```
void merge_sort (int *v, int n)
{
  int *v_aux, tam_cadeia=1, j, i;
  if (!(v_aux=(int *)malloc(n*sizeof(int)))) exit(1);
  while (tam_cadeia<n)
  {
    for (i=0; i<n; v_aux[i]=v[i++]);
    j=0;
    while (j<n-tam_cadeia)
    {
       intercala (v, v_aux, j, j+tam_cadeia-1,
       j+tam_cadeia, ((j+2*tam_cadeia-1)<n)?
       (j+2*tam_cadeia-1):(n-1));
       j=j+2*tam_cadeia-1<n?j+2*tam_cadeia:n;
    }
    tam_cadeia*=2;
  }
}
```

```
void intercala (int *v, int *v_aux, int limesqesq, int limesqdir,
int limdiresq, int limdirdir) {
  int deve_continuar=1, esq_menor, IND=limesqesq;
  while (deve_continuar)   {
    esq_menor=v_aux[limesqesq]<v_aux[limdiresq];
    v[IND++]=esq_menor?v_aux[limesqesq++]:
    v_aux[limdiresq++];
    deve_continuar=limesqesq<=limesqdir&&
    limdiresq<=limdirdir;
  }
  while (limesqesq<=limesqdir)
    v[IND++]=v_aux[limesqesq++];
  while (limdiresq<=limdirdir)
    v[IND++]=v_aux[limdiresq++];
}
```

Classificação por Intercalação

Quanto à complexidade do algoritmo apresentado nos slides anteriores, em uma análise superficial, pode ser determinada se considerarmos o seguinte: tam_cadeia, atualizada por duplicações sucessivas, assume valores do conjunto [1,2,4,8,16...$\lfloor n/2.0 \rfloor$], sendo a repetição principal controlada pela condição tam_cadeia <= n/2, o que a qualifica como **O (log n)**. Em cada passagem, cada elemento do vetor é copiado uma vez e intercalando uma vez (na função intercala).

Classificação por Intercalação

O *enquanto* intermediário, i.e, o segundo *enquanto* do algoritmo principal, apenas distribui o processamento sobre os sucessivos subvetores. Isto acarreta no máximo **2n** movimentos de dados em cada fase. Logo, o procedimento todo é da ordem de **2n log n**, ou seja, **O (n log n)**.

Como uma análise mais profunda fugiria do escopo desta disciplina, ficaremos apenas neste nível de analise.

Métodos de Ordenação que utilizam o Princípio de Distribuição

Exemplo: considere o problema de ordenar um baralho com 52 cartas não ordenadas. Suponha que ordenar o baralho implica em colocar as cartas de acordo com a ordem

A < 2 < 3 < ... < 10 < J < Q < K < ♣ < ♦ < ♥ < ♠

Para ordenar por distribuição, basta seguir os passos abaixo:

1. Distribuir as cartas em 13 montes, colocando em cada monte todos os ases, todos os dois, todos os três, ..., todos os reis.
2. Colete os montes na ordem acima (*as* no fundo, depois os *dois*, etc.), até o rei ficar no topo.

Métodos de Ordenação que utilizam o Princípio de Distribuição

3. Distribua novamente as cartas abertas em 4 montes, colocando em cada monte todas as cartas de paus, todas as cartas de ouros, todas as cartas de copas e todas as cartas de espadas.
4. Colete os montes na ordem indicada acima (paus, ouros, copas e espadas).

Métodos baseados no princípio de distribuição são também conhecidos como ordenação digital, radixsort ou bucketsort. Neste caso não existe comparação entre chaves.

Métodos de Ordenação que utilizam o Princípio de Distribuição

Outro exemplo: As antigas classificadoras de cartões perfurados também utilizam o princípio da distribuição para ordenar uma massa de cartões.

Dificuldades de implementar este método:

- Problema de lidar com cada monte.
- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode se tornar proibitiva.

Heapsort

Heapsort é um método de ordenação cujo princípio de funcionamento é o mesmo utilizado para a ordenação por seleção.

Selecione o maior (ou menor) item do vetor e a seguir troque-o com o item que está na última (ou primeira) posição do vetor; repita estas duas operações com os **n - 1** itens restantes; depois com os **n - 2** itens; e assim sucessivamente.

Heapsort

O custo para encontrar o maior (ou o menor) item entre **n** itens e de **n - 1** comparações.

Este custo pode ser reduzido?

SIM.

Este custo pode ser reduzido através da utilização de uma estrutura de dados chamada de fila de prioridades.

Fila de Prioridades

Fila:

Sugere espera por algum serviço.

Prioridade:

Sugere que o serviço será fornecido com base em um critério.

Fila de Prioridade:

Conjunto de elementos com o comportamento elemento-de-maior-valor (menor-valor) é o primeiro a abandonar o conjunto de elementos.

Aplicações de Filas de Prioridades

- Sistemas operacionais usam filas de prioridades, onde as chaves representam o tempo em que eventos devem ocorrer.

- Alguns métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.

- Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal do computador por uma nova página.

Fila de Prioridades

Operações Principais:

1. Construir uma fila de prioridades a partir de um conjunto com **n** itens;
2. Retirar o item com maior prioridade;
3. Restaurar a fila de prioridades.

Forma de implementação:

Árvore binária

Árvore Binária

Considerando as características de uma árvore binária de busca um aluno atento chegará a seguinte reflexão:

As chaves poderiam ser inseridas, uma a uma, em uma árvore binária de busca;

Após a inserção de todas as chaves a árvore poderia ser percorrida, por exemplo, em in-ordem e as chaves seriam obtidas em ordem crescente.

Árvore Binária

Desvantagens da utilização de uma árvore binária de busca:

- Necessidade de área de memória adicional para o armazenamento da árvore;
- O que aconteceria se as chaves já se encontrarem em ordem ou ordem inversa?
  - Seria gerada um árvore degenerada. O que significa que para a inserção do ***i***-ésimo elemento seriam requeridas ***i-1*** comparações, o que, praticamente, elimina a vantagem de se utilizar uma árvore no processo.

Heap Sort

As deficiências da classificação utilizando árvore binária ordenada são eliminadas no método denominado **heap sort**.

O heap sort é um método *in situ* de complexidade constante, independente da ordem da entrada.

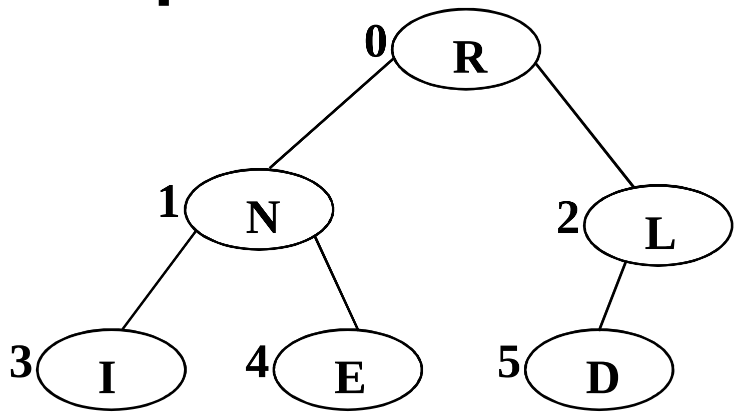
Um heap decrescente de tamanho **n** é implementado utilizando-se uma árvore binária quase completa representada sequencialmente, com a característica de que todo nó possui um valor maior ou igual aos valores armazenados em seus filhos, caso estes existam.

Heap Sort

Árvore Binária quase completa é uma árvore binária onde:

1. Cada folha na árvore está no nível **d** ou no nível **d-1**;
2. Para cada nó **nd** na árvore com um descendente direito no nível **d**, todos os descendentes esquerdos de **nd** que forem folhas estiverem também no nível **d**.

Exemplo: Árvore binária quase completa

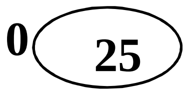


Índices	0	1	2	3	4	5
Valores	R	N	L	I	E	D

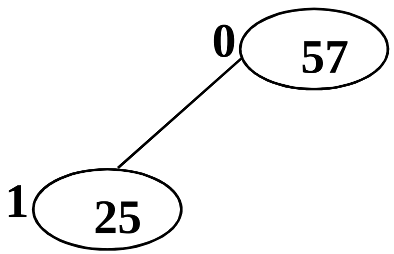
Relação: info[j] ≤ info[(j-1)/2]
para 0 ≤ ((j-1)/2) < j ≤ n-1

Para uma melhor compreensão vamos analisar o processo de construção de um heap partindo de um conjunto de **n** elementos (chaves) sobre o exemplo onde o vetor de chaves originalmente é:

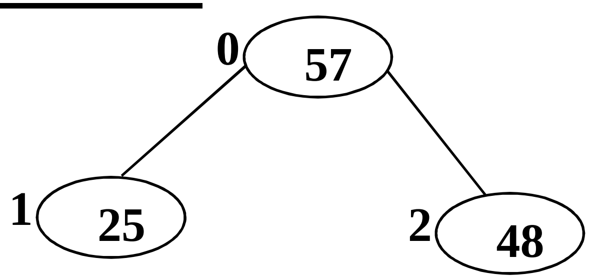
25 57 48 37 12 92 86 33



57 25 48 37 12 92 86 33



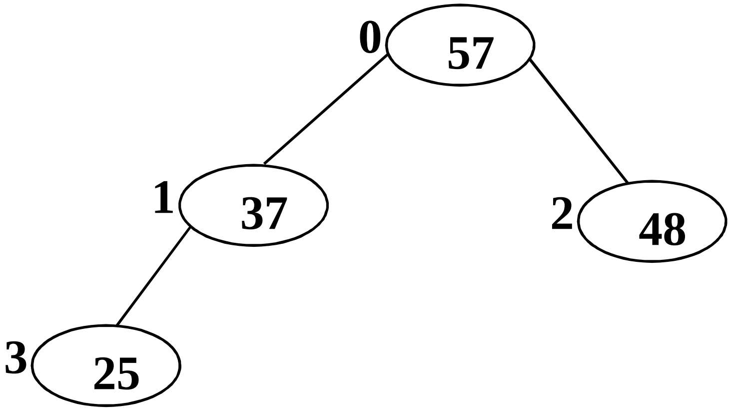
57 25 48 37 12 92 86 33



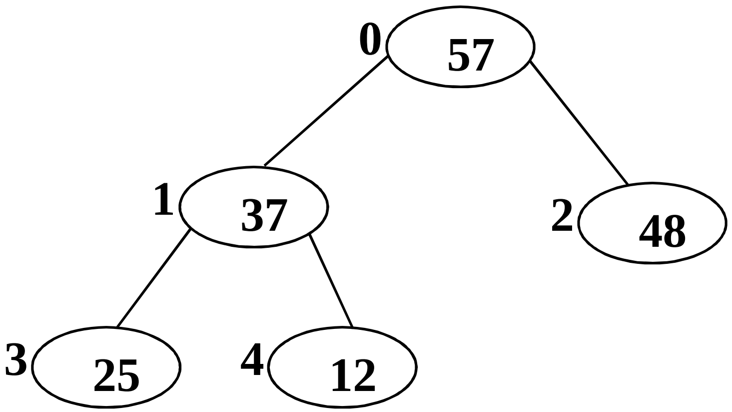
Heap Sort

<u>57 25 48</u> 37 12 92 86 33

<u>57 37 48 25</u> 12 92 86 33



<u>57 37 48 25 12</u> 92 86 33

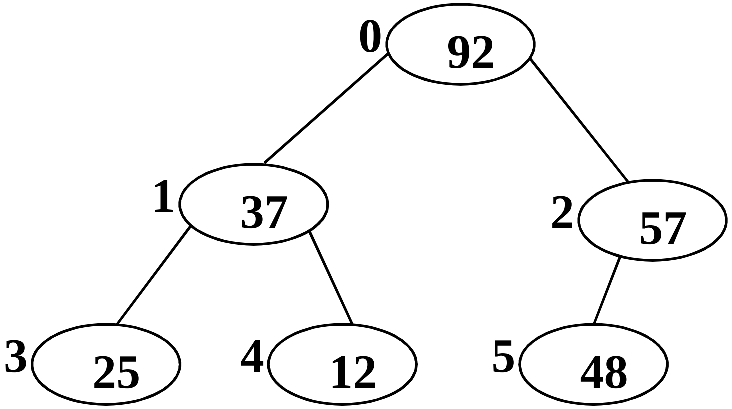


Heap Sort

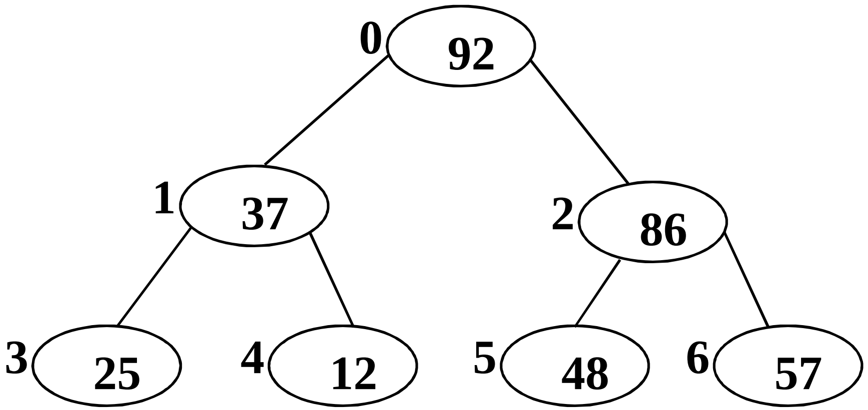
57 37 48 25 12 92 86 33

57 37 92 25 12 48 86 33

92 37 57 25 12 48 86 33

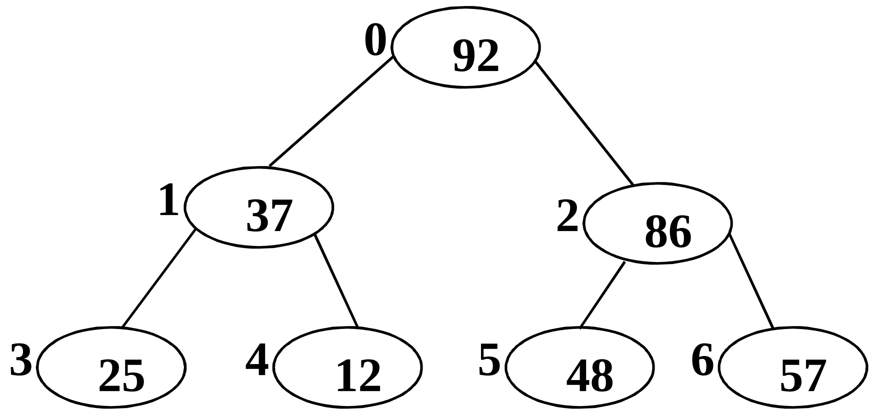


92 37 86 25 12 48 57 33

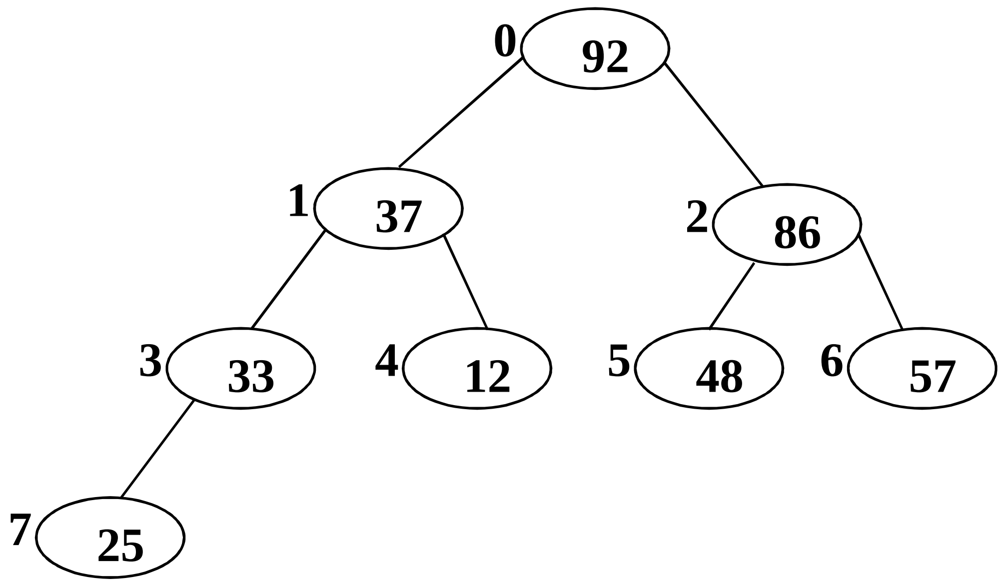


Heap Sort

92 37 86 25 12 48 57 33



92 37 86 33 12 48 57 25



Exercício:

Com base no que foi discutido implemente uma função, em C, que receba, como parâmetros, um vetor de inteiros e a quantidade de elementos no mesmo e retorne um vetor que represente um heap decrescente com os valores contidos inicialmente no vetor.

```
heap (int *x, int n)
{
    int i, elt, s, f;
    for (i=1; i<n; i++)
    {
        elt = x[i];
        s = i;
        f = (s-1)/2;
        while (s>0 && x[f]<elt)
        {
            x[s] = x[f];
            s = f;
            f = (s-1)/2;
        }
        x[s] = elt;
    }
}
```

Heap Sort

Agora que já definimos como receber um conjunto de chaves e transformá-lo em um heap devemos determinar como iremos utilizá-lo.

Se observarmos a característica básica da árvore que representa o heap, perceberemos que a raiz contém o elemento de maior valor do conjunto de chaves.

Sendo assim, podemos removê-lo e posicioná-lo no final do vetor que armazena o heap. Contudo, para isso temos que reorganizar o heap, mantendo sua propriedade e liberando espaço no final do vetor para colocar o elemento mencionado.

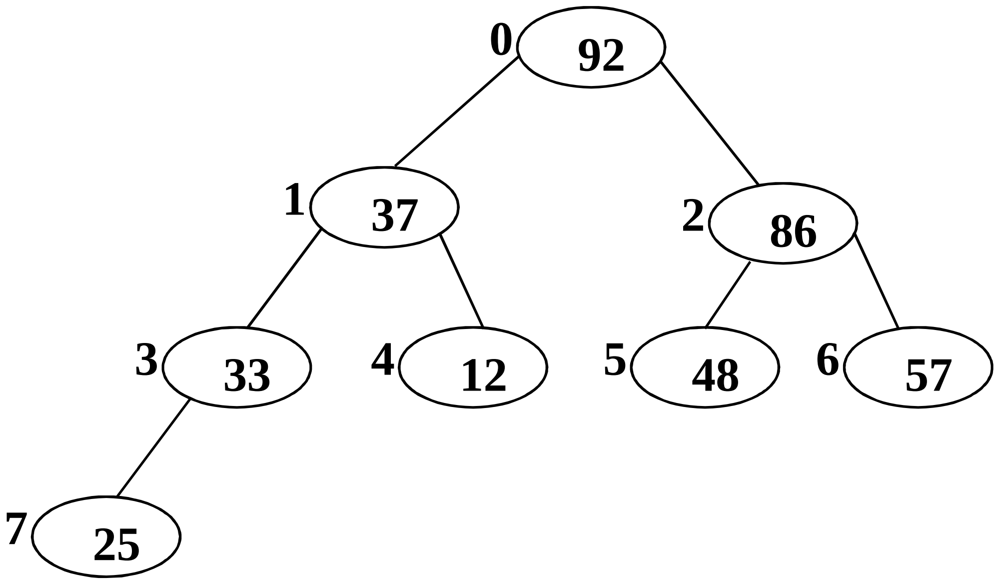
Heap Sort

Vamos analisar este processo retomando o exemplo anterior.

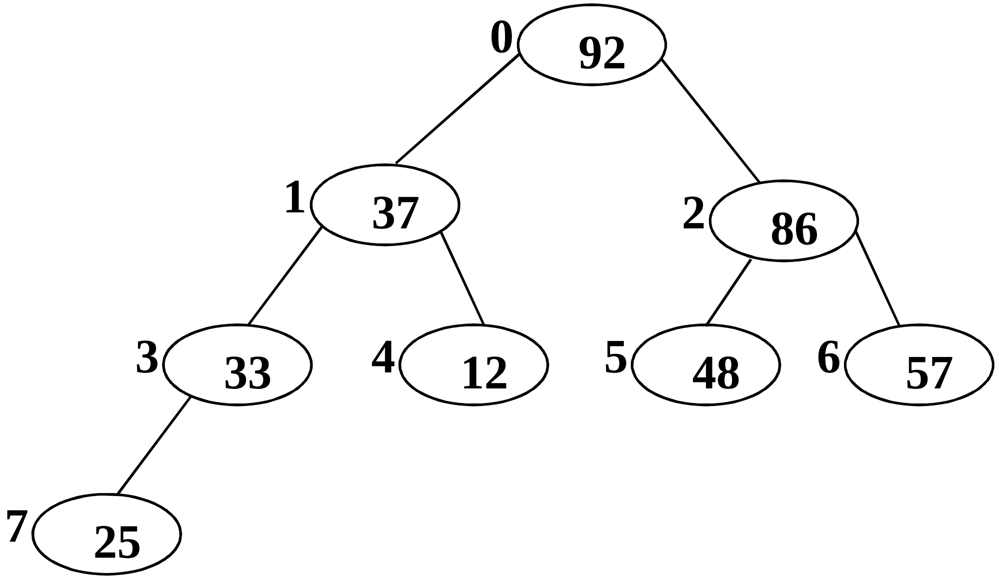
Considerando o vetor de chaves:

92 37 86 33 12 48 57 25

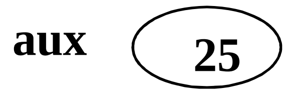
Que representa o heap:



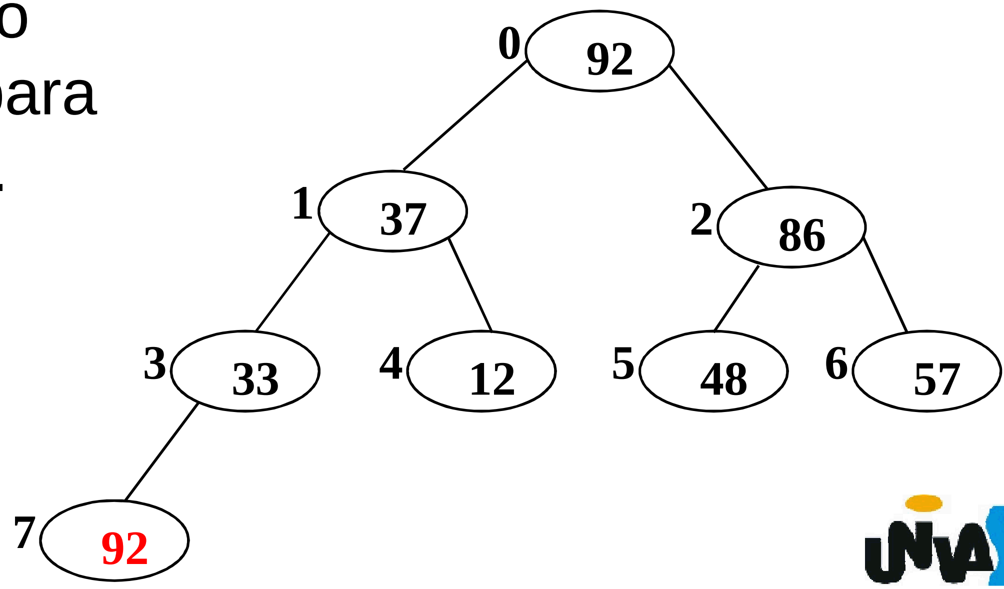
Heap Sort



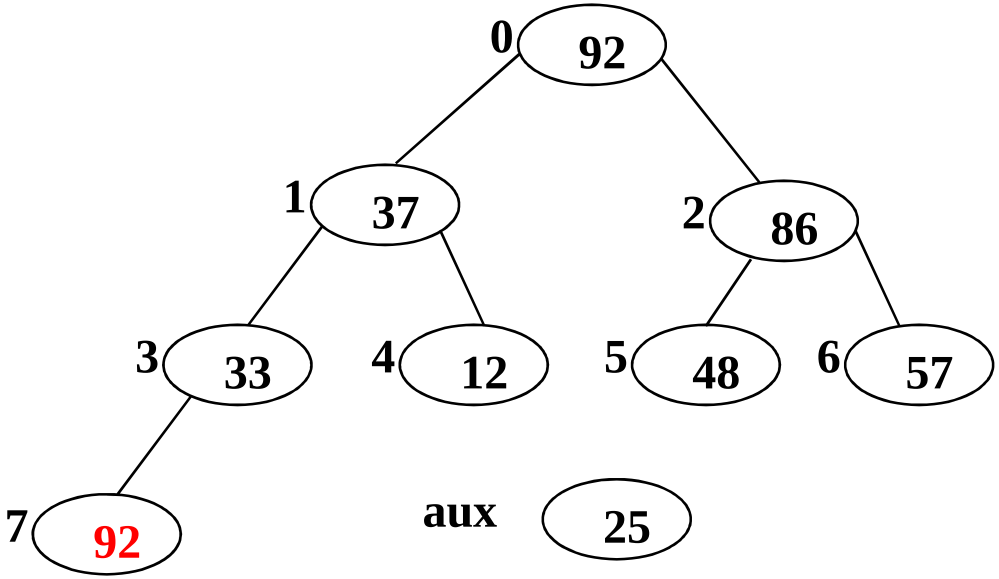
Armazenaremos o último elemento do nosso vetor em uma variável auxiliar.



Agora podemos copiar o elemento de maior valor para aposição final do vetor.



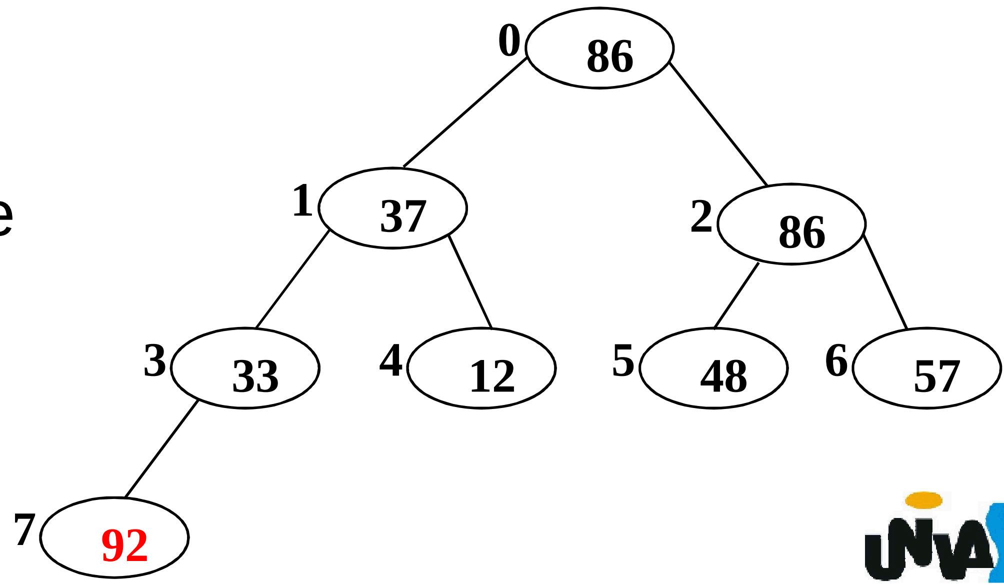
Heap Sort



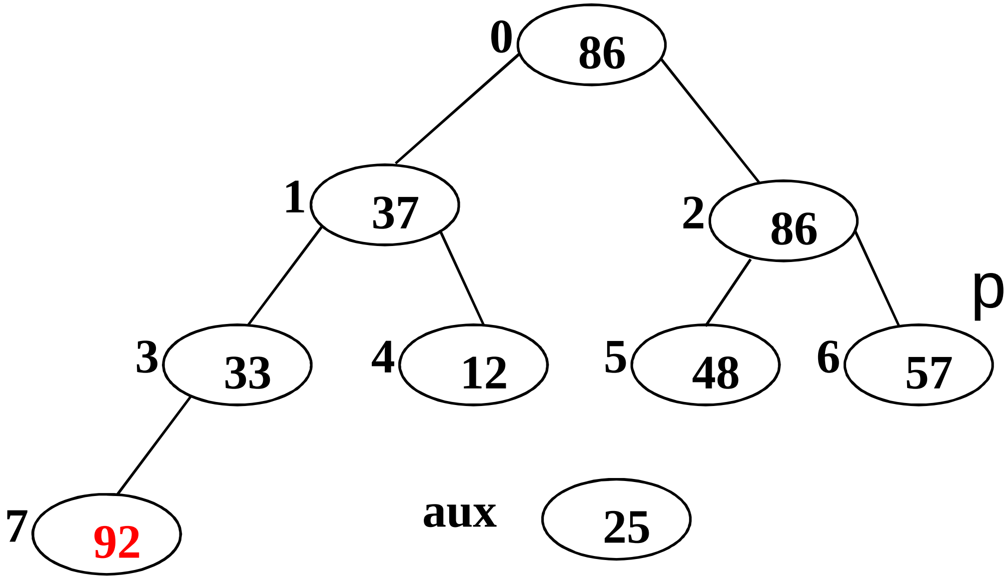
Podemos considerar a posição com índice zero como livre no vetor, reorganizar o heap e posicionar o valor armazenado em aux na posição correta.

Como fazer isto?

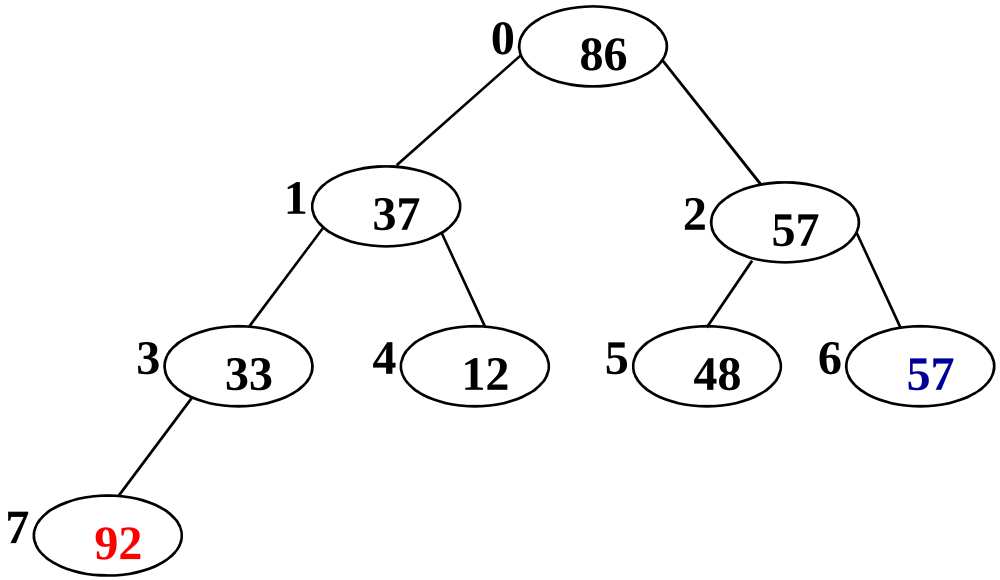
Escolhendo o maior dentre os filhos da raiz e deslocá-lo para tal posição.



Heap Sort



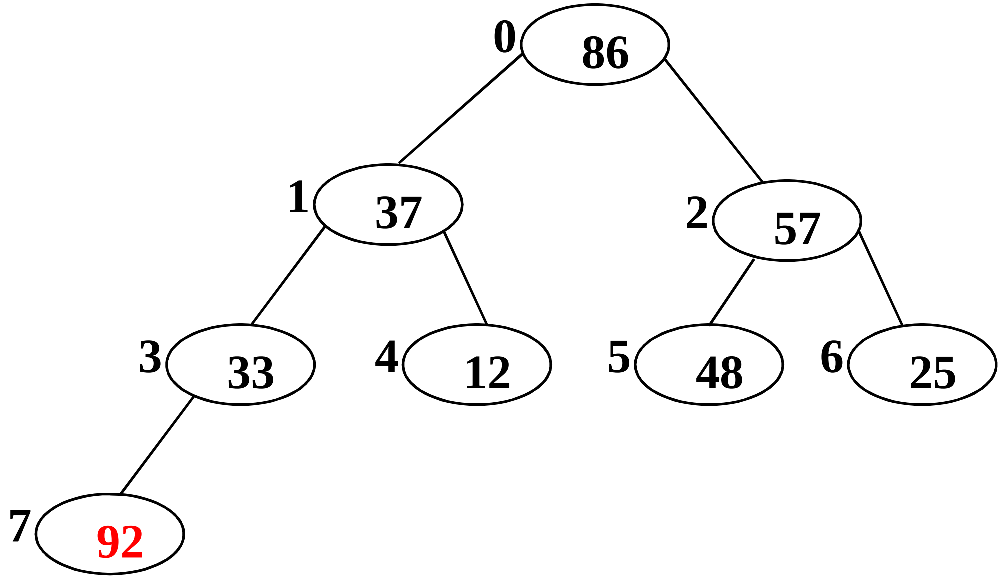
Mantendo este raciocínio, teremos a posição com índice 2 livre e a preencheremos com seu filho de maior valor.



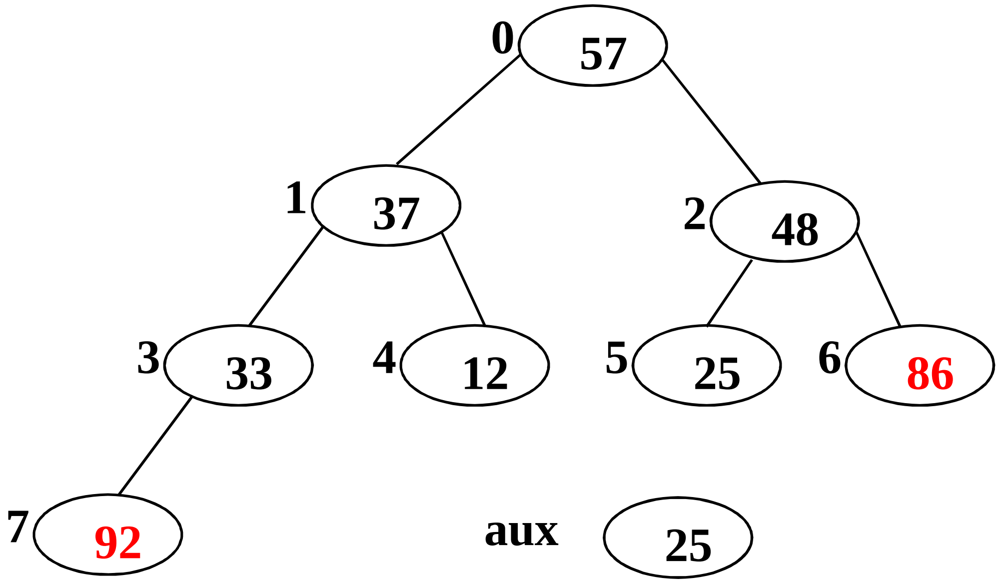
Quando chegarmos ao último filho que foi movido teremos a posição de inserção do valor em aux.

Após sua inserção teremos o heap do próximo slide.

Heap Sort

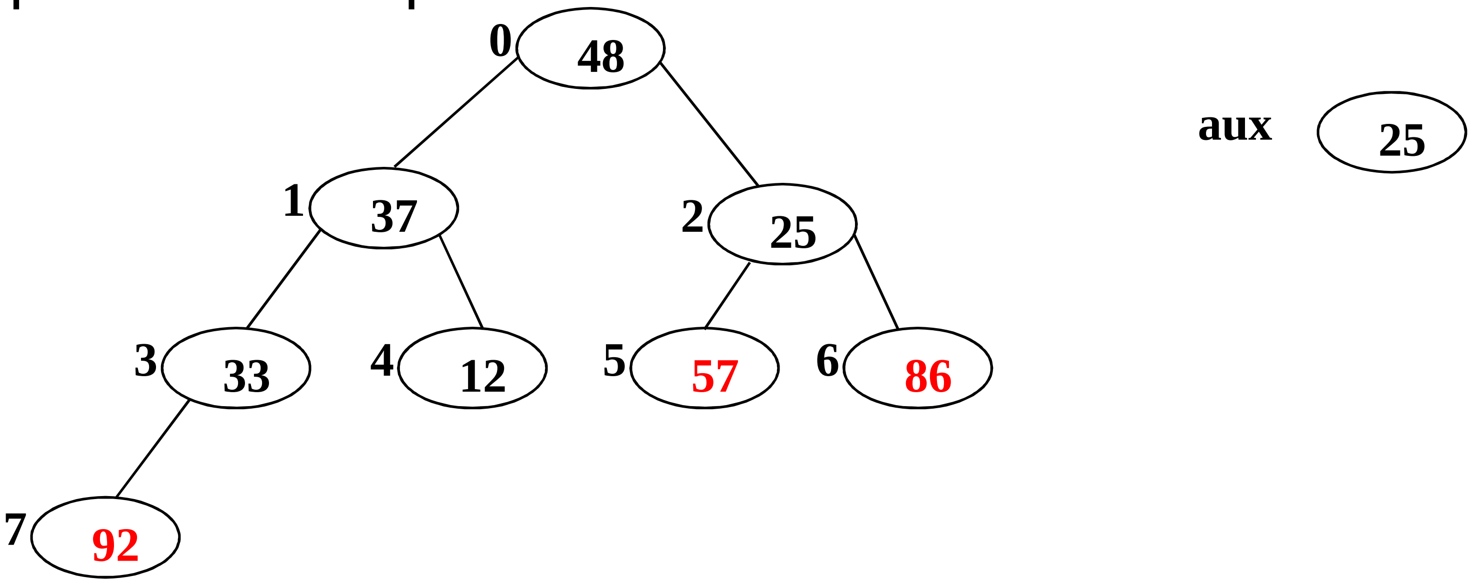


Aplicando este procedimento ao subvetor de n-1 teremos:

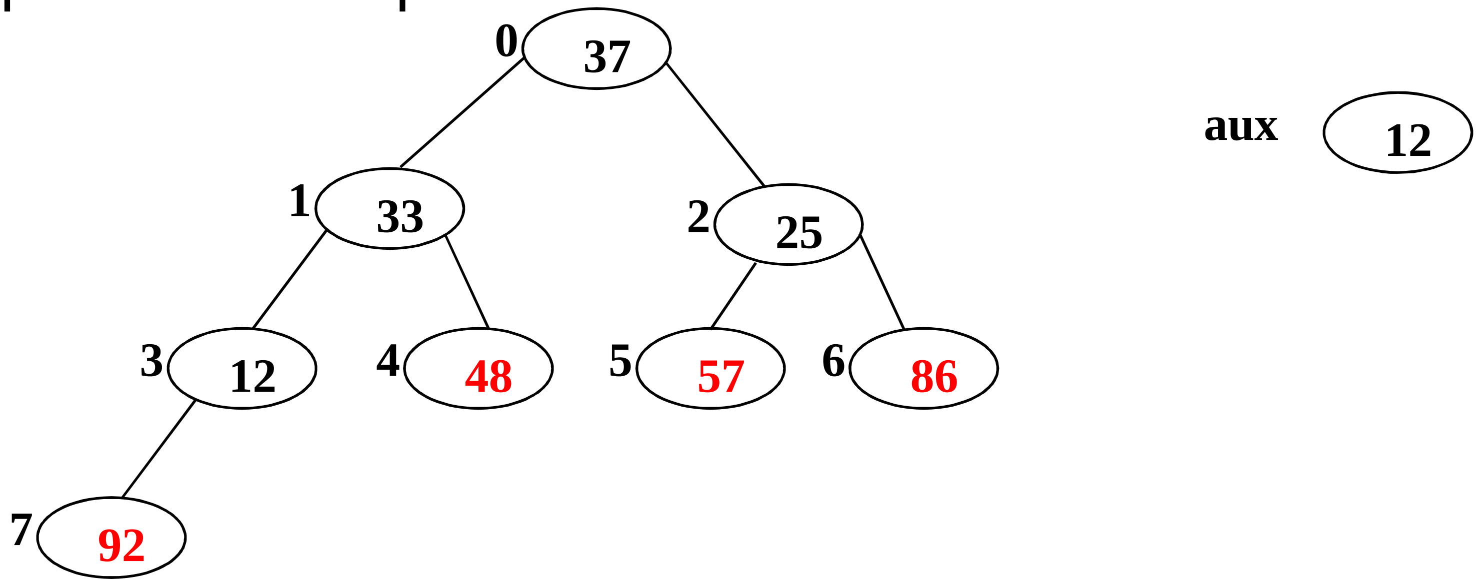


Heap Sort

Aplicando este procedimento ao subvetor de n-2 teremos:

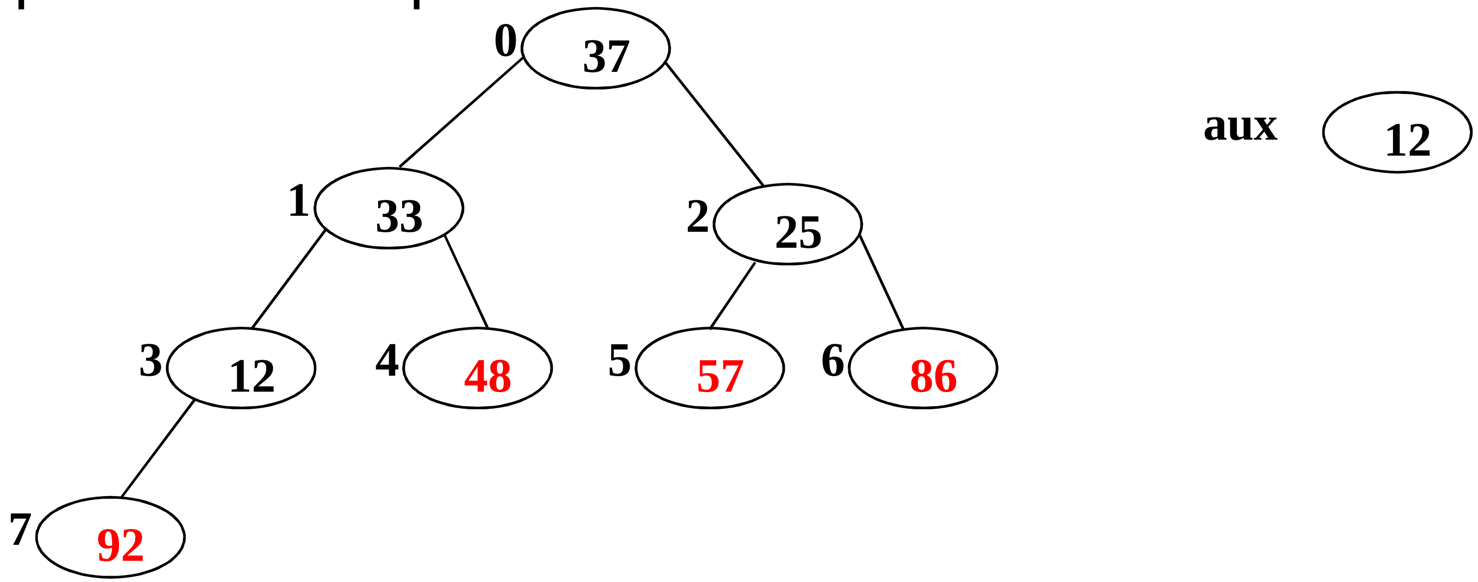


Aplicando este procedimento ao subvetor de n-3 teremos:

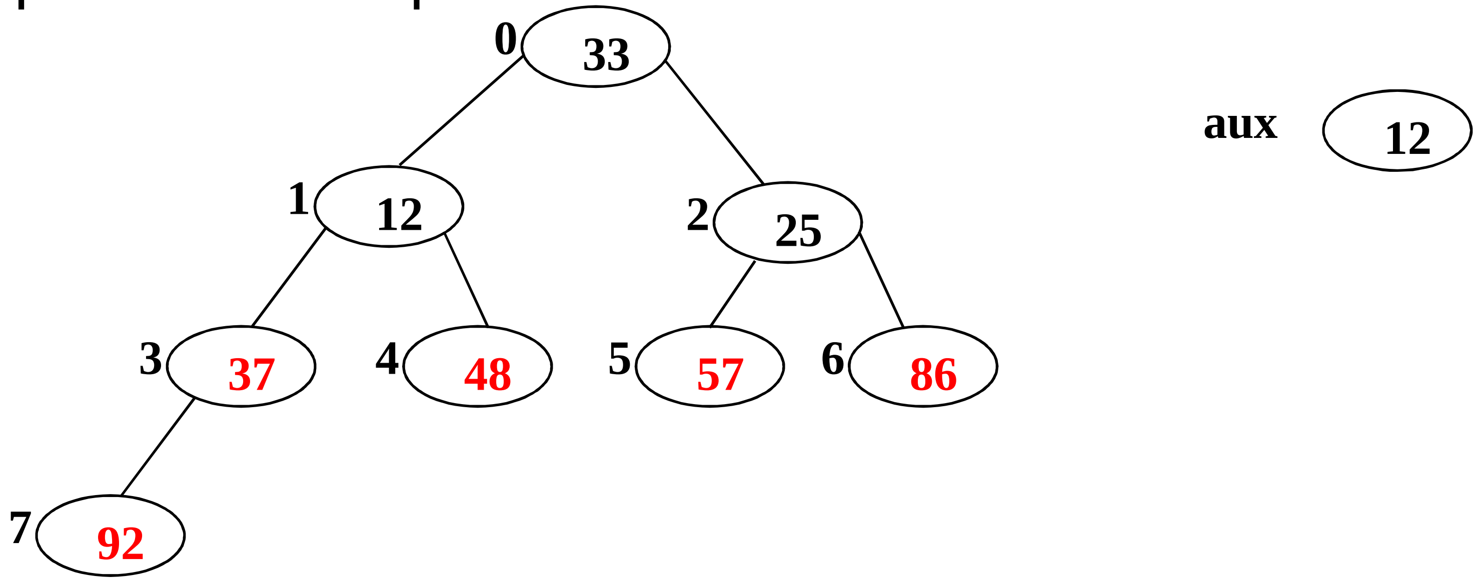


Heap Sort

Aplicando este procedimento ao subvetor de n-4 teremos:

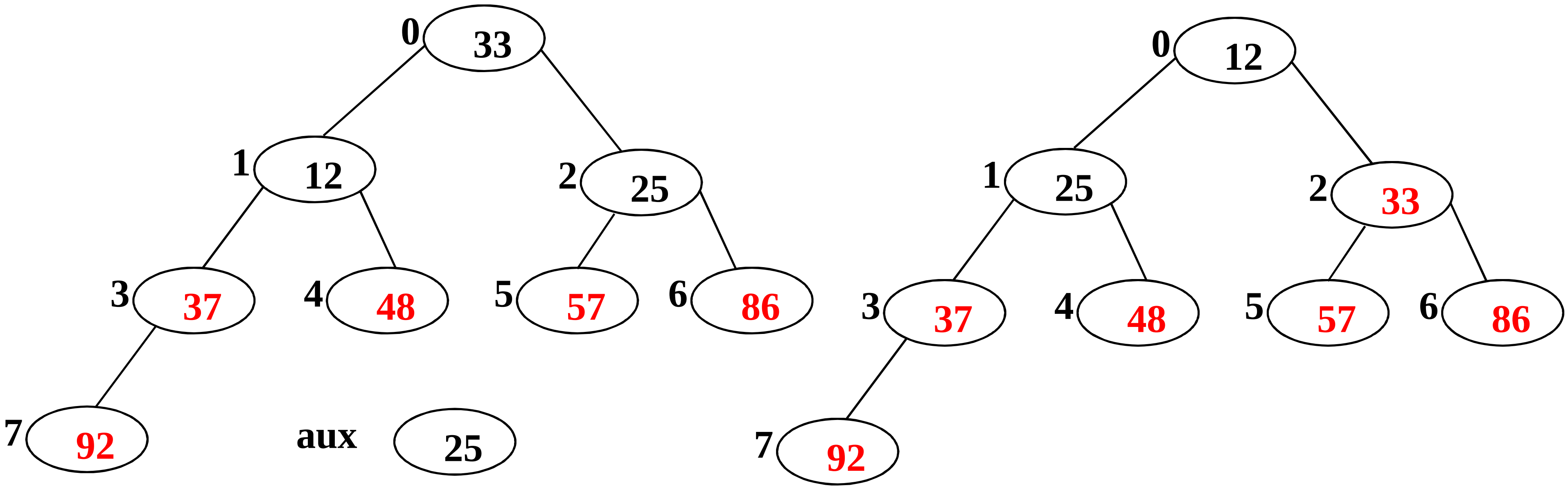


Aplicando este procedimento ao subvetor de n-5 teremos:



Heap Sort

Aplicando este procedimento ao subvetor de n-6 teremos:



Neste caso não movemos diretamente o filho, analisamos seu valor e apenas se este for menor que aux o movemos.

Com base no que foi discutido, codifique uma função, na linguagem C, que receba um vetor (de inteiros) e o número de elementos no mesmo e através do método *heap sort* ordene de forma crescente os elementos do vetor.

```
heapsort (int *x, int n)
{
  int i, elt, s, f, aux;
  /*fase de pré-processamento - cria heap inicial*/
  for (i=1; i<n; i++)
  {
    elt = x[i];
    s = i;
    f = (s-1)/2;
    while (s>0 && x[f]<elt)
    {
      x[s] = x[f];
      s = f;
      f = (s-1)/2;
    }
    x[s] = elt;
  }
```

```
/*fase de seleção = remove x[0] várias vezes, inserindo-o
em sua posição correta e acertando o heap*/
for (i=n-1; i>0; i--) {
  aux = x[i];
  x[i] = x[0];
  f = 0;
  if (i==1)
    s = -1;
  else
    s = 1;
  if (i>2 && x[2]>x[1])
    s = 2;
  while (s>=0 && aux<x[s]) {
    x[f] = x[s];
    f = s;
    s = 2*f+1;
    if (s+1 <= i-1 && x[s]<x[s+1])
      s = s+1;
    if (s > i-1)
      s = -1;
}
x[f] = aux;   }/*chave de fechamento do for*/   } /*chave de fechamento da função*/
```

Heap Sort

Para analisar o heap sort, observe que uma árvore binária completa com *n* nós tem log (*n*+1) níveis. Por conseguinte, se cada elemento no vetor fosse uma folha, exigindo que fosse filtrado pela árvore inteira durante a criação e o ajuste do heap, a classificação ainda seria **O**(*n* log *n*).

No caso médio, o heap sort não é tão eficiente quanto o quick sort. Experimentos indicam que o heap sort exige o dobro do tempo do quick sort para a entrada classificada aleatoriamente. Entretanto, o heap sort é bem superior ao quick sort no pior caso. Na realidade, o heap sort permanece **O** (*n* log *n*) no pior caso.

Heap Sort

Essa classificação não é muito eficiente para *n* pequeno devido à sobrecarga da criação do heap inicial e do cálculo da posição de pais e filhos.

A exigência de espaço para o heap sort (índices do vetor à parte) requer somente um registro adicional para armazenamento temporário durante a troca, desde que usada a implementação em vetor de uma árvore binária quase completa.

Comparação entre os Métodos

A ordenação interna é utilizada quando todos os registros do arquivo cabem na memória principal.

Quadros comparativos do tempo total real para ordenar arranjos com 500, 5.000, 10.000 e 30.000 registros na ordem aleatória, na ordem ascendente e na ordem descendente, respectivamente.

Em cada tabela, o método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.

Ordem Aleatória dos Registros

	500	5000	10000	30000
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

Comparação entre os Métodos

Ordem Ascendente dos registros

	500	5000	10000	30000
Inserção	1	1	1	1
Seleção	128	1524	3066	-
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

Ordem Descendente dos Registros

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

Classificação

É importante perceber que, quando o tamanho de uma lista *n* é pequeno, uma classificação **O**(n^2) é em geral mais eficiente do que uma classificação **O**(n log n). Isto acontece porque usualmente as classificações **O**(n^2) são muito simples de programar e exigem bem poucas ações além de comparações e trocas em cada passagem. Por causa dessa baixa sobrecarga, a constante de proporcionalidade é bem pequena. Em geral, uma classificação **O**(n log n) é muito complexa e emprega um grande

Classificação

número de operações adicionais em cada passagem para diminuir o número das passagens subseqüentes. Sendo assim, sua constante de proporcionalidade é maior. Quando *n* é grande, n^2 supera n log n, de modo que as constantes de proporcionalidade não desempenham um papel importante na determinação da classificação mais veloz. Entretanto, quando *n* é pequeno, n^2 não é muito maior que n log n de modo que uma grande diferença nessas constantes freqüentemente faz com que a classificação **O**(n^2) seja mais rápida.