Introdução à Ciência da Computação II

Heaps & Ordenação Heap-Sort

Prof. Ricardo J. G. B. Campello

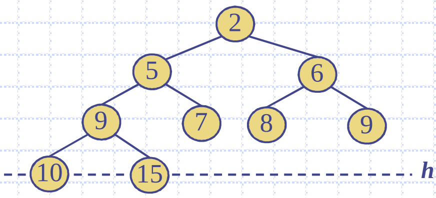
Aula de Hoje

- Heaps
  - Definição e Propriedades
  - Implementação em Vetores
- Ordenação Heap-Sort

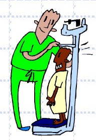
2

Heaps

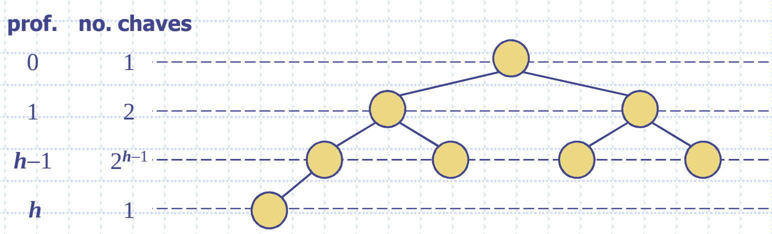
- Um **heap** é uma árvore binária que armazena chaves em seus nós e satisfaz as propriedades:
  - Ordem: para cada nó v, exceto o nó raiz, tem-se que:
    - chave(v) $\geq$ chave(pai(v))
  - Completude: é **completa**, i.e., se h é a altura, tem-se que:
    - para $i = 0, \ldots, h-1$ existem 2^i nós na profundidade i
    - na profundidade h, os nós existentes estão à esquerda dos ausentes
- Nota:
  - Ordem das chaves pode ser $\leq$



Altura de um Heap

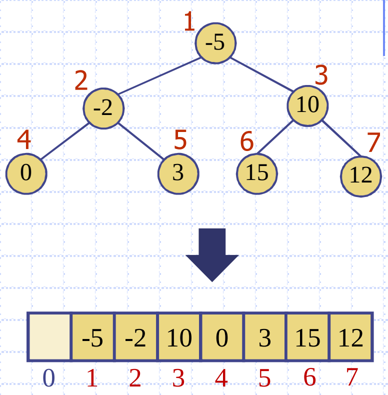


- **Teorema:**
  - Um heap armazenando n chaves possui altura h de ordem $O(\log n)$
- **Prova**:
  - Dado que existem 2^i chaves na prof. $i = 0, \ldots, h-1$ e ao menos 1 chave na prof. h, tem-se $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1 = (2^h - 1) + 1 = 2^h$
  - Logo, $n \geq 2^h$, ou seja, $h \leq \log n \Rightarrow \boxed{h \text{ é } O(\log n)}$ **log base 2!**



Estrutura em Vetor

- Utiliza o conceito de **função numeradora por nível**:
  - f(***v***) = 1 se ***v*** é a raiz da árvore ***T***
  - f(***v***) = 2f(***u***) se ***v*** é filho esq. de ***u***
  - f(***v***) = 2f(***u***)+1 se ***v*** é filho dir. de ***u***
- Cada nó da árvore é armazenado na célula de um vetor
  - O índice é dado pela função numeradora
  - O valor da função numeradora fornece imediatamente a localização dos filhos e do pai



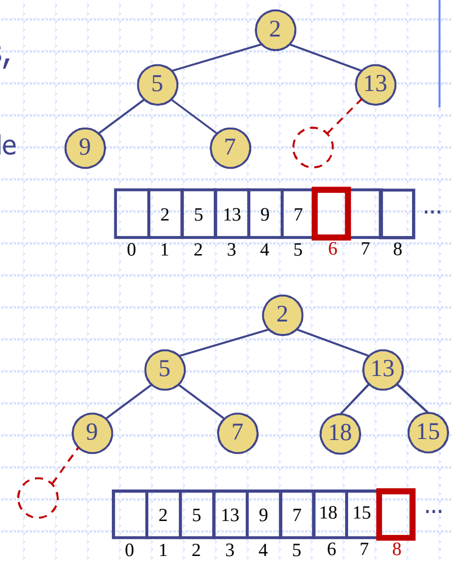
Estrutura em Vetor

- Para inserir e remover itens do heap é preciso manter controle do número ***n*** de chaves armazenadas nele
- Para isso podemos definir o heap como um registro:

```
#define MAX 1000

typedef struct {
          double V[MAX+1]; /* vetor */
          int n;      /* no. de itens */
          } Heap;

Heap H;
```

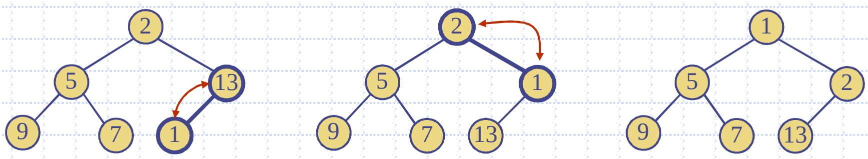
Inserção de Nova Chave

- Se o heap possui n chaves, insere-se no índice $n + 1$
  - Nunca viola a propriedade de **completude** !
  - Vide exemplos ao lado
- Mas e a propriedade de **ordem**... ?
  - Depende da chave a ser inserida...



Restauração da Ordem (bubbling-up)

- Após a inserção de um novo Item com chave k, a propriedade de ordem do heap pode ser violada
- O algoritmo **bubbling-up** (ou **upheap**) restaura a propriedade de ordem trocando os itens caminho acima a partir do nó de inserção
- Termina quando o novo item inserido com chave k alcança a raiz ou um nó cujo pai possui uma chave menor ou igual a k
- Dado que o heap possui altura $O(\log n)$, executa em tempo $O(\log n)$



Rotina de Inserção

```
int inserir(Heap *H, double k){
   if (H->n < MAX){
      H->V[H->n + 1] = k; /* insere item */
      H->n++;             /* atualiza n */
      bubbling_up(H);     /* restaura heap */
      return 1;
   }
   else return 0;         /* heap cheio */
}
```

9

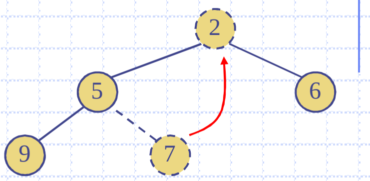
Rotina de Restauração de Ordem

```
void bubbling_up(Heap *H){

     /* No quadro... */

}
```

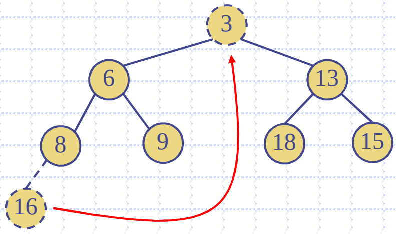
10

Remoção de Chave

- Em heaps, a remoção é normalmente da chave que se encontra na raiz
  - menor (ou maior) chave
  - filas de prioridade...

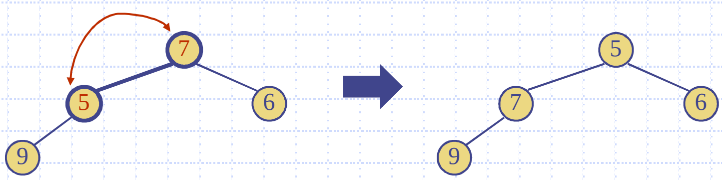


- Se o heap possui n chaves, remove-se o item no índice 1 (a raiz) e substitui-se por aquele no índice n
- Mantém a **completude**, mas e a **ordem**... ?



Restauração da Ordem (bubbling-down)

- Após a remoção, a propriedade de ordem do heap pode ser violada
- O algoritmo **bubbling-down** (ou **downheap**) restaura a prop. de ordem trocando os itens caminho abaixo a partir da raiz
  - Termina quando o item de chave k movido para a raiz alcança um nó folha ou um nó que não possui filho com chave menor do que k
  - Quando ambos os filhos (esquerdo e direito) possuem chave menor do que k, a troca é feita com o filho de menor chave
- Dado que o heap possui altura $O(\log n)$, executa em tempo $O(\log n)$



Rotina de Remoção

```
double remover(Heap *H){
   double raiz;
   if (H->n > 0){
      raiz = H->V[1];         /* guarda raiz */
      H->V[1] = H->V[H->n]; /* troca raiz */
      H->n--;               /* atualiza n */
      if (H->n > 1)
         bubbling_down(H); /* restaura heap */
      return raiz;
   }
   else printf("Heap Vazio!");
}
```

Rotina de Restauração de Ordem

```
void bubbling_down(Heap *H){

     /* Exercício... */

}
```

Análise

- Como se tem acesso direto aos índices da raíz (1), nó de inserção $(\boldsymbol{n}+1)$ e nó de remoção $(\boldsymbol{n})$ do heap, as ações de inserção e remoção propriamente ditas consomem tempo $\boldsymbol{O}(1)$
  - Mas inserir e remover executam em tempo $\boldsymbol{O}(\log \boldsymbol{n})$ devido à necessidade de restauração de ordem do heap

Operação	Tempo
checar no. elementos	$\boldsymbol{O}(1)$
checar se vazio	$\boldsymbol{O}(1)$
inserir	$\boldsymbol{O}(\log \boldsymbol{n})$
remover	$\boldsymbol{O}(\log \boldsymbol{n})$

15

Heap-Sort

- Podemos utilizar um heap para ordenar uma coleção de itens / chaves:

1. Insira os itens no heap um a um via uma série de operações inserir (Fase 1)
2. Obtenha os elementos via uma série de operações remover (Fase 2)

Algoritmo ***HeapSort***$(\boldsymbol{V}, \boldsymbol{n})$
 Entrada: Vetor $\boldsymbol{V}$ de $\boldsymbol{n}$ itens
 Saída: Vetor $\boldsymbol{V}$ ordenado
 $\boldsymbol{H} \leftarrow$ novo heap
 para $\boldsymbol{i} \leftarrow 0$ **até** $\boldsymbol{n} - 1$ **faça**
  inserir$(\boldsymbol{H}, \boldsymbol{V}[\boldsymbol{i}])$
 para $\boldsymbol{i} \leftarrow 0$ **até** $\boldsymbol{n} - 1$ **faça**
  $\boldsymbol{V}[\boldsymbol{i}] \leftarrow$ ***remover***$(\boldsymbol{H})$

Ordem crescente para heap com chave mínima na raiz e decrescente para heap com chave máxima na raiz

16

Análise Heap-Sort

- Em ambas as fases de Heap-Sort, cada uma das operações correspondentes (inserir ou remover) será $O(\log q)$
  - onde q é o tamanho do heap no momento da operação
- Dado que são efetuadas n operações em cada fase, tem-se:

$$O\left(\sum_{q=1}^{n} \log q\right) \Rightarrow O\left(\log \prod_{q=1}^{n} q\right) \Rightarrow O(\log n!) \Rightarrow \boxed{O(n \log n)}$$

17

Observações

- Heap-sort é em geral muito mais rápido que algoritmos de ordem quadrática ($O(n^2)$), como insertion-sort e selection-sort:
  - exceto para n muito pequeno
  - constantes de tempo podem ficar significativas
- 1a fase pode ser ainda mais eficiente ($O(n)$) através de uma técnica alternativa (*bottom-up*) para a construção do heap
  - mas a ordem do algoritmo continua $O(n \log n)$ devido à 2a fase
  - logo, essa melhoria não é significativa para n grande
- Pode ser implementado de forma **In-Place** (vide exercícios)

18

Comparações de Desempenho

- Segundo (Ziviani, 2004), pág. 97, os métodos de ordenação podem ser divididos em métodos **simples** e **eficientes**
- Métodos simples são aqueles que, conforme o próprio nome sugere, são intuitivos e de implementação fácil, mas possuem tempo $O(n^2)$
- Apesar do comportamento assintótico ruim, os métodos simples podem ser mais rápidos que métodos com comportamento assintótico melhor na ordenação de seqüências pequenas
- Métodos eficientes são aqueles que possuem tempo $O(n \log n)$
- Daqueles vistos, são considerados simples os algoritmos da bolha (bubble-sort), seleção (selection-sort) e inserção (insertion-sort)
- São eficientes: heap-sort, merge-sort e quick-sort

19

Comparações (cont.)

- Complexidade $O(n^2)$ dos métodos da bolha, selection e insertion-sort, para o pior caso, caso médio* e melhor caso (selection-sort e bolha), inviabiliza sua utilização em seqüências grandes e/ou quando se deseja garantir um limite superior de tempo razoável
- São os métodos indicados (em especial insertion-sort) para a ordenação de seqüências pequenas ou quando estabilidade é obrigatória
- O tempo $O(n \log n)$ de heap-sort, merge-sort e quick-sort é excelente considerando que é possível demonstrar que não existe algoritmo de ordenação baseado em comparações com tempo assintótico inferior a este

20

* Valor esperado (considerando uma seqüência a ser ordenada aleatória)

Comparações (cont.)

- (Ziviani, 2004), pág. 117, menciona que análises experimentais sugerem que heap-sort é, em média, em torno de 2 vezes mais lento que quick-sort
- Segundo (Goodrich & Tamassia, 2002/2005), experimentos sugerem que, quando a seqüência cabe toda na memória, o algoritmo quick-sort *in-place** é usualmente mais eficiente que heap-sort *in-place*, sendo considerado, em média, o algoritmo mais rápido de ordenação nessas condições
- O problema de quick-sort, devido à sua complexidade $O(n^2)$ no pior caso, é quando necessitamos garantir fortemente um limitante superior razoável para o tempo de execução

21

Comparações (cont.)

- Heap-sort se torna a alternativa ideal quando a seqüência cabe toda na memória e queremos garantir um limite de tempo superior razoável, pois seu tempo $O(n \log n)$ aliado à sua implementação *in-place* torna este um dos mais rápidos algoritmos de ordenação
- Segundo (Goodrich & Tamassia, 2002/2005), a implementação *in-place* de merge-sort, embora possível, não é competitiva devido à sua complexidade
- Logo, a versão seqüencial (não recursiva) de merge-sort se torna o algoritmo ideal quando a seqüência não cabe toda na memória principal e deve ser recuperada de um dispositivo secundário
- O tempo $O(n \log n)$ de merge-sort, aliado à sua capacidade de acesso seqüencial aos dados, torna este um dos mais rápidos algoritmos de ordenação nessas situações

22

Resumo dos Principais Algoritmos de Ordenação

Segundo (Goodrich & Tamassia, 2002/2005)

Algoritmo	Tempo	Notas
insertion-sort	$O(n^2)$ $\Omega(n)$	◈ simples ◈ ∃ *in-place* ◈ para pequenas BD (< 1K)
heap-sort	$O(n \log n)$	◈ eficiente ◈ ∃ *in-place* ◈ para grandes BD (1K — 1M)
quick-sort	$O(n^2)$ $O(n \log n)$ média	◈ eficiente ◈ ∃ *in-place** ◈ para grandes BD (1K — 1M)
merge-sort	$O(n \log n)$	◈ eficiente ◈ acesso seqüencial aos dados ◈ para BD muito grandes (> 1M)

* Tabelas com experimentos para BDs de diferentes tamanhos são apresentadas em (Ziviani, 2004), p. 115

23

Exercícios

- Seja o seguinte conjunto de itens com chaves numéricas: (4) , (-10) , (50) , (28) , (0) , (12) , (7)
  - Insira esses itens um a um em uma árvore heap (heap ascendente, ou seja, chave mínima na raiz), nessa ordem
    - Ilustre a configuração da **árvore** heap após cada inserção e também o **vetor** correspondente
  - Remova um a um os itens do heap
    - Ilustre a configuração da **árvore** heap após cada remoção e também o **vetor** correspondente
- Repita o exercício anterior para outras seqüências de chaves à sua escolha

Exercícios

- A rotina de remoção do heap vista em aula apenas apresenta um alerta se o heap estiver vazio. Modifique esta rotina para que um ponteiro nulo seja retornado nesse caso. Se o heap não estiver vazio, a rotina deve alocar dinamicamente espaço em memória, armazenar nele o elemento a ser removido do heap (raiz) e retornar um ponteiro para o respectivo endereço
- Implemente em C a rotina `bubbling_down`, que restaura a propriedade de ordem do heap após uma remoção
- Implemente em C uma versão **In-Place** do algoritmo HeapSort. Dicas: (i) Modifique o struct Heap substituindo o vetor por um ponteiro que apontará para o próprio vetor a ser ordenado; (ii) Note que, na fase 1, o i-ésimo elemento a ser inserido no heap será inserido na i-ésima posição do vetor, e que apenas os elementos anteriores poderão ser afetados pela restauração de ordem; (iii) Note que a cada remoção a célula correspondente no final do heap fica disponível

Bibliografia

- M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in C++/Java*, John Wiley & Sons, 2002/2005
- N. Ziviani, *Projeto de Algoritmos*, Thomson, 2a. Ed, 2004