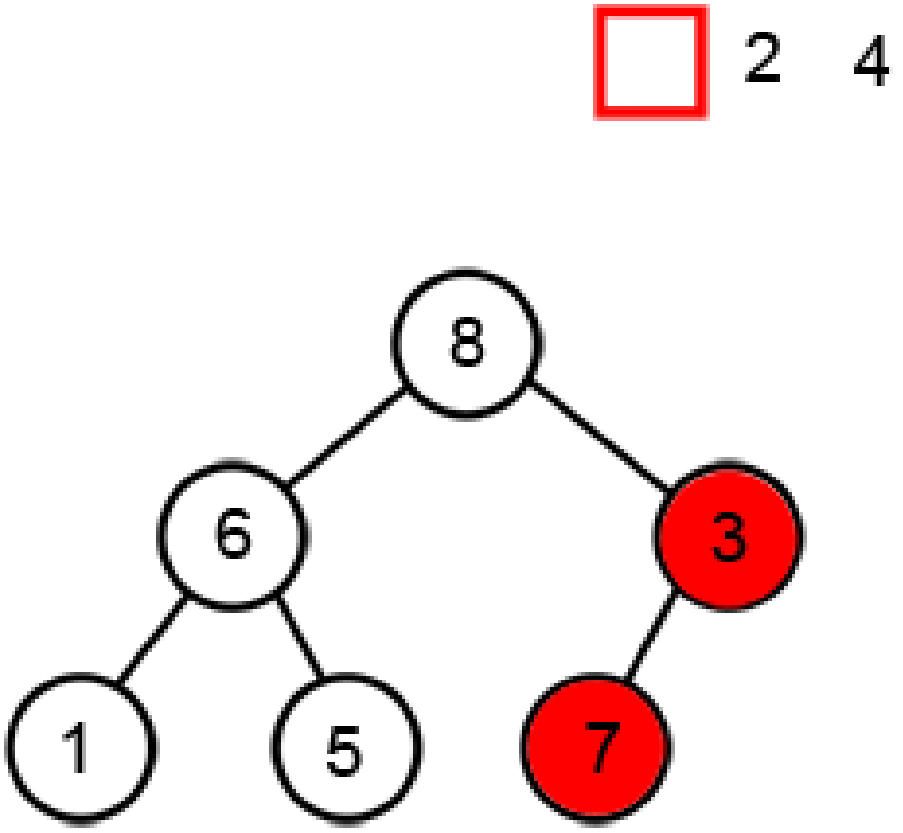


Heap Sort

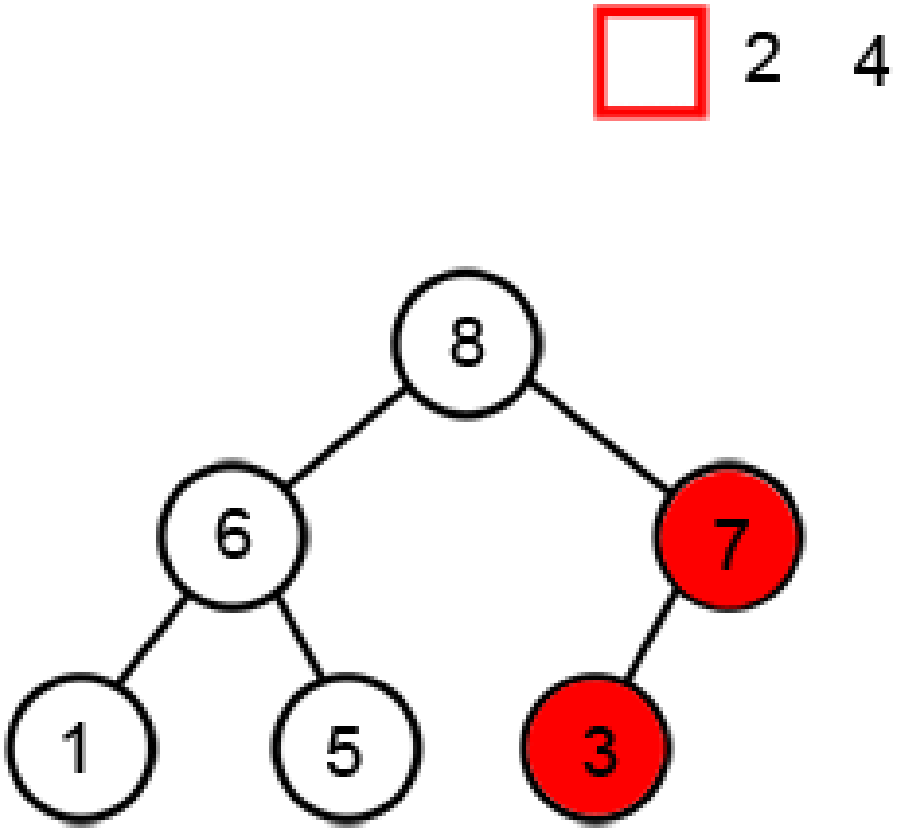
7 2 4



Heap Sort

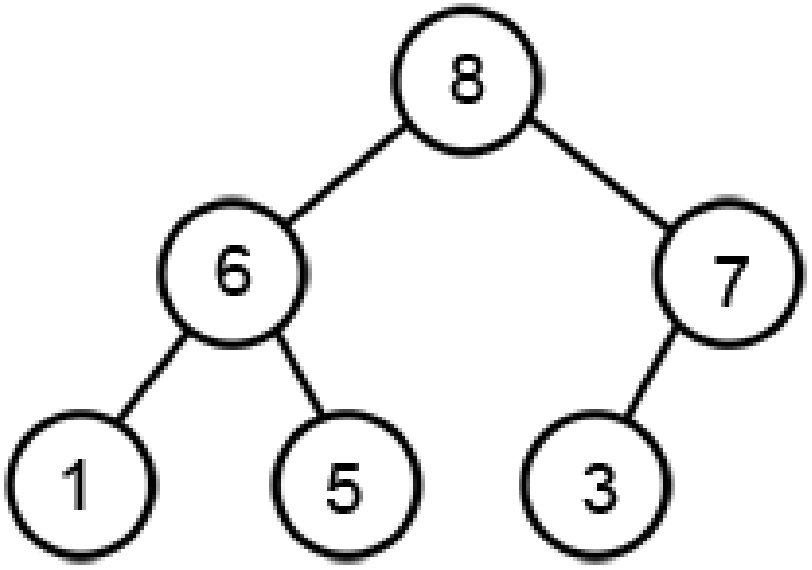


Heap Sort



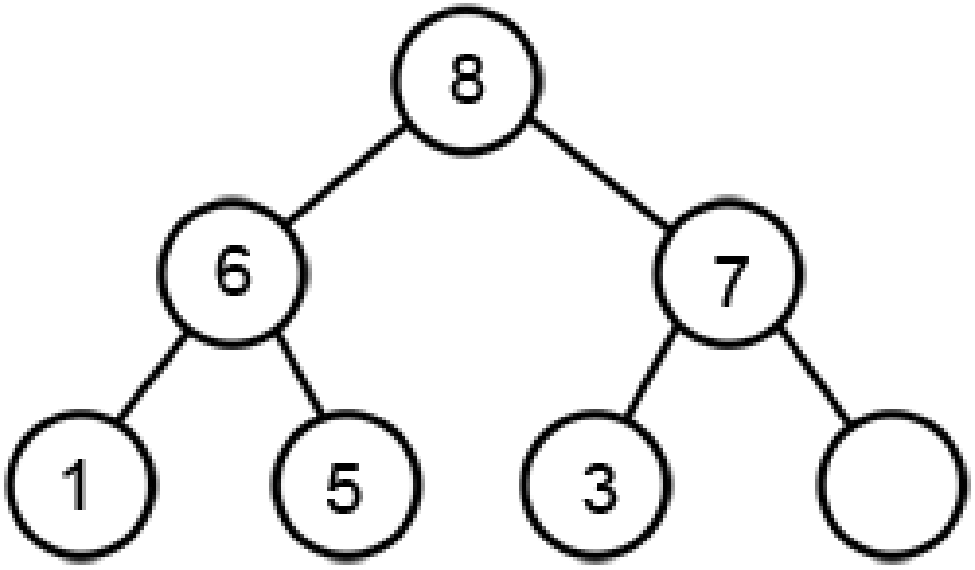
Heap Sort

2 4



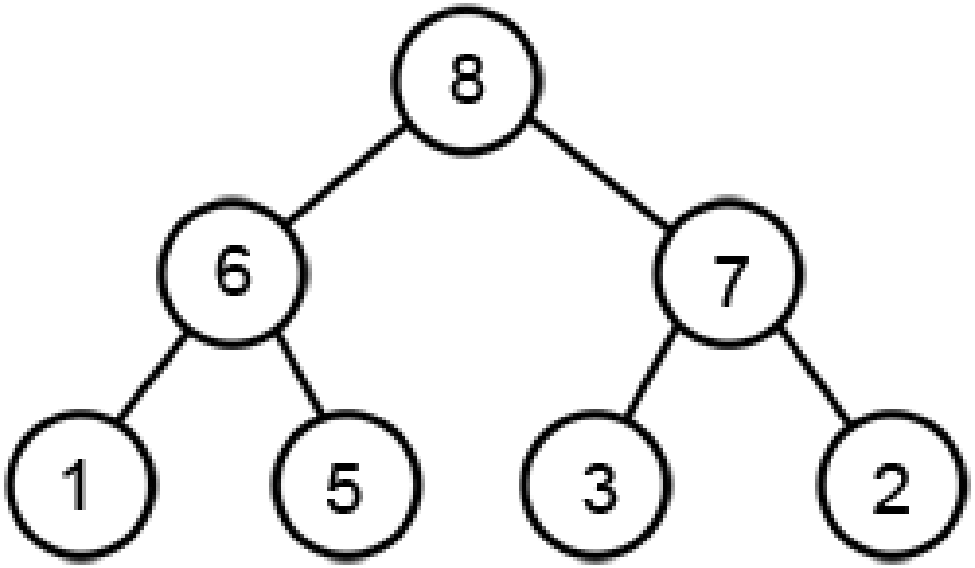
Heap Sort

2 4



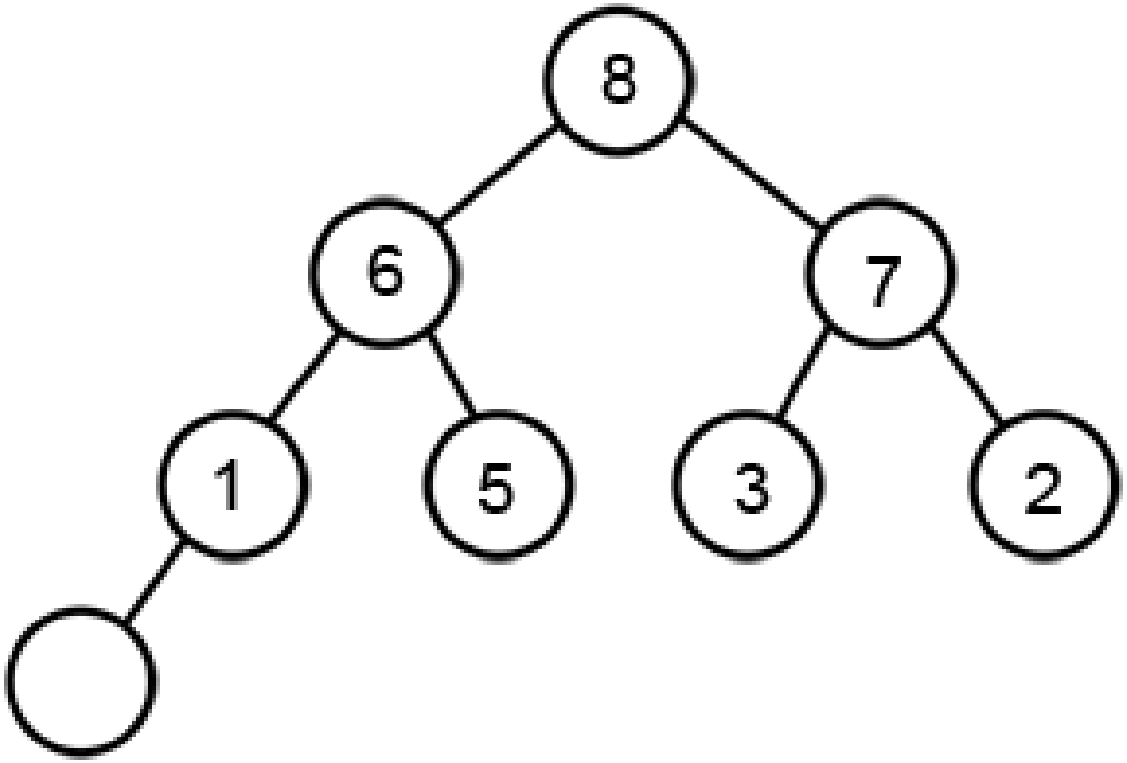
Heap Sort

4

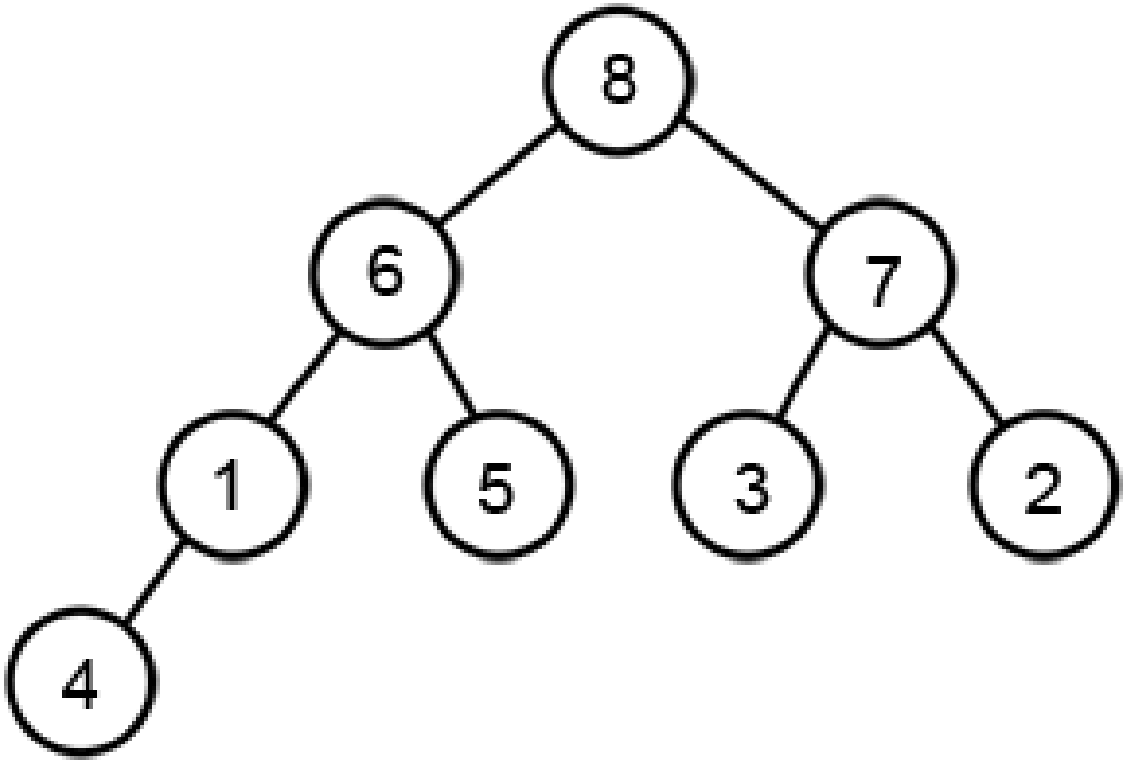


Heap Sort

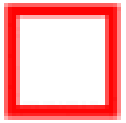
4

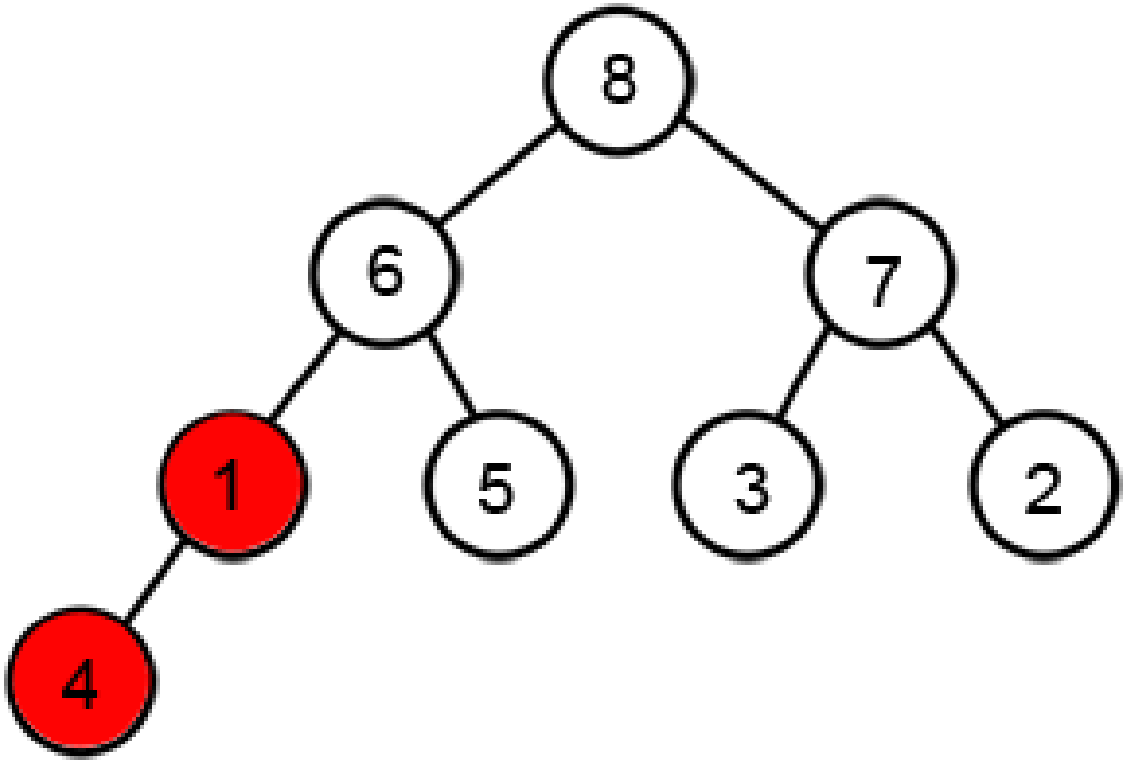


Heap Sort

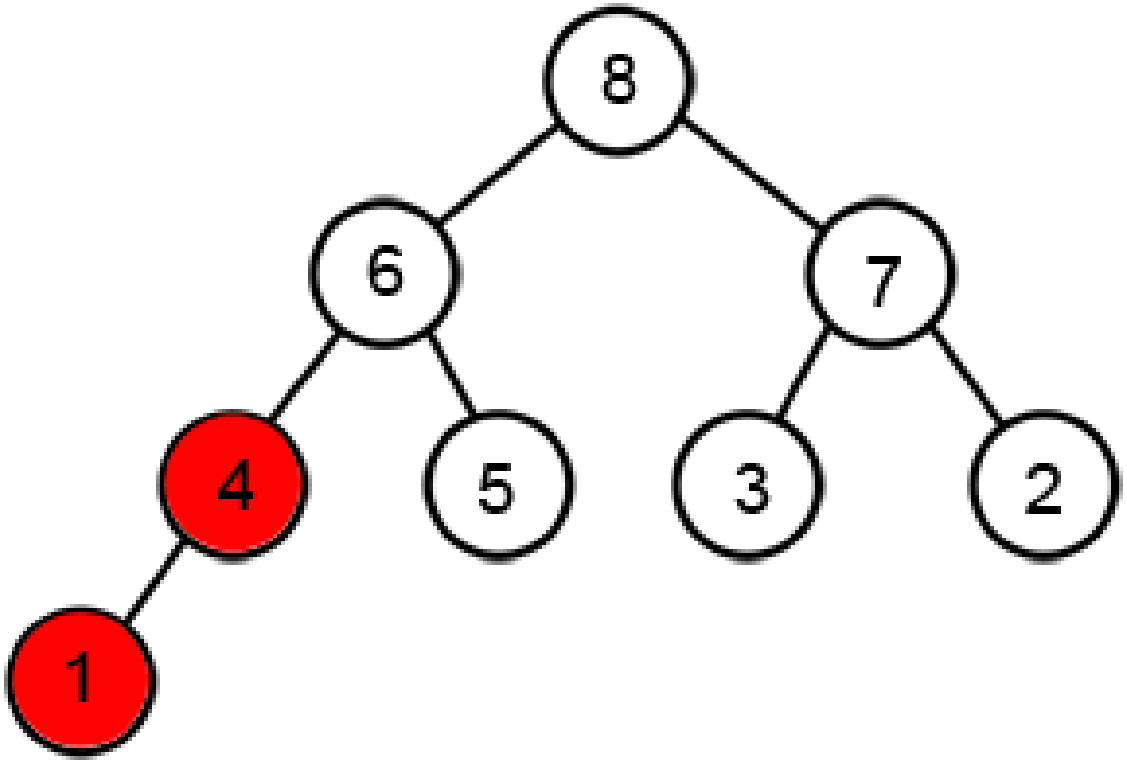


Heap Sort

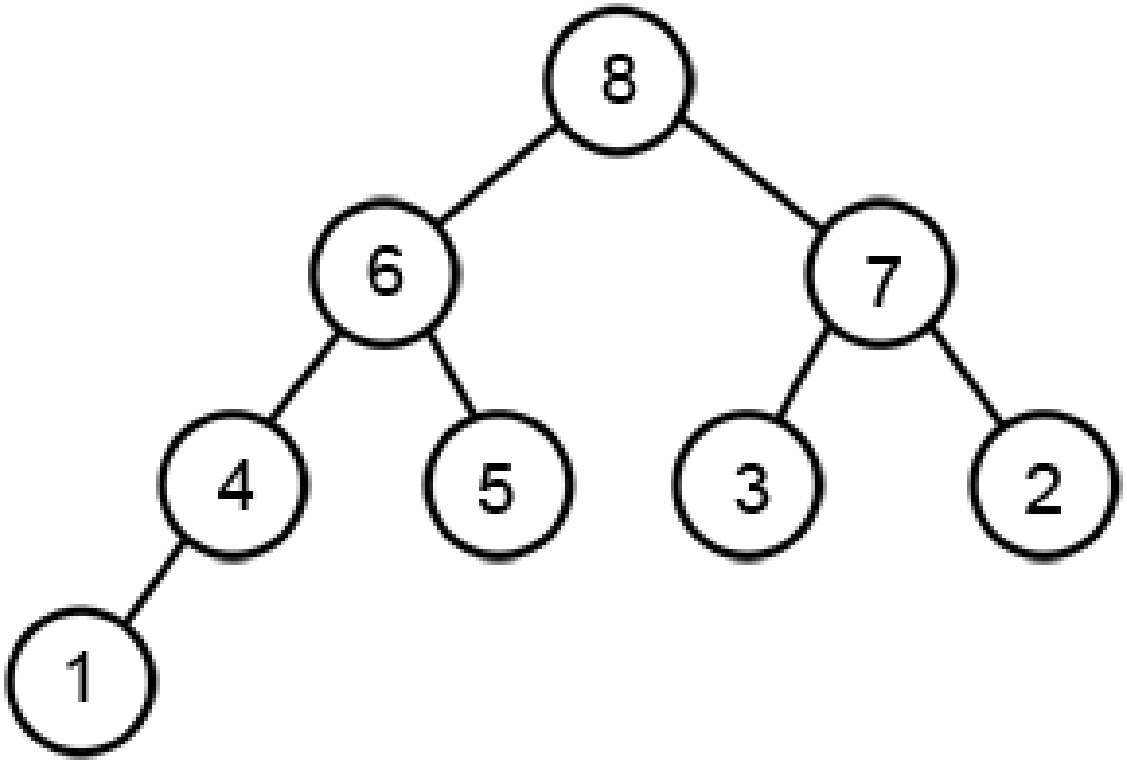




Heap Sort



Heap Sort



Heap Sort

```
void heapSort(int *vet, int n){
    int i, aux;
    for(i=(n-1)/2; i>=0; i--){
        heapify(vet,i,n-1);
    }
    for (i=n-1; i>=1; i--){
        swap(&vet[0],&vet[i]);
        heapify(vet, 0, i-1);
    }
}
```

Heap Sort

- Complexidade O(n log n) no pior e médio caso
- Mesmo tendo a mesma complexidade no caso médio, o algoritmo Heap Sort acaba sendo mais lento que algumas implementações do Quick Sort
- Quick Sort necessita de vetor auxiliar

Heap Sort

- Estabilidade:
  - O algoritmo é considerado instável, pois há a possibilidade de elementos com mesma chave mudar de posição no processo de ordenação

Selection Sort

- Algoritmo no qual Heap Sort foi baseado
- Partições ordenadas e desordenadas

Selection Sort

- Funcionamento do algoritmo:
  1. n = 0
  2. A parte ordenada está em [0,n]
     - Se n = tamanho do vetor, fim do algoritmo
  3. Percorre o vetor, comparando termo a termo, e guarda o índice do menor deles em uma variável (k)
  4. Troca V[k] e V[n+1] de posição
  5. n = n+1
  6. Volta ao passo 2

Selection Sort

||	| 5 | 4 | 3 | 2 | 1 |
| 1 |	| 4 | 3 | 2 | 5 |
| 1 | 4 |	| 3 | 2 | 5 |
| 1 | 2 |	| 3 | 4 | 5 |
| 1 | 2 | 3 |	| 4 | 5 |
| 1 | 2 | 3 |	| 4 | 5 |
| 1 | 2 | 3 | 4 |	| 5 |
| 1 | 2 | 3 | 4 |	| 5 |
| 1 | 2 | 3 | 4 | 5 |	||

Selection Sort

- Alta complexidade:
  - $O(n^2)$ para todos os casos
- Algoritmo instável

Benchmark

- Foram conduzidos testes com vetores de 1000, 5000, 10000, 50000 variáveis.
- Testes feitos com Heap Sort e Selection Sort
- Configurações da máquina:
  - Intel Core i5 3.1GHz
  - 8GB RAM

Benchmark

Tabela 1 – Análise com uso de vetor de 1000 posições

	Desempenho - HS	Desepenho - SS	% Melhora	Chaves comparadas	Registros copiados
Seed 3	205	3266	1593%	16824/500500	9045/1000
Seed 5	241	4065	1686%	16824/500501	9045/1001
Seed 7	245	3174	1295%	16824/500502	9045/1002
Seed 9	262	6155	2349%	16824/500503	9045/1003
Seed 11	504	8231	1633%	16824/500504	9045/1004

Benchmark

Tabela 2 – Análise com uso de vetor de 5000 posições

	Desempenho - HS	Desepenho - SS	% Melhoria	Chaves comparadas	Registros copiados
Seed 3	1204	79332	6589%	107706/12502500	57162/5000
Seed 5	1201	80271	6683%	107706/12502501	57162/5001
Seed 7	1168	79861	6837%	107706/12502502	57162/5002
Seed 9	1235	83015	6721%	107706/12502503	57162/5003
Seed 11	1189	85201	7165%	107706/12502504	57162/5004

Benchmark

Tabela 3 – Análise com uso de vetor de 10000 posições

	Desempenho - HS	Desepenho - SS	% Melhoria	Chaves comparadas	Registros copiados
Seed 3	2633	291494	11970%	235536/50005000	124349/10000
Seed 5	2575	294748	12346%	235536/50005001	124349/10001
Seed 7	2509	294353	12631%	235536/50005002	124349/10002
Seed 9	2531	293736	12505%	235536/50005003	124349/10003
Seed 11	2619	296058	12204%	235536/50005004	124349/10004

Benchmark

Tabela 4 – Análise com uso de vetor de 50000 posições

	Desempenho - HS	Desepenho - SS	% Melhoria	Chaves comparadas	Registros copiados
Seed 3	16576	5790599	35833%	1409935/1250025000	737624/50000
Seed 5	15424	5774493	38338%	1409935/1250025001	737624/50001
Seed 7	16492	5790546	36011%	1409935/1250025002	737624/50002
Seed 9	15310	5779453	38649%	1409935/1250025003	737624/50003
Seed 11	15355	5858401	39053%	1409935/1250025004	737624/50004

Conclusão

- O algoritmo Heap Sort se mostrou mais eficiente em todos os testes
- Quanto maior o número de ítens, maior a diferença
- Apesar de ser mais “complexo” em teoria, o desempenho de Heap Sort é mais eficiente do que Selection Sort
- Complexidade HS = $O(n \log n)$
- Complexidade SS = $O(n^2)$

Considerações finais

UNIFEI - Universidade Federal de Itajubá

Carlos H. Reis - carlos_henreis@unifei.edu.br

Erick T. A. da Silva - erickthomas.alves@hotmail.com

Fabio Rocha da Silva - fabio.r.s.255@gmail.com

Flavio Wander de A. Batista – flaviowander@msn.com

João Paulo M. Oliveira Silva – jpaulo.motta@gmail.com