CI1057: Algoritmos e Estruturas de Dados III
Lista de Prioridades e Heap

Profa. Carmem Hara

Departamento de Informática/UFPR

10 de julho de 2024

Lista de Prioridades

Uma lista de prioridades é uma estrutura de dados composta por itens com chaves que dão suporte a duas operações básicas: inserção de um novo item e remoçaõ do item que tem a maior (ou menor) chave.

Aplicações

- eventos processados em ordem cronológica
- escalonamento de processos em sistemas computacionais
- computações numéricas (maiores erros são tratados primeiro)
- implementação do algoritmo de compressão

TAD Lista de Prioridades

- `void inicializa( Lista *l )`
  inicializa a lista de prioridades `L` vazia
- `int listaVazia( Lista l )`
  retorna 1 de `L` estiver vazia e 0, caso contrário
- `void insere( TipoItem v, Lista *l )`
  insere o item `v` na lista `L`
- `TipoItem removeMax( Lista *l )` ou
  `TipoItem removeMin( Lista *l )`
  retorna o valor máximo (ou mínimo) na lista `L`
- `TipoItem obtemMax ( Lista l )` ou
  `TipoItem obtemMin( Lista l )`
  retorna o valor máximo (ou mínimo) sem removê-lo

TAD List de Prioridades - Implementação

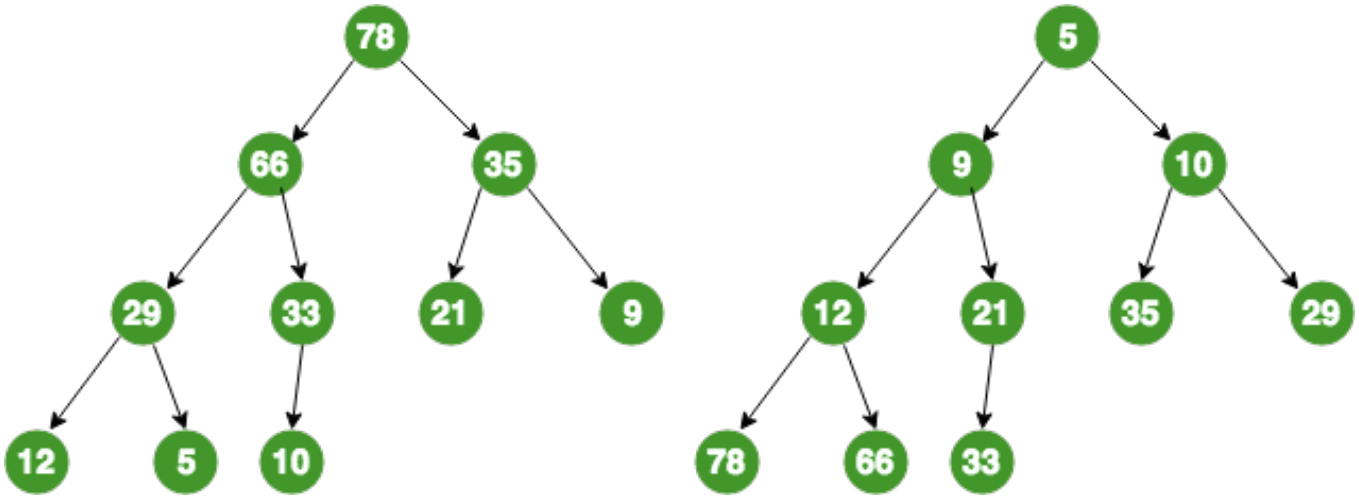
Possíveis estruturas de dados

- Arrays não ordenados
  inserção: $O(1)$, remoção do máximo: $O(n)$
- Arrays ordenados
  inserção: $O(n)$, remoção do máximo: $O(1)$
- Lista ligada não ordenada
  inserção: $O(1)$, remoção do máximo: $O(n)$
- Lista ligada ordenada
  inserção: $O(n)$, remoção do máximo: $O(1)$

Árvore em Ordem Heap

Uma árvore está em ordem (max-)*heap* se a chave em cada nodo é maior ou igual às chaves dos seus filhos.

Exemplo de árvores em ordem max-heap e min-heap:



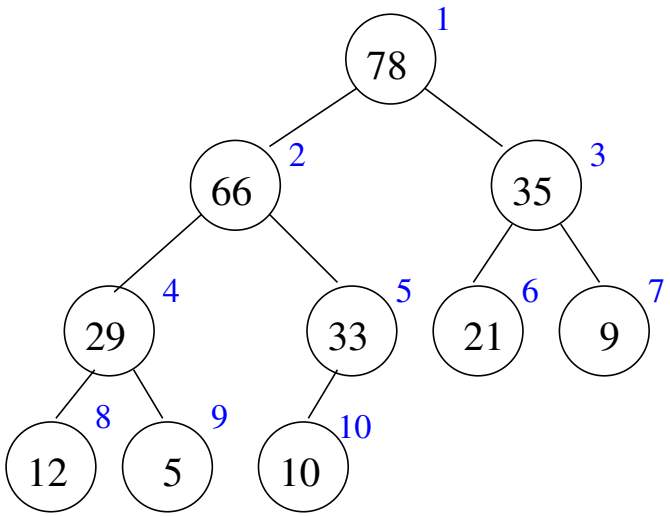
Heap

Um **heap** é uma árvore binária *quase completa* em ordem heap.
A estrutura pode ser eficientemente representada por um array.

- A altura de um heap com n nodos internos $\lceil lg(n+1) \rceil$
- Implementação de uma lista de prioridades com heap
  Inserção: $O(lg\ n)$, remoção do máximo: $O(lg\ n)$

Heap: Representação com Array

- Representação compacta, não precisa armazenar apontadores
- Necessidade de saber a quantidade máxima de chaves do heap
- Posição no array: percurso por nível da árvore
  - raiz está na posição 1
  - $pai(i) = \lceil i/2 \rceil$
  - $esq(i) = 2 * i$
  - $dir(i) = 2 * i + 1$



78	66	35	29	33	21	9	12	5	10
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Algoritmos em Heaps

Todos os algoritmos seguem o padrão:

1. alteração da raiz ou de uma folha, que pode violar a ordem-heap
2. arrumar a árvore de forma top-down (`arrumaHeapDown`) ou de forma bottom-up (`arrumaHeapUp`) para manter a ordem-heap

- `arrumaHeapDown` e `arrumaHeapUp` requerem apenas percorrer um caminho na árvore da raiz até uma folha ou de uma folha até a raiz.
- Ambos são $O(lg\ n)$

Estrutura de Dados

```
1 #define MAX 100
2
3 typedef struct heap {
4   int tamHeap;
5   TipoItem elem[MAX+1];
6 } Heap;
```

ArrumaHeapUp

Sobe na árvore, trocando os valores do nodo k com o seu pai quando necessário até atingir a raiz ou quando não houver mais troca.

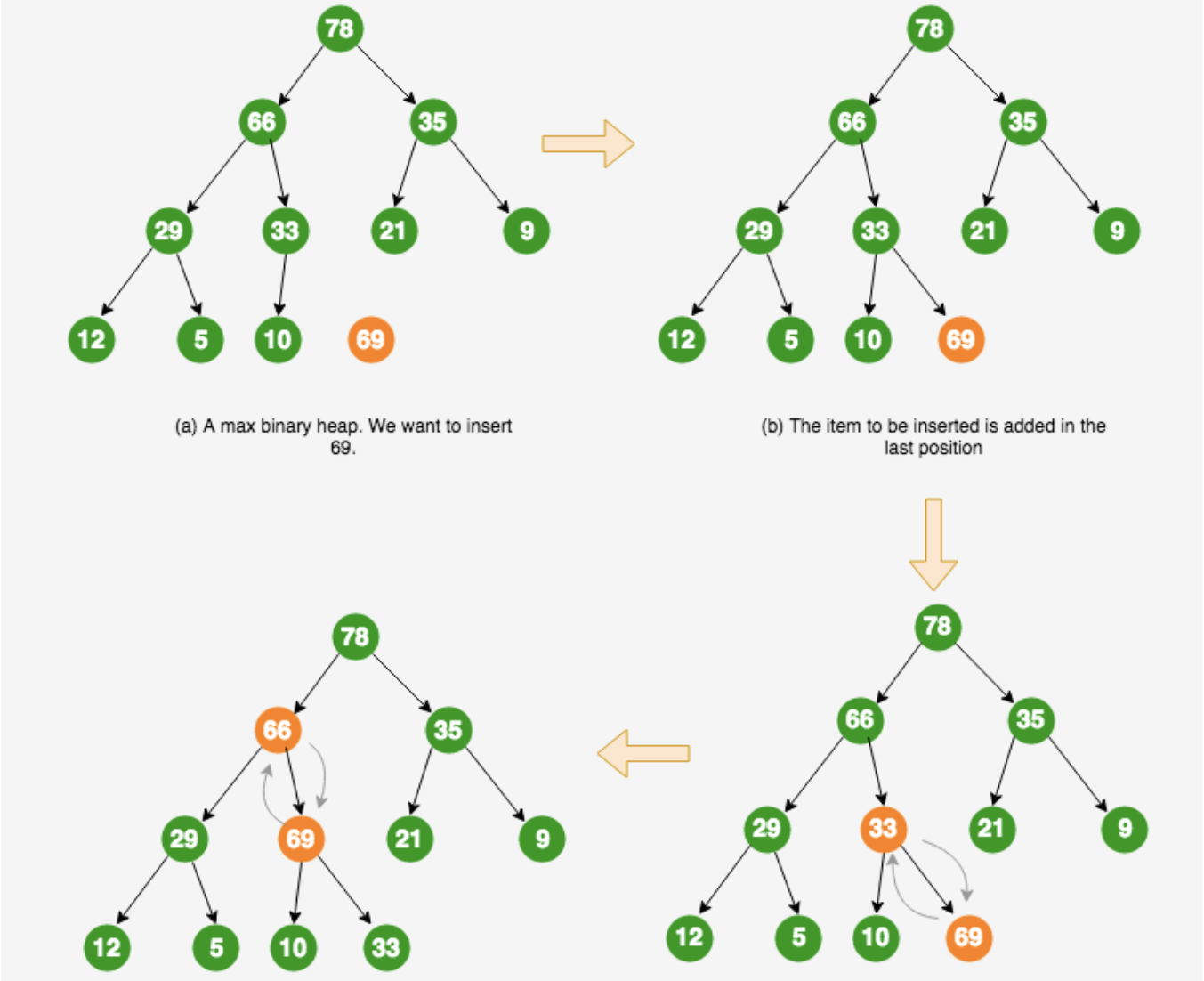
```
1 void arrumaHeapUp( Heap *h, int k ){
2   while( k > 1 && lt( h->elem[pai(k)], h->elem[k] )){
3     troca( &h->elem[pai(k)], &h->elem[k] );
4     k= pai(k);
5   }
6 }
```

ArrumaHeapDown

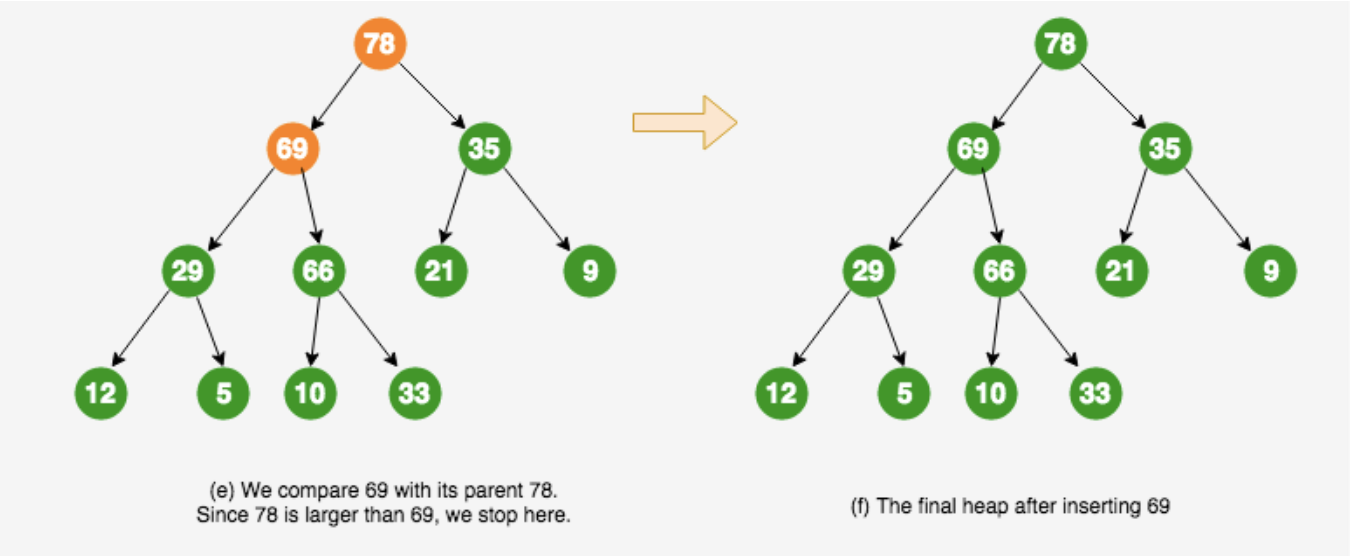
- Considera que as subárvores do nodo com número k já satisfazem a ordem (max)-heap.
- Troca os valores do nodo com o maior dos seus filhos
- Finaliza quando chegar a uma folha ou quando não houver mais troca

```
1  void arrumaHeapDown( Heap *h, int k ){
2    int e, d, maior;
3
4    e= esq(k);
5    d= dir(k);
6    if( e <= h->tamHeap && gt(h->elem[e], h->elem[k]) )
7      maior= e;
8    else maior= k;
9    if( d <= h->tamHeap && gt(h->elem[d], h->elem[maior] )
10     maior= d;
11   if( maior != k ){
12     troca( &h->elem[k], &h->elem[maior] );
13     arrumaHeapDown( h, maior );
14   }
15 }
```

Heap: Inserção



Heap: Inserção (Cont.)



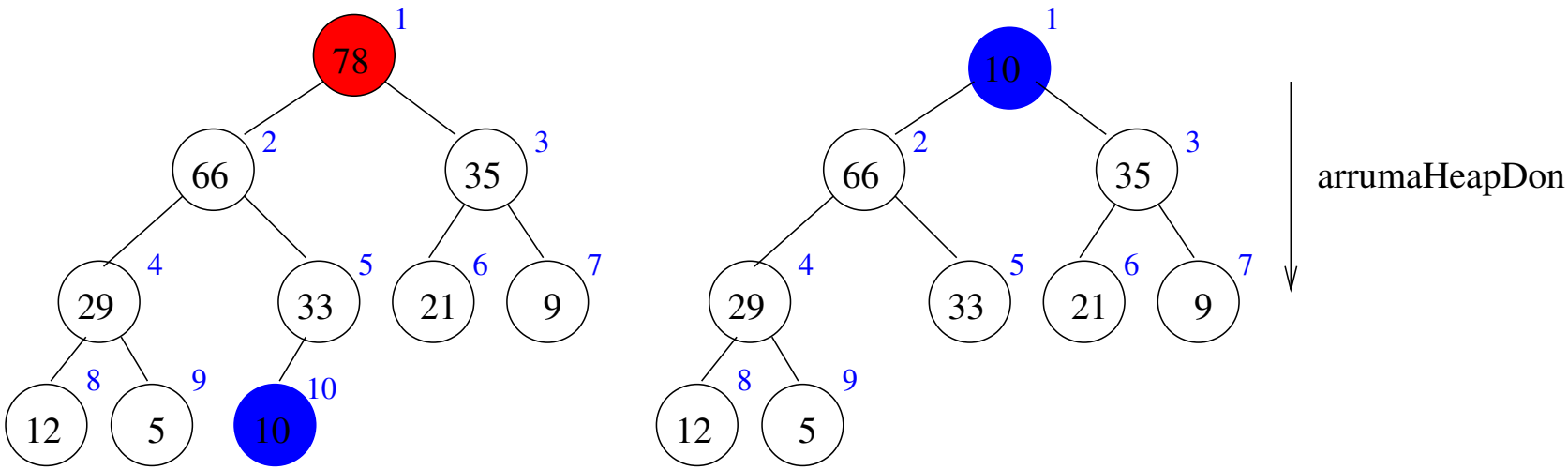
(e) We compare 69 with its parent 78.
Since 78 is larger than 69, we stop here.

(f) The final heap after inserting 69

Heap: Inserção - Implementação

```
1 void init( Heap *h ){
2   h->tamHeap= 0;
3 }
4
5 void insere( TipoItem v, Heap *h ){
6   h->tamHeap++;
7   h->elem[h->tamHeap] = v;
8   arrumaHeapUp( h, h->tamHeap );
9 }
```

Heap: Remoção



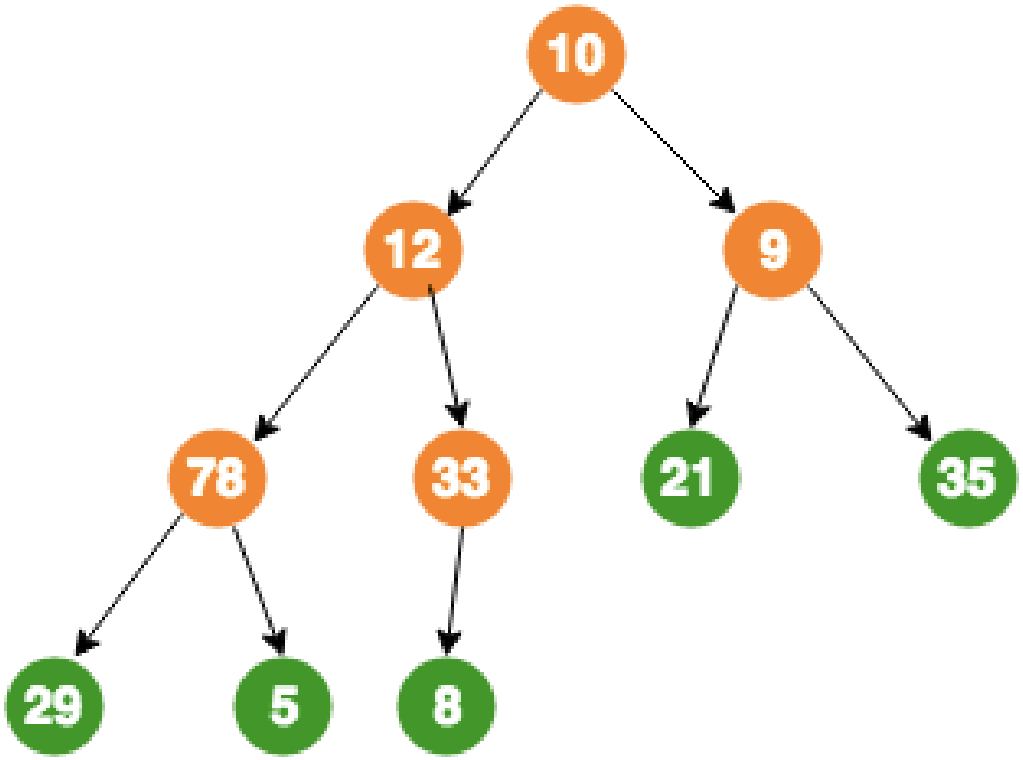
Heap: Remoção - Implementação

```
1 TipoItem removeMax( Heap *h ){
2   TipoItem v;
3
4   v= h->elem[1];
5   h->elem[1]= h->elem[h->tamHeap];
6   h->tamHeap--;
7   arrumaHeapDown( h, 1 );
8   return v;
9 }
```

Colocar um Array em Ordem Heap

Idéia:

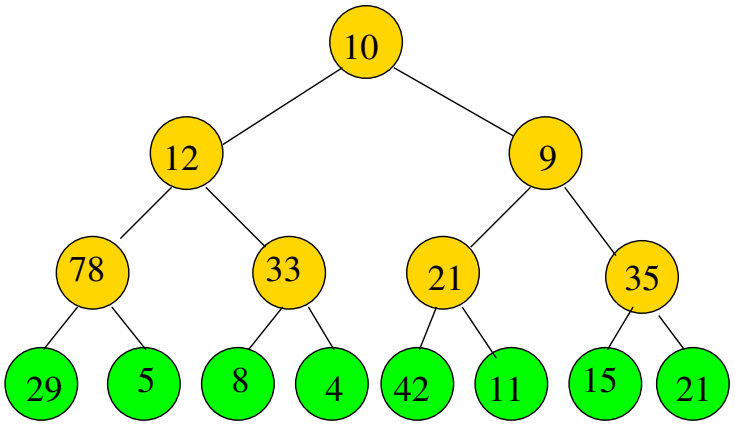
Coloca os nodos não folha em ordem-heap.



Implementação: constroiHeap

```
1 void constroiHeap( Heap *h) {
2     int i;
3
4     for (i = h->tamheap/2; i > 0; i--) {
5         arrumaHeapDown( h, i );
6     }
7 }
```

Custo da Construção do Heap



altura	#nodos	custo
4	1	3*1
3	2	2*2
2	4	1*4
1	8	0*8

Dado um heap com n nodos:

- $\#nodos(h) = \lceil \frac{n}{2^h} \rceil$
- $custoPorAltura(h) = (h-1) * c * \frac{n}{2^h}$ para uma constante c
- $custoTotal = 0c * \frac{n}{2^1} + 1c * \frac{n}{2^2} + 2c * \frac{n}{2^3} + 3c * \frac{n}{2^3} + \ldots + (h-1) * \frac{n}{2^h}$
- $custoTotal = \sum_{k=1}^{h-1} kc * \frac{n}{2^{k+1}} = \frac{nc}{2} * \sum_{k=1}^{h-1} \frac{k}{2^k}$

Custo da Construção do Heap (cont.)

$$\sum_{k=1}^{h-1} \frac{k}{2^k} < \sum_{k=1}^{\infty} \frac{k}{2^k}$$

$$\sum_{k=1}^{\infty} \frac{k}{2^k} = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} \ldots = S$$

$$\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{2}{2^3} + \frac{1}{2^4} + \frac{3}{2^4} \ldots = S$$

$$(\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \ldots) + \frac{1}{2} * (\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \ldots)$$

Primeiro termo: soma de série geométrica: $\frac{a_1}{1-r}$, onde $a = r = \frac{1}{2}$

Segundo termo: $\frac{1}{2} * S$

$$\frac{\frac{1}{2}}{1 - \frac{1}{2}} + \frac{1}{2} * S = 1 + \frac{1}{2} * S$$

$$\frac{1}{2}S = 1 \qquad S = 2$$

Custo da Construção do Heap

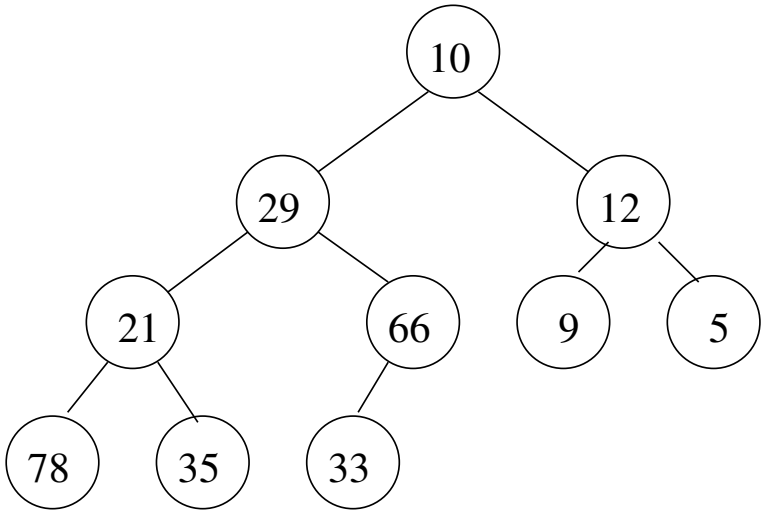
$$\sum_{k=1}^{h-1} kc * \frac{n}{2^{k+1}} = \frac{nc}{2} * \sum_{k=1}^{h-1} \frac{k}{2^k} = \frac{nc}{2} * 2 = nc$$

O custo da construção do heap é linear ($O(n)$).

HeapSort

- Colocar o array em ordem max-heap
- Como o maior elemento está na posição 1, basta trocá-lo com a sua posição correta no array ordenado
- Diminuir o tamanho do heap
- Colocar o array em ordem max-heap

Exemplo:



HeapSort - Implementação

```
1  void heapSort( Heap *h) {
2    int i, tam;
3
4    constroiHeap( h );
5    tam= h->tamHeap;
6    for (i = h->tamHeap; i >= 2; i--) {
7      troca( h->elem[1], h->elem[i] );
8      h->tamHeap--;
9      arrumaHeapDown( h, 1 );
10   }
11   h->tamHeap= tam;
12 }
```

```
1  void heapSort( Heap *h) {
2    int i, tam;
3
4    constroiHeap( h );
5    tam= h->tamHeap;
6    for (i = tam; i >= 2; i--)
7      h->elem[i]= removeMax( h );
8    h->tamHeap= tam;
9  }
```

Custo do Heapsort

- Construção do heap: $O(n)$
- Remoção de cada elemento: $n * O(lg\ n)$
- Custo total: $O(n\ lg\ n)$

Referências

- Livro Sedgewick, Cap. 9
- Livro Cormen Cap. 7