UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Bauru

Faculdade de Ciências

Disciplina: Estruturas de Dados I – **ED1**
Professora: Simone das Graças Domingues Prado
e-mail: simone.prado@unesp.br

Apostila 05 - Heap

Objetivos:

⇒ Trabalhar com Filas de Prioridades
⇒ Trabalhar com Árvores Heap
⇒ Implementar rotinas básicas de Árvores Heap, tais como inserção, remoção e alteração de valores.
⇒ Implementar a rotina de ordenação usando Heap (Heapsort)

Conteúdo:

1. Fila de Prioridades

Como dito por Preiss (2000, p. 307), uma fila de prioridade é uma lista de itens na qual a cada item está associada a uma prioridade. Por exemplo, um gerenciador de impressão. O gerenciador poderá imprimir os arquivos na ordem em que chegam (usaria uma fila - FIFO), ou poderia ser implementadas prioridades nessa fila. O gerenciador poderia imprimir primeiro os arquivos menores e depois os maiores, criando assim uma fila de prioridades.

Como observado por Szwarcfiter & Markenzon (1994,p.175), em muitas aplicações é mais importante a prioridade dada a um elemento que o próprio elemento, ou seja, quando é criada uma lista de tarefas a serem realizadas são colocados graus de prioridades para que as tarefas sejam resolvidas da melhor forma.

De uma maneira geral usa-se valor numérico para a prioridade. Pode-se resolvê-las de forma ascendente (do menor valor da prioridade para o maior) ou de forma descendente (do maior valor para o menor).

Existem várias formas de manipular a fila de prioridades (fazer inserção, remoção, alteração de valores): lista não ordenada, lista ordenada, árvore binária e Heap. Veja suas vantagens e desvantagens:

- Lista não ordenada, seja sequêncial (representação estática) ou encadeada (representação dinâmica) - não exigirá esforço na inserção, mas exigirá na remoção do elemento, já que terá de buscar pelo maior elemento da lista. Para a alteração do valor do elemento deve-se fazer uma busca pelo elemento, o que exige tempo.
- Lista ordenada - facilita a remoção, já que se retira o primeiro elemento, mas dificulta a inserção e alteração de elementos.
- Árvore binária de busca - facilita a inserção, alteração e remoção, principalmente se forem balanceadas (AVL), mas são muito sofisticadas para implementar uma fila de prioridades (Preiss(2000,p.309)).
- Heap - implementação útil e bastante usada, onde as operações são otimizadas. Na inserção temos O(n), na remoção O(1), na alteração O(n) e na construção O(n log n) (Szwarcfiter & Markenzon (1994,p.177)).

2. Heap - Definições

Definição 01 - Drozdek, A.(2002, p.242)

Um tipo particular de árvore binária, chamada HEAP (na realidade, Heap Máxima), tem as seguintes propriedades:

(1) o valor de cada nó não é menor do que os valores armazenados em cada um dos seus filhos;
(2) a árvore é perfeitamente balanceada e as folhas no último nível estão todas nas posições mais à esquerda.

Se na primeira propriedade valer o contrário, ou seja, o valor de cada nó não é maior do que os valores armazenados em cada um dos filhos, tem-se uma Heap Mínima.

Definição 02: Tenenbaum et al.(1995, p.448)

"Defina um Heap descendente (conhecido também como Heap Máxima ou árvore descendente parcialmente ordenada) de tamanho n como uma árvore binária quase completa de n nós tal que o conteúdo de cada nó seja menor ou igual ao conteúdo de seu pai. Se a representação seqüencial de uma árvore binária quase completa for usada, essa condição se reduzirá à inequação:

info[j] <= info[(j-1)/2] para 0 <= ((j-1)/2)<j<= n-1"

"É possível também definir um Heap ascendente (ou uma Heap Mínima) como uma árvore binária quase completa de modo que o conteúdo de cada nó seja maior ou igual ao conteúdo de seu pai."

Definição 03: Preiss (2000, p.309)

"Uma (Min) Heap é uma árvore, T = { R, T_0, T_1, ..., T_{n-1}}, com as seguintes propriedades:

(1) toda subárvore de T é uma Heap,

(2) a raiz de T é menor ou igual a raiz de toda subárvore de T.

Isto é, R <= R_i para todo i, $0 \leq i < n$, onde R_i é a raiz de T_i."

Perceba que numa Árvore Heap Máxima, a raiz possui sempre o maior elemento do conjunto. Numa Árvore Heap Mínima tem-se sempre o menor elemento na raiz. Daí os nomes Heap Máxima e Heap Mínima.

Veja alguns exemplos. Na figura 01 são apresentadas três Árvores Heap Máxima e na figura 02, Árvores Heap Mínima. Na figura 03 são mostrados exemplos de Árvores Binárias que não são Heap.

Figura01. Árvores Heap Máxima

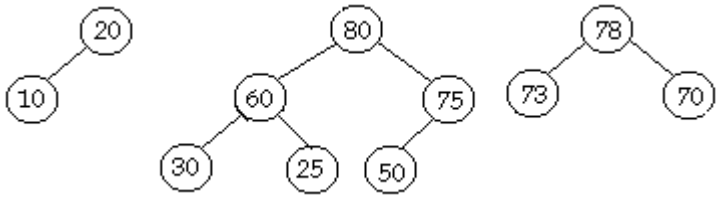


Figura02. Árvores Heap Mínima

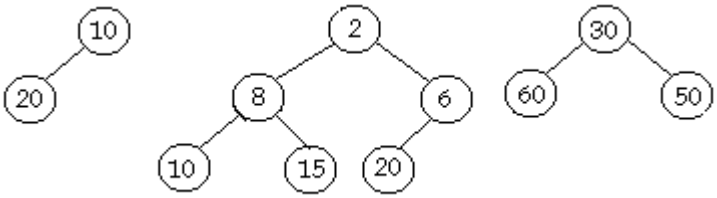
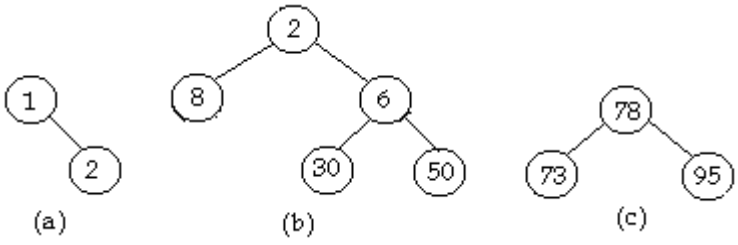


Figura03. Árvores Binárias não Heap



Departamento de Computação
Av. Engº Luiz Edmundo Carrijo Coube 14-01 Vargem Limpa Bauru-SP CEP: 17033-360
Fone: (14) 3103-6034 Fax: (14) 3103-6079 email: dcogeral@fc.unesp.br

Nas figura03(a) e figura03(b) o problema está em ter uma subárvore à direita e não à esquerda, já que na Árvore Heap as folhas no último nível devem estar em posições mais à esquerda. Na Figura3(c) o problema consiste em ter uma árvore binária onde o filho à esquerda é menor que a raiz e o filho à direita maior que a raiz. O certo é ter os filhos sempre menores (Heap Máximo) ou sempre maiores (Heap Mínimo) que a raiz.

Para a implementação de uma Árvore Heap usa-se uma representação estática, onde se constrói um vetor e coloca os elementos nas posições que deveriam aparecer na árvore. Aqui como temos uma árvore binária completa, podemos calcular que o valor máximo de nós é igual a $2^{(\text{altura}-1)}$ -1.

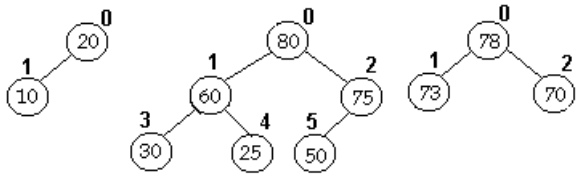
A construção do vetor pode ser feita como:

```
typedef int Arvore [Max];
```

onde o valor do nó fica armazenado na sua posição, enumerada a partir da raiz.

A representação das árvores da figura01 seria:

Figura 01. Árvores Heap Máxima



Portanto,

[0]	[1]	[2]	[3]	[4]	[5]
20	10				
80	60	75	30	25	50
78	73	70			

Pode-se observar que numa Árvore Heap Máxima o pai é sempre maior que seus filhos, então, por essa representação (info[j] <= info[(j-1)/2], para $0 < j \leq n-1$):

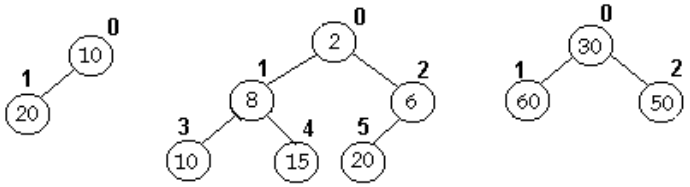
(1) n = 2; então info[j] <= info[(j-1)/2], para $0 < j \leq 1$
j = 1 → Info[1] =10 ≤ Info[0] = 20 (filho esquerda é menor que o pai)

(2) n = 6; então info[j] <= info[(j-1)/2] para $0 < j \leq 5$
j = 1 → Info[1] =60 ≤ Info[0] = 80 (filho esquerda (60) é menor que o pai (80))
j = 2 → Info[2] =75 ≤ Info[0] = 80 (filho direita (75) é menor que o pai (80))
j = 3 → Info[3] =30 ≤ Info[1] = 60 (filho esquerda (30) é menor que o pai (60))
j = 4 → Info[4] =25 ≤ Info[1] = 60 (filho direita (25) é menor que o pai (60))
j = 5 → Info[5] =50 ≤ Info[2] = 75 (filho esquerda (50) é menor que o pai (75))

(3) n = 3; então info[j] <= info[j/2] para $0 \leq j \leq 3$
j = 1 → Info[1] =73 ≤ Info[0] = 78 (filho esquerda (73) é menor que o pai (78))
j = 2 → Info[2] =70 ≤ Info[0] = 78 (filho direita (70) é menor que o pai (78))

Para as árvores da figura02 tem-se a representação:

Figura 02. Árvores Heap Mínima



Portanto,

[0]	[1]	[2]	[3]	[4]	[5]
10	20				
2	8	6	10	15	20
30	60	50			

onde Info[j] ≥ Info[(j-1)/2], para 0< j ≤ n, já que são Árvores Heap Mínimas

(1) n = 2; então info[j] ≥ info[(j-1)/2], para 0 < j ≤ 1
j = 1 → Info[1] =20 ≥ Info[0] = 10 (filho esquerda é maior que o pai)

(2) n = 6; então info[j] ≥ info[(j-1)/2], para 0 < j ≤ 5
j = 1 → Info[1] =8 ≥ Info[0] = 2 (filho esquerda (8) é maior que o pai (2))
j = 2 → Info[2] =6 ≥ Info[0] = 2 (filho direita (6) é maior que o pai (2))
j = 3 → Info[3] =10 ≥ Info[1] = 8 (filho esquerda (10) é maior que o pai (8))
j = 4 → Info[4] =15 ≥ Info[1] = 8 (filho direita (15) é maior que o pai (8))
j = 5 → Info[5] =20 ≥ Info[2] = 6 (filho esquerda (20) é maior que o pai (6))

(3) n = 3; então info[j] ≥ info[j/2], para 0 ≤ j ≤ 3
j = 1 → Info[1] =60 ≥ Info[0] = 30 (filho esquerda (60) é maior que o pai (30))
j = 2 → Info[2] =50 ≥ Info[0] = 30 (filho direita (50) é maior que o pai (30))

3. Inserindo itens em uma Árvore Heap Máxima

Numa inserção de elemento, na Árvore Heap, preocupa-se com duas coisas:

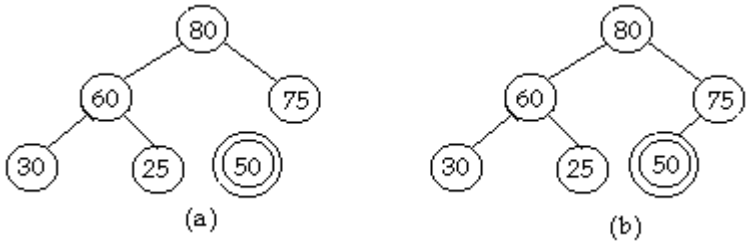
(1) formato correto da árvore e

(2) a ordenação como um Heap.

Assim, ao inserir, deve-se encontrar o lugar certo (próxima posição no nível da árvore - à esquerda e depois à direita) e verificar se não altera a ordenação da Árvore Heap, neste caso o Heap máximo. Se alterar, deve-se fazer deslocamentos nos elementos da árvore até chegar na ordenação correta.

Veja um exemplo. Suponha que se tenha a Árvore Heap Máxima da figura04(a) e o elemento 50 é inserido. A posição deve ser à esquerda de <75>, pois é nesse lugar que um novo nó poderá ser inserido. Ao colocar o <50>, percebe-se que ele não desordena a Árvore Heap Máxima. Então a inserção é feita sem problemas (figura04(b)).

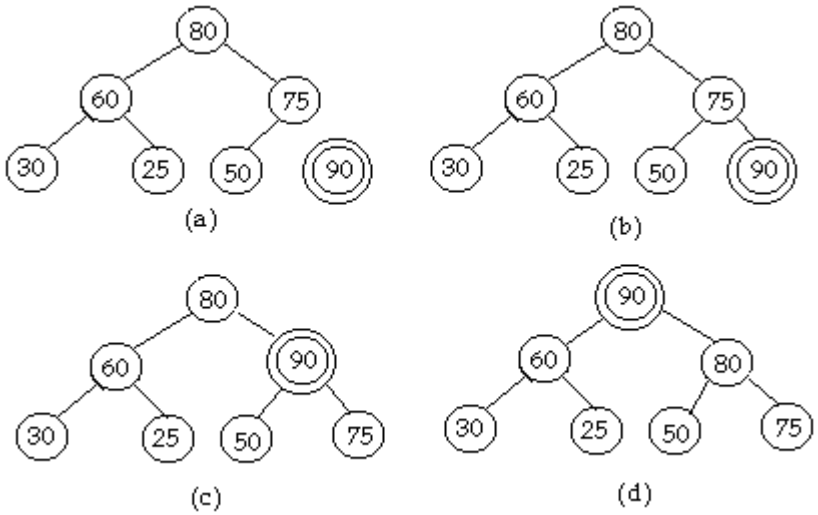
Figura04. Inserção do elemento 50 na Árvore Heap Máxima



Ou seja, se o Heap era: 80, 60, 75, 30, 25
O Heap se torna: 80, 60, 75, 30, 25, 50

Outro exemplo. Suponha que se tenha a Árvore Heap Máxima da figura05(a) e o elemento 90 deve ser inserido. A posição de inserção do novo nó é à direita de <75>. Ao colocar o <90>, percebe-se que ele não pode ocupar essa posição já que ele desordena a Árvore Heap Máxima (já que 90 é maior que o pai – 75). Então ocorre uma troca: ele pelo seu pai. Então o <75> desce e o <90> sobe (figura05.c). Só que ainda permanece desordenada, já que o pai de <90> é menor que ele. Então troca o nó <80> pelo <90> (figura05.d) e aí obtem-se a Árvore Heap Máxima.

Figura05. Inserção do elemento 90 na Árvore Heap Máxima



Ou seja, Heap: 80, 60, 75, 30, 25, 50
Na figura05.b => 80, 60, 75, 30, 25, 50, 90
Na figura05.c => 80, 60, 90, 30, 25, 50, 75
Na figura05.d => 90, 60, 80, 30, 25, 50, 75

Um algoritmo poderia ser:

- Coloque o novo item (N) no final do Heap (folha na posição apropriada)
- Enquanto (pai de N for menor que N) ou (N não estiver na raiz)
  troque N pelo seu pai.

Veja as rotinas em C:

```
void subir(int i, def_heap arvore){
      int troca, j = (i-1)/2;
      if( j>=0 && arvore[i]> arvore[j]){
            troca = arvore[i];
            arvore[i] = arvore[j];
            arvore[j] = troca;
            subir(j,arvore);}}

int Insere01(int nro, def_heap arvore){
   int n;
   if(cheia(arvore)) return 0;
   n = qtidade(arvore);
   arvore[n]=nro;
   subir(n,arvore);
   return 1;
}
```

No programa em C, o item é inserido e depois ele vai "subindo" até ficar na posição certa. Uma outra possibilidade (que é dada em Tenenbaum et al(1995, p.449)) é abrir espaço a partir de uma posição vazia da Árvore Heap Máxima até a raiz, procurando pelo primeiro elemento maior ou igual ao novo item. Quando essa posição for encontrada o elemento é inserido.

Veja a rotina em C:

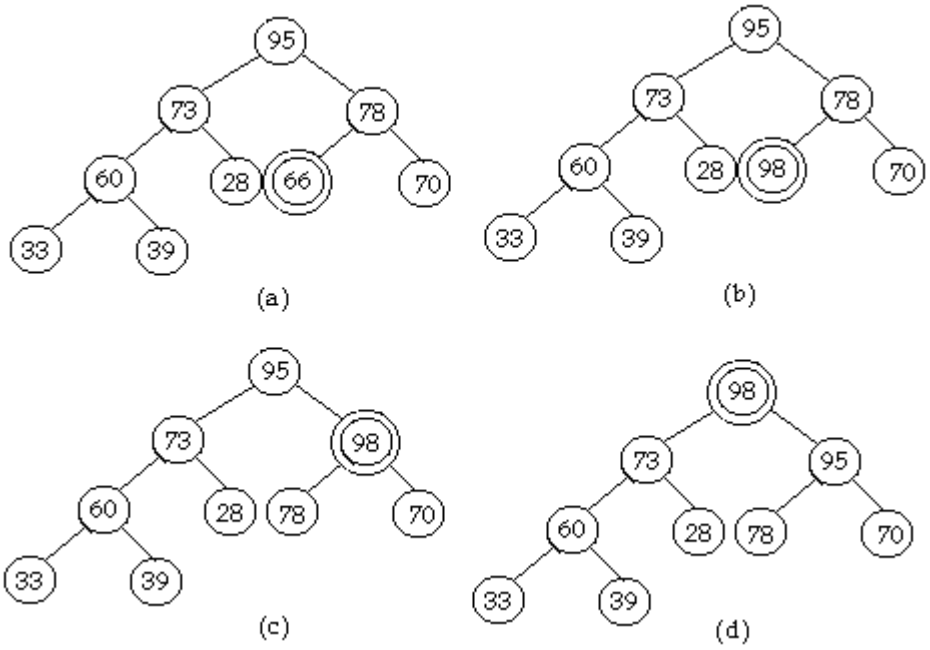
```
int Insere02(int valor, def_heap arvore){
    int s = qtidade(arvore);
   int f = (s - 1)/2;
   if (cheia(arvore)) return 0;
   while (s>0 && arvore[f]< valor){
               arvore[s]=arvore[f];
               s=f;
               f = (s-1)/2;}
   arvore[s]=valor;
   return 1;
}
```

4. Alterando valores de itens em uma Árvore Heap Máxima

Pode acontecer que se queira simplesmente mudar o valor de um dado item da Árvore Heap. Só que a alteração poderá exigir o rearranjo dos elementos na Árvore. Já que se deve mantê-la ordenada.

Como exemplo, veja a figura06 abaixo, onde se trocou o valor <66> por <98> (figura06.b). Como a Árvore tem de se manter ordenada, é preciso fazer deslocamentos. Primeiramente verifica-se que o <98> não poderia ficar abaixo de <78>, então troca-se o <98> pelo seu pai <78> (figura06.c). Aí se observa que o pai de <98> é menor que ele, aí troca-se o <98> pelo seu pai <95> (figura06.d). Pronto. A Árvore ficou ordenada.

Figura06. Alteração numa Árvore Heap Máxima



Departamento de Computação
Av. Engº Luiz Edmundo Carrijo Coube 14-01 Vargem Limpa Bauru-SP CEP: 17033-360
Fone: (14) 3103-6034 Fax: (14) 3103-6079 email: dcogeral@fc.unesp.br

Ou seja, Heap: 95, 73, 78, 60, 28, [66], 70, 33, 39
Na figura06.b => 95, 73, 78, 60, 28, [98], 70, 33, 39
Na figura06.c => 95, 73, [98], 60, 28, 78, 70, 33, 39
Na figura06.d => [98], 73, 95, 60, 28, 78, 70, 33, 39

Um algoritmo poderia ser:

- Troque o valor do item (Y por X)
- verifique se X é maior ou menor que seus filhos
- Se X > filhos e menor que pai, sobe com o elemento
- se X < filhos, desce o elemento.

Em C:

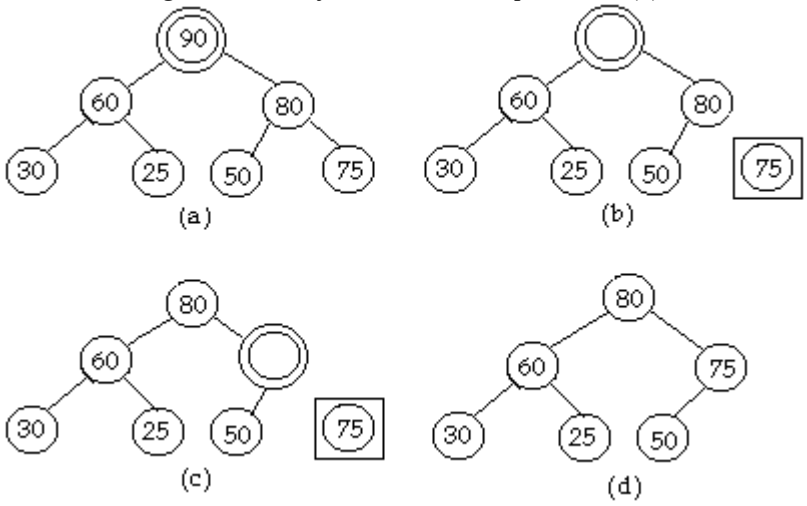
```
int altera_prioridade(int nro1, int nro2, def_heap arvore){
   int x,i,n = qtidade(arvore)+1;

   if(vazia(arvore)) return 0;
   for(i=0;(i<n && arvore[i]!=nro1);i++);
   if(arvore[i]!=nro1) return 0;
   arvore[i]=nro2;
   x = maior_filho(i,n,arvore);
   if (arvore[i] < arvore[x]) descer(i,arvore);
   else subir(i,arvore);
   return 1;
}
```

5. Removendo itens de uma Árvore Heap Máxima

A remoção na Árvore Heap sempre acontece na raiz. A questão fica em como reorganizar a árvore depois dessa remoção. A idéia é verificar qual posição deveria ser removida para que a árvore permaneça no formato correto. A partir daí fazer deslocamentos para que aquela posição possa ser removida e o elemento, que estava ali, ser realocado.

Veja um exemplo. Suponha a Árvore Heap Máxima da figura07(a). O ítem a ser removido é o elemento 90 (raiz da árvore). A posição que deve ser removida é onde está o <75>, já que ele está mais a direita e no último nível. Então, tem de mover o nó vazio (lacuna) para baixo até uma posição onde o elemento 75 possa ser reinserido na árvore. Para fazer esse movimento da lacuna e tendo uma Árvore Heap Máxima, a lacuna é movida para o maior elemento, neste caso para a direita. Assim o elemento <80> ocupa a raiz. Como 75 > 50, a lacuna é ocupada pelo <75>.

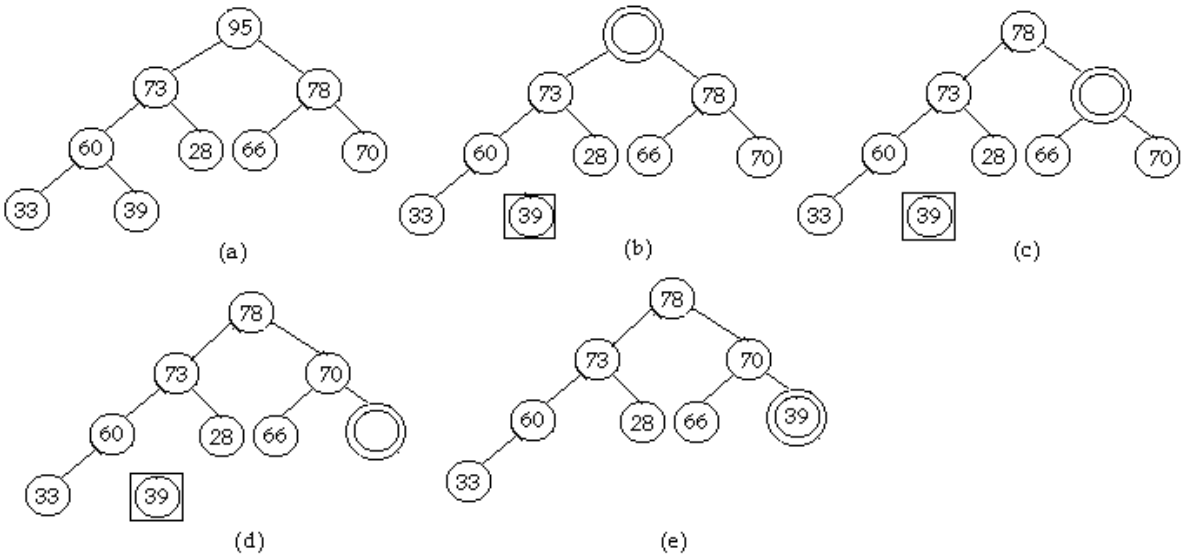
Figura07. Remoção na Árvore Heap Máxima (1)



Ou seja, Heap: 90, 60, 80, 30, 25, 50, 75
Na figura07.b => <>, 60, 80, 30, 25, 50, (75)
Na figura07.c => 80, 60, <>, 30, 25, 50, (75)
Na figura07.d => 80, 60, 75, 30, 25, 50

Veja um exemplo mais complicado na figura08, onde é removido o elemento <95> e é a posição do elemento <39> que deve ser removida para que a árvore continue sendo Heap Máxima.

Figura08. Remoção na Árvore Heap Máxima (2)



Ou seja, Heap: 95, 73, 78, 60, 28, 66, 70, 33, 39
Na figura08.b => <>, 73, 78, 60, 28, 66, 70, 33, (39)
Na figura08.c => 78, 73, <>, 60, 28, 66, 70, 33, (39)
Na figura08.d => 78, 73, 70, 60, 28, 66, <>, 33, (39)
Na figura08.e => 78, 73, 70, 60, 28, 66, 39, 33, <>

Um algoritmo poderia ser:

- Extraia o elemento da raiz
- Faça: mova o maior filho da lacuna para ocupar essa posição vazia
  Enquanto existe filho para ser movido.
- Se necessário extraia e insira o último elemento do Heap.

A rotina em C fica sendo:

```
int maior_filho(int f, int k, def_heap arvore){
   int s = 2*f+1;
   if((s + 1) <= k && arvore[s]<arvore[s+1]) s = s+1;
   if (s > k) return (-1);
   else return s;
}

void ajusta_heap(int R, int K, def_heap arvore){
   int f = R;                              // R = posição vaga
   int s = maior_filho(f,K-1,arvore);      // K = qt elementos da arvore
   if(s >= 0 && arvore[K] < arvore[s]){
      arvore[f] = arvore[s];
      ajusta_heap(s,K,arvore);}
   else arvore[f] = arvore[K];
}

int Remove_heap01(int *valor, def_heap arvore){
   int k = qtidade(arvore);
   if(vazia(arvore))return 0;
   *valor = arvore[0];
   ajusta_heap(0,k-1,arvore);
   arvore[k-1]=0;
   return 1;
}
```

Uma outra possibilidade é remover a raiz, alocar na raiz o ultimo elemento do Heap e descer com esse elemento, se necessário (Szwarcfiter & Markenzon(1994,p183)

```
void descer(int posicao, def_heap arvore){
   int pos, troca, n = qtidade(arvore);;
   pos = posicao*2+1; /* filho esquerdo */
   if(pos < n)
         if(arvore[pos+1] > arvore[pos]) pos++; /*filho direito*/
         if(arvore[posicao] < arvore[pos]){
               troca = arvore[pos];
               arvore[pos] = arvore[posicao];
               arvore[posicao]= troca;
               descer(pos,arvore);}
}

int Remove_heap02(int *valor, def_heap arvore){
   int n = qtidade(arvore);
   if(vazia(arvore))return 0;
   *valor = arvore[0];
   arvore[0] = arvore[n-1];
   descer(0,arvore);
   arvore[n-1]=0;
   return 1;
}
```

6. Heapsort

Heapsort é um método eficiente de ordenação. Como dito em Tenenbaum (1995,p.4570), no caso médio, o Heapsort não é tão eficiente quanto o quicksort, entretanto é bem superior ao quicksort no pior caso.

A idéia é remover sucessivamente a raiz da Árvore Heap Máxima, colocá-la no final do vetor e regenerar a Heap, que agora tem um elemento a menos.

Em C ficaria:

```
void heapsort01(def_heap arvore, int R[])
{  int i, n=qtidade(arvore);
   for(i=n-1; i>0; i--) Remove_heap01(&(R[i]),arvore);
   R[0] = arvore[0];
}
```

Ou pode-se usar :

```
void heapsort02(def_heap arvore, int R[])
{  int i, n=qtidade(arvore);
   for(i=n-1; i>0; i--) Remove_heap02(&(R[i]),arvore);
   R[0] = arvore[0];
}
```

7. Exercícios

Para os exercícios de 01 a 04, não use o computador para resolver.

1. Verificar se as sequências abaixo correspondem ou não a um Heap
a) 33, 32, 28, 31, 26, 29, 25, 30, 27
b) 33, 32, 28, 31, 29, 26, 25, 30, 27

2. Sejam os Heap especificados a seguir:
a) [95, 85, 92, 47, 70, 91, 34, 20, 40, 46], altere o valor 70 para 93
b) [95, 85, 92, 47, 70, 91, 34, 20, 40, 46], altere o valor 70 para 19
c) [17, 15, 17, 10, 8, 17, 5] altere o valor 5 para 20

3. Para cada uma das sequências de chaves a seguir, determine a Árvore Heap obtida quando as chaves são inseridas uma a uma na ordem dada em uma Heap Máxima inicialmente vazia:
a) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
b) 3, 1, 4, 1, 5, 9, 2, 6, 5, 4
c) 2, 7, 1, 8, 2, 8, 1, 8, 2, 8
d) 18, 25, 41, 34, 14, 10, 52, 50, 48
e) 5, 55, 12, 25, 98, 91, 24, 56, 17, 23
f) 10, 5, 7, 25, 20, 3, 1, 2, 6, 4
g) 5, 9, 2, 5, 8, 12, 43, 78, 65, 34, 1, 0, 56, 32, 99
h) 46, 40, 20, 34, 91, 70, 47, 92, 85, 95
i) 50, 70, 90, 40, 35, 80, 85, 97, 95, 99
j) 12, 45, 39, 56, 50, 34, 70, 45, 87, 100
k) 32, 8, 1, 12, 32, 26, 2, 9, 10, 5
l) 60, 13, 35, 29, 9, 27, 83, 68, 64, 75

4. Sejam os Heap especificados a seguir
a) [6, 16, 18, 20, 21, 56, 67, 67, 92, 98], remova três elementos.
b) [17, 12, 8, 5, 3, 6, 2, 4, 2,1], remova três elementos.
c) [55, 45, 28, 31, 33, 26, 25, 30, 27, 26, 32], remova cinco elementos.
d) [33, 32, 28, 31, 29, 26, 25, 30, 27], remova cinco elementos

5. Faça uma rotina para verificar se a árvore binária é uma Heap.
6. Faça uma rotina para verificar se uma Árvore Heap está no formato correto.
7. Faça uma rotina para verificar se a Árvore Heap é uma Heap Máxima.
8. Faça uma rotina para verificar se a Árvore Heap é uma Heap mínima.
9. Faça uma rotina para encontrar numa Heap mínima o maior elemento
10. Faça uma rotina para encontrar numa Heap Máxima o menor elemento
11. Faça uma rotina de inserção numa Árvore Heap mínima
12. Faça uma rotina de remoção numa Árvore Heap mínima
13. Faça uma rotina de Heapsort numa Árvore Heap mínima

8. Bibliografia

DROZDEK, Adam. **Estruturas de Dados e Algoritmos em C++**. Tradução de Luiz Sérgio de Castro Paiva; Revisão de Flávio Soares Corrêa da Silva. São Paulo: Pioneira Thomson Learning, 2002. 579p.

PEREIRA, Silvio do Lago. **Estruturas de Dados Fundamentais: Conceitos e Aplicações**. São Paulo: Érica, 1996. 246p.

PREISS, Bruno R. **Estruturas de Dados e Algoritmos: Padrões de projetos orientados a objeto com Java.** Tradução: Elizabeth Ferreira. Rio de Janeiro: Campus, 2000. 566p.

SCHILDT, Herbert. **C, Completo e Total**. 3a.ed. Tradução e Revisão Técnica: Roberto Carlos Mayer. São Paulo: Pearson Education do Brasil, 1997. 827p.

SZWARCFITER, James Luiz & MARKENZON, Lilian. **Estruturas de Dados e seus Algoritmos**. Rio de Janeiro: LTC-Livros Técnicos e Científicos Ed., 1994. 320p.

TENENBAUM, Aaron M; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J**. Estruturas de Dados usando C**. Tradução: Teresa Cristina Félix de Souza; Revisão Técnica e Adaptação de Programas: Roberto Carlos Mayer. São Paulo: Makron Books, 1995. 884p.