Algoritmos e Estruturas de Dados: Heap

Rômulo Silva de Oliveira
Departamento de Automação e Sistemas – DAS – UFSC

romulo@das.ufsc.br
http://www.das.ufsc.br/~romulo
Maio/2011

Referências

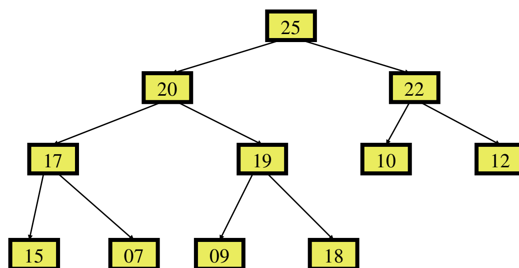
- Mastering Algorithms with C
  Kyle Loudon
  O´Reilly, 1999
- Livros de algoritmos e estruturas de dados em geral

Heap – Introdução 1/13

- Heap é uma árvore na qual cada nodo filho tem um valor menor que o nodo pai
- O nodo raiz é o maior em toda a árvore
- Possível inverter a orientação
  - Cada nodo filho tem um valor maior que seu nodo pai
- Heaps são *partially ordered* pois
  - Nodos ao longo de cada galho tem uma ordenação específica
  - Nodos de um nível não estão necessariamente ordenados com relação aos nodos de outro nível
- Heap onde cada filho é menor que seu pai: *top-heavy*
- Heap onde cada filho é maior que seu pai: *bottom-heavy*

Heap – Introdução 2/13

- Heap top-heavy

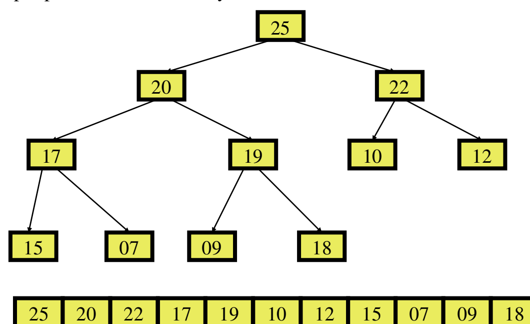


Heap – Introdução 3/13

- Uma forma de representar heaps é armazenar os nodos contiguamente em um array
- Na ordem que eles são encontrados na caminhada transversal
- O pai de cada nodo na posição i do array está posicionado em $(i - 1)/2$
- Os filhos esquerdo e direito de um nodo estão localizados nas posições
  $2i + 1$ e $2i + 2$
- O último nodo é o nodo mais a direita no último nível

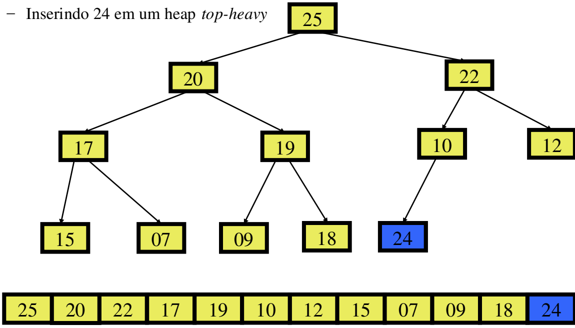
Heap – Introdução 4/13

- Heap representado como array



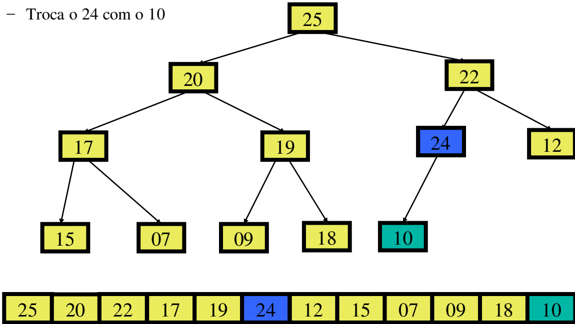
Heap – Introdução 5/13

- Inserção no Heap
  - Inserindo 24 em um heap *top-heavy*



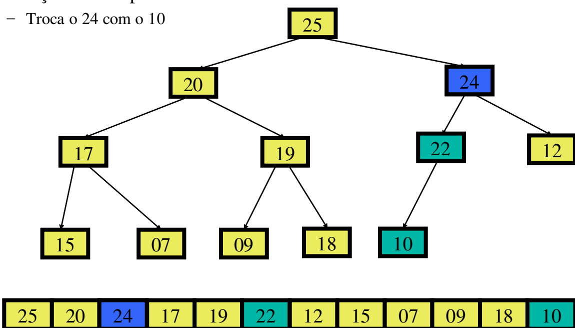
Heap – Introdução 6/13

- Inserção no Heap
  - Troca o 24 com o 10



Heap – Introdução 7/13

- Inserção no Heap
  - Troca o 24 com o 10



Heap – Introdução 8/13

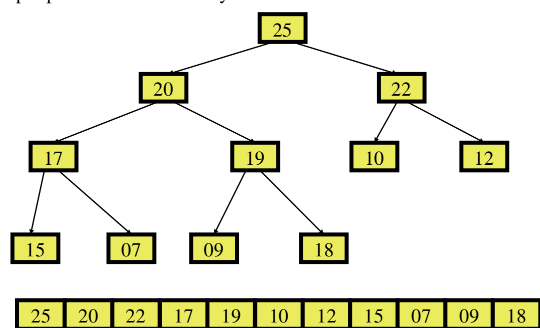
- Ajuste após inserir um nodo
- Necessário considerar apenas o galho no qual o nodo foi inserido
- Move-se para cima nível a nível
  - Compara o nodo filho com o nodo pai
- Se a ordem entre pai e filho está errada: Troca de lugar
- Continua até que nenhuma troca é necessária, ou chega-se no topo da árvore

Heap – Introdução 9/13

- A remoção ocorre do nodo no topo
- O conteúdo do último nodo é copiado para o topo
- O heao precisa ser reajustado
- O ajuste inicia pelo nodo raiz e continua em direção aos níveis mais baixos
- Em cada nível, se o nodo e os nodos filhos estão na ordem errada
  - Seus conteúdos são trocados
  - Continua-se com o nodo filho mais fora de ordem
  - Até chegar nas folhas, ou não ser necessário fazer uma troca

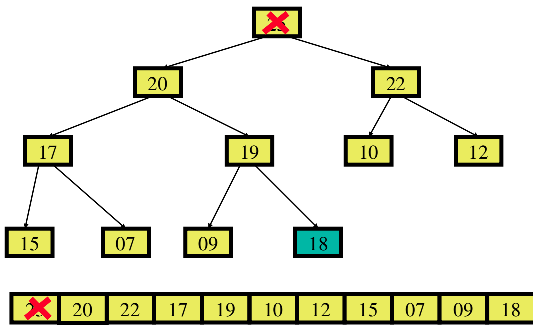
Heap – Introdução 10/13

- Heap representado como array



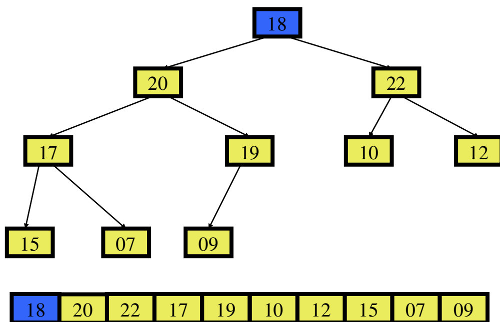
Heap – Introdução 11/13

- Elimina o 25



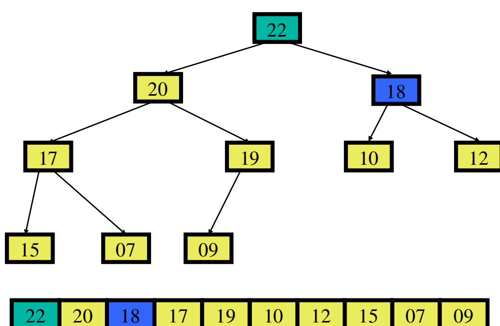
Heap – Introdução 12/13

- Heap representado como array



Heap – Introdução 13/13

- Troca o 18 com o 22



Heap – Interface 1/5

- ***heap_init***
  - void heap_init(Heap *heap,
    int (*compare)(const void *key1, const void *key2),
    void (*destroy)(void *data));
- **Retorno**
  - Nenhum.
- **Descrição**
  - Inicializa o heap especificado por *heap*. Esta operação deve ser chamada para um heap antes que o heap possa ser usado com qualquer outra operação. O argumento *compare* é uma função usada por várias operações sobre o heap para comparar nodos quando ajustando o heap. Esta função deve retornar 1 se key1>key2, 0 se key1=key2 e -1 se key1<key2 para um heap "top-heavy". Para um heap "bottom-heavy", *compare* deve inverter os casos que retornam +1 e -1. O argumento *destroy* prove uma maneira para liberar dados alocados dinamicamente quando *heap_destroy* é chamada. Por exemplo, se o heap contem dados dinamicamente alocados usando *malloc*, *destroy* deverá ser igual a *free* para liberar os dados quando o heap for destruido. Para dados estruturados contendo diversos membros alocados dinamicamente, *destroy* deverá ser uma função definida pelo usuário que chama *free* para cada membro alocado dinamicamente assim como para a própria estrutura. Para um heap contendo dados que não devem ser liberados, *destroy* deve ser NULL.

Heap – Interface 2/5

- ***heap_destroy***
  - void heap_destroy(Heap *heap);
- **Retorno**
  - Nenhum.
- **Descrição**
  - Destrói o heap especificado por *heap*. Nenhuma outra operação é permitida após a chamada de *heap_destroy* a não ser que *heap_init* seja chamada novamente. A operação *heap_destroy* remove todos os nodos do heap e chama a função passada como *destroy* para *heap_init* uma vez para cada nodo quando o mesmo é removido, desde que *destroy* não seja NULL.

Heap – Interface 3/5

- ***heap_insert***
  - int heap_insert(Heap *heap, const void *data);
- **Retorno**
  - 0 se a inserção do nodo teve sucesso, ou -1 caso contrário.
- **Descrição**
  - Insere um nodo no heap especificado por *heap*. O novo nodo contem um pointer para os dados, logo a memória referenciada por *data* deverá permanecer válida pelo tempo que o nodo permanecer no heap. É responsabilidade do chamador gerenciar a memória associada com os dados.

Heap – Interface 4/5

- ***heap_extract***
  - int heap_extract(Heap *heap, void **data);
- **Retorno**
  - 0 se a extração do nodo teve sucesso, ou -1 caso contrário.
- **Descrição**
  - Extrai o nodo do topo do heap especificado por *heap*. Após o retorno, *data* aponta para os dados armazenados no nodo que foi extraido. É responsabilidade do chamador gerenciar a memória associada com os dados.

Heap – Interface 5/5

- ***heap_size***
  - int heap_size(const Heap *heap);
- **Retorno**
  - Número de nodos no heap.
- **Descrição**
  - Macro que avalia o número de nodos no heap especificado por *heap*.

Heap – Header 1/2

```
#ifndef HEAP_H
#define HEAP_H

typedef struct Heap_ {

    int         size;

    int         (*compare)(const void *key1, const void *key2);
    void         (*destroy)(void *data);
    void         **tree;

} Heap;
```


Heap – Header 2/2

```
void heap_init(Heap *heap,
                  int (*compare)(const void *key1, const void *key2),
                  void (*destroy)(void *data));

void heap_destroy(Heap *heap);

int heap_insert(Heap *heap, const void *data);

int heap_extract(Heap *heap, void **data);

#define heap_size(heap) ((heap)->size)

#endif
```


Heap – Implementação 1/10

```
#include <stdlib.h>
#include <string.h>

#include "heap.h"

//  Define private macros used by the heap implementation

#define heap_parent(npos) ((int)(((npos) - 1) / 2))

#define heap_left(npos) (((npos) * 2) + 1)

#define heap_right(npos) (((npos) * 2) + 2)
```


Heap – Implementação 2/10

```
void heap_init(Heap *heap,
        int (*compare)(const void *key1, const void *key2),
        void (*destroy)(void *data)) {

    heap->size = 0;
    heap->compare = compare;
    heap->destroy = destroy;
    heap->tree = NULL;

    return;

}
```

Heap – Implementação 3/10

```
void heap_destroy(Heap *heap) {

    int i;

    if (heap->destroy != NULL) {

        for (i = 0; i < heap_size(heap); i++) {
            heap->destroy(heap->tree[i]);
        }
    }

    free(heap->tree);
    memset(heap, 0, sizeof(Heap));
    return;
}
```


Heap – Implementação 4/10

```
int heap_insert(Heap *heap, const void *data) {

    void *temp;

    int ipos, ppos;

    if ((temp = (void **)realloc(heap->tree, (heap_size(heap) + 1) * sizeof(void *)))
                    == NULL) {
        return -1;
    }
    else {
        heap->tree = temp;
    }

    // Insert the node after the last node

    heap->tree[heap_size(heap)] = (void *)data;
```


Heap – Implementação 5/10

```
    // Heapify the tree by pushing the contents of the new node upward
    ipos = heap_size(heap);
    ppos = heap_parent(ipos);

    while (ipos > 0 && heap->compare(heap->tree[ppos], heap->tree[ipos]) < 0) {
        // Swap the contents of the current node and its parent
        temp = heap->tree[ppos];
        heap->tree[ppos] = heap->tree[ipos];
        heap->tree[ipos] = temp;

        // Move up one level in the tree to continue heapifying
        ipos = ppos;
        ppos = heap_parent(ipos);
    }

    heap->size++;

    return 0;
}
```


Heap – Implementação 6/10

```
int heap_extract(Heap *heap, void **data) {

    void *save,   *temp;

    int  ipos,  lpos,  rpos,  mpos;

    if (heap_size(heap) == 0)
        return -1;

    *data = heap->tree[0];
```


Heap – Implementação 7/10

```
    // Adjust the storage used by the heap

    save = heap->tree[heap_size(heap) - 1];

    if (heap_size(heap) - 1 > 0) {

        if ((temp = (void **)realloc(heap->tree, (heap_size(heap) - 1) * sizeof(void *)))
                        == NULL) {
            return -1;
        }
        else {
            heap->tree = temp;
        }
        heap->size--;
    }
```


Heap – Implementação 8/10

```
    else {

            // Manage the heap when extracting the last node.

            free(heap->tree);
            heap->tree = NULL;
            heap->size = 0;
            return 0;
    }
```

Heap – Implementação 9/10

```
// Copy the last node to the top

heap->tree[0] = save;

// Heapify the tree by pushing the contents of the new top downward
ipos = 0;
lpos = heap_left(ipos);
rpos = heap_right(ipos);
```


Heap – Implementação 10/10

```
    while (1) {          // Select the child to swap with the current node
        lpos = heap_left(ipos);
        rpos = heap_right(ipos);
        if (lpos < heap_size(heap) && heap->compare(heap->tree[lpos], heap-> tree[ipos]) > 0)
            mpos = lpos;
        else
            mpos = ipos;
        if (rpos < heap_size(heap) && heap->compare(heap->tree[rpos], heap->tree[mpos]) > 0)
            mpos = rpos;
        //  When mpos is ipos, the heap property has been restored
        if (mpos == ipos)
            break;
        else {                  // Swap the contents of the current node and the selected child
            temp = heap->tree[mpos];
            heap->tree[mpos] = heap->tree[ipos];
            heap->tree[ipos] = temp;
            ipos = mpos;        // Move down one level in the tree to continue heapifying
        }
    }
    return 0;
}
```


Heap – Sumário

- Estrutura simples
- Estrutura auxiliar útil em alguns casos
- Exemplo: ordenação

- Usado para implementar filas de prioridade

- Filas de prioridades são extremamente úteis