Algoritmos e Estruturas de Dados II

Filas de Prioridade & Heaps

Ricardo J. G. B. Campello

Parte deste material é baseado em adaptações e extensões de slides disponíveis em http://ww3.datastructures.net (Goodrich & Tamassia).

TAD Fila de Prioridade

- Armazena **Itens**.
- **Item**: par (chave, informação).
- Operações principais:
  - remove(F): remove e retorna o item com maior prioridade (menor ou maior chave) da fila F.
  - insere(F, **x**): insere um item **x** = (k,e) com chave k.
- Operações auxiliares:
  - proximo(F): retorna o item com menor (maior) chave da fila F, sem removê-lo.
  - conta(F), vazia(F).

2

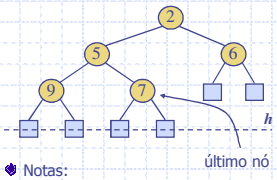
TAD Fila de Prioridade

- Diferentes Realizações
  - Estáticas
    - Lista estática (arranjo) ordenada
    - Lista estática (arranjo) não ordenada
    - **Heap em arranjo**
  - Dinâmicas
    - Lista dinâmica ordenada
    - Lista dinâmica não ordenada
    - **Heap dinâmico**
- Cada realização possui vantagens e desvantagens

3

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) ≥ 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 de profundidade i
    - na profundidade h, os nós existentes estão à esquerda dos ausentes
- Convenciona-se aqui:
  - chaves nos nós internos
  - último nó: nó interno mais à direita de profundidade $h-1$

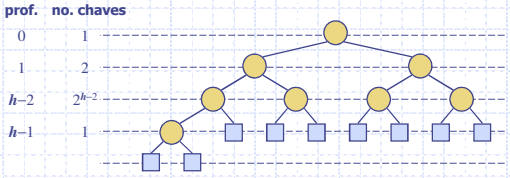


- Notas:
  - Ordem das chaves pode ser ≤
  - Folhas não precisam de fato ocupar espaço de memória

Altura de um Heap

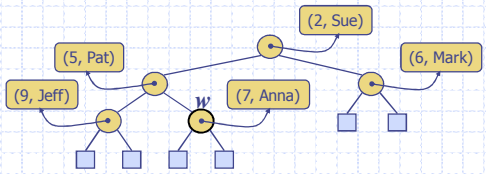
- **Teorema:**
  - Um heap armazenando n chaves possui altura h de ordem $O(\log n)$.
- **Prova**:
  - Dado que existem 2^i chaves na prof. $i = 0, \ldots, h-2$ e ao menos 1 chave na prof $h-1$, tem-se $n \geq 1+2+4+\ldots+2^{h-2}+1 = (2^{h-1}-1)+1 = 2^{h-1}$.
  - Logo, $n \geq 2^{h-1}$, i.e., $h \leq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

log base 2!



Filas de Prioridade com Heaps

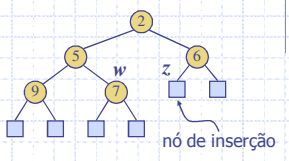
- Armazena-se um Item (chave, informação) em cada nó interno.
- Mantém-se o controle sobre a localização do último nó (w).
- Remove-se sempre o Item armazenado na raiz, devido à propriedade de ordem do heap.
  - Fila Ascendente: Menor chave (maior prioridade) na raiz do heap.
  - Fila Descendente: Maior chave (maior prioridade) na raiz do heap.

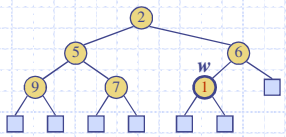


6

Inserção

- Método insere do TAD fila de prioridade corresponde à inserção de um Item com chave k no heap.
- O algoritmo de inserção consiste de 3 passos:
  - Encontrar nó de inserção z
    - novo último nó w
  - Armazenar o Item com chave k em z e expandir z como um nó interno
  - Restaurar ordem do heap
    - discutido a seguir

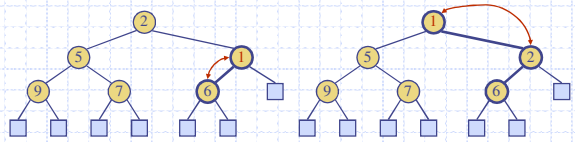




Por simplicidade apenas as chaves estão sendo mostradas nos nós do heap.

Restauração da Ordem (bubbling-up)

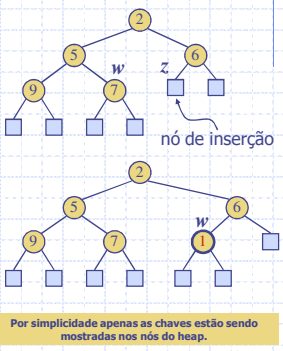
- Após a inserção de um novo Item com chave k, a propriedade de ordem do heap pode ser violada.
- O algoritmo **bubbling-up** (ou **upheap**) restaura a propriedade de ordem trocando os itens caminho acima a partir do nó de inserção.
- Termina quando o Item de chave k inserido alcança a raiz ou um nó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



Inserção

Algoritmo insert(F, *x*)
  z ← *findInsertionNode*(F)
  expandExternal(F, *z*)
  replace(F, *z*, *x*)
  F.lastnode ← *z* // atualiza nó w
  upHeap(F) // bubbling up

Notas:
- Fila de Prioridade é Heap
- Heap é Árvore Binária
- Logo:
  - Fila de Prioridade usa TAD AB
  - Acrescenta controle de último nó
    - Campo / Atributo **F.lastnode**



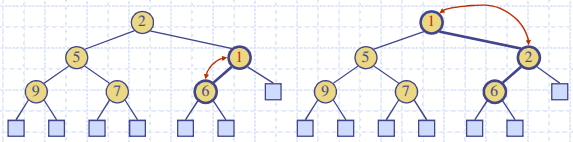
Por simplicidade apenas as chaves estão sendo mostradas nos nós do heap.

9

Restauração da Ordem (bubbling-up)

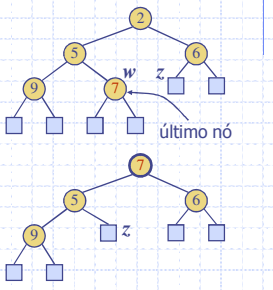
Algoritmo *upHeap*(F)
  w ← F.lastnode
  enquanto (**não** *isRoot*(F, *w*)) **e** (*key*(F, *w*) > *key*(F, *parent*(F, *w*))) **faça**
    swap(F, *w*, *parent*(F, *w*))
    w ← *parent*(F, *w*)

Nota:
- Acrescenta ao TAD AB operação *key*(F, p) que retorna chave do nó referenciado por p.



Remoção

- Método remove do TAD fila de prioridade corresponde à remoção do Item da raiz.
- O algoritmo de remoção consiste de 3 passos:
  - Substituir o Item da raiz com aquele do último nó *w*.
  - Transformar *w* e seus filhos em um nó externo:
    - novo nó de inserção *z*.
  - Restaurar ordem do heap
    - discutido a seguir.

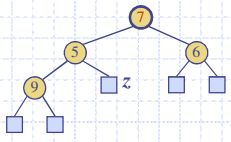


11

Remoção

- Método remove do TAD fila de prioridade corresponde à remoção do Item da raiz.
- O algoritmo de remoção consiste de 3 passos:
  - Substituir o Item da raiz com aquele do último nó *w*.
  - Transformar *w* e seus filhos em um nó externo:
    - novo nó de inserção *z*.
  - Restaurar ordem do heap
    - discutido a seguir.

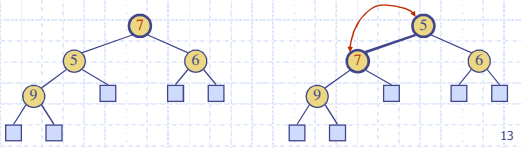
Utilizar o último nó para substituição da raiz na remoção possui várias vantagens, entre elas:
- Completude garantida com a simples operação de fusão (passo 2).
- Retorno automático do nó de inserção *z*.
- Implementação em tempo constante através de arranjo (discutida posteriormente).



12

Restauração da Ordem (bubbling-down)

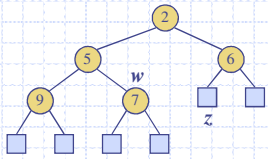
- Após a remoção, a propriedade de ordem do heap pode ser violada.
- O algoritmo **bubbling-down** (ou **downheap**) restaura a propriedade de ordem trocando os itens caminho abaixo a partir da raiz.
  - Termina quando o Item de chave k movido para a raiz alcança um nó que não possui filho interno com chave menor do que k.
  - Quando ambos os filhos são internos e 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 $O(\log n)$.

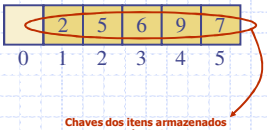


13

Implementação em Arranjo

- Implementação segue aquela de árvore binária:
  - Função numeradora por nível
- Operação insere:
  - insere no índice $n + 1$
    - acesso direto ao nó z !
- Operação remove:
  - remove do índice 1 e substitui por aquele do índice n
    - acesso direto à raiz e a w !





Chaves dos itens armazenados nos nós do heap.

14

Análise

- Como se tem acesso direto aos índices da raíz (1), nó de inserção (n+1) e último nó (n) do heap, as ações de inserção e remoção propriamente ditas consomem tempo $O(1)$.
- Mas insere e remove executam em tempo $O(\log n)$ devido à restauração de ordem do heap.
- proximo não requer restauração de ordem e portanto é $O(1)$.

Tempos de Execução

Operação	Tempo
conta, vazia	$O(1)$
proximo	$O(1)$
insere	$O(\log n)$
remove	$O(\log n)$

- A tabela acima vale também para a implementação dinâmica do heap.
- O único esforço adicional advém da atualização do último nó e do nó de inserção nas operações de inserção e remoção.
- Como vimos, esse esforço também é da ordem $O(\log n)$, e portanto não altera a ordem de complexidade dos respectivos métodos.

15

Comparação de Filas de Prioridade

Via Lista Não-Ordenada

Operação	Tempo
proximo	$O(n)$
insere	$O(1)$
remove	$O(n)$

Via Lista Ordenada

Operação	Tempo
proximo	$O(1)$
insere	$O(n)$
remove	$O(1)$

Via Heaps

Operação	Tempo
proximo	$O(1)$
insere	$O(\log n)$
remove	$O(\log n)$

16

Revisão (Ordenação via Filas de Prioridade)

- Podemos utilizar uma fila de prioridade para ordenar uma coleção de elementos comparáveis:
  1. Insira os elementos na fila um a um via uma série de operações insere (Fase 1).
  2. Retorne os elementos via uma série de operações remove (Fase 2).
- O tempo de execução deste método depende da implementação da fila de prioridade.

Algoritmo *PQ-Sort*(L)
Entrada: lista L de itens
Saída: lista L ordenada.
$F \leftarrow$ nova fila de prioridade
enquanto não *lista_vazia*(L)
  $x \leftarrow$ *remover_frente*(L)
  insere(F, x)
enquanto não *vazia*(F)
  $x \leftarrow$ *remove*(F)
  inserir_final(L, x)

Ordem crescente para fila ascendente (chave mínima) e decrescente para fila descendente (chave máxima)

17

Heap-Sort

- Utilizando uma fila de prioridade baseada em heap, podemos ordenar uma seqüência de n elementos através do algoritmo PQ-Sort em tempo $O(n \log n)$.
- Em ambas as fases de PQ-Sort, cada uma das operações correspondentes (insere ou remove) será $O(\log q)^*$.
- Dado que são efetuadas n operações em cada fase, tem-se:

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

- O algoritmo resultante é chamado **heap-sort**.

* PS. q é o tamanho da fila de prioridade no momento da operação.

18

Heap-Sort

- Heap-sort é em geral muito mais rápido que algoritmos de ordenação quadráticos ($O(n^2)$) , como insertion-sort e selection-sort:
  - Exceto para n muito pequeno (constantes de tempo ficam significativas).
- Sua 1a fase pode ser ainda mais eficiente ($O(n)$) através de uma técnica alternativa para a construção do heap chamada *bottom-up*
  - mas a ordem do algoritmo continua $O(n \log n)$ devido à 2a fase
  - logo, essa melhoria não é significativa para n grande
- Também admite realização tipo **in-place**, que não constrói um heap explicitamente mas apenas rearranja os elementos na seqüência original
  - Cormen et al. "Introduction to Algorithms", MIT Press, 2001.
- Assim como em outros algoritmos, se os elementos forem muito grandes para serem eles próprios movimentados, pode-se movimentar (ordenar) apenas uma seqüência de apontadores ou referências para os mesmos
  - **ordenação indireta**

19

Exercícios

- Seja o seguinte conjunto de itens com chaves numéricas e informação dada por caracteres:
  - (4,A) , (-10,J) , (50,C) , (28,K) , (0,P) , (12,U) , (7,L)
  - Insira esses itens um a um em uma árvore heap, nessa ordem, respeitando a ascendência das chaves
    - Ilustre a configuração do heap após cada inserção
  - Remova um a um os itens do heap segundo a ordem de prioridade ascendente (menor chave = maior prioridade)
    - Ilustre a configuração do heap após cada remoção