MAT A40 - Estrutura de Dados e Algoritmos I [1]

Dr. George Lima
Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade Federal da Bahia

[1] Este módulo foi baseado em conteúdos presentes em "A. V. Aho, J. D. Ullman, *Foundations of Computer Science* (C edition). Computer Science Press (W.H. Freeman), 1995.".

Módulo 7

Heap e aplicações

Informação ordenada por prioridade

- Diversas aplicações requerem a recuperação rápida de informação por prioridade.
- Listas e árvores binárias de busca podem ser usadas. Heap é uma alternativa mais eficiente.
  1. Heap é uma árvore binária ordenada que se mantém aproximadamente balanceada.
  2. A implementação de heap pode ser feita através de um vetor, o que provê eficiência tanto no armazenamento quanto na recuperação e manipulação de informações.
  3. Busca de elemento de maior prioridade é feita em $O(1)$ enquanto que remoção e inclusão custam $O(\log n)$, numa heap com n elementos.

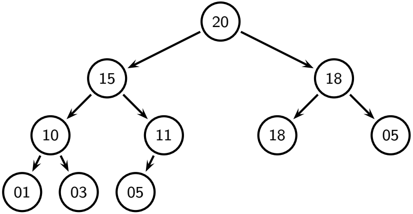
Heap

Definição

Uma heap é uma árvore binária ordenada, com cada nó tendo como identificador a prioridade associada à informação nele contida. Dois tipos de heaps são comuns, heap máxima e heap mínima. Para o primeiro tipo, para cada nó, sua prioridade nunca é menor que a prioridade de seus filhos. Na heap mínima, a prioridade de cada nó não é maior que a de seus filhos.

- Adotaremos apena a heap máxima neste módulo. Os algoritmos relacionados à heap mínima são similares.
- Duas aplicações para heaps serão abordadas, filas de prioridade e ordenação.

Heap máxima – Ilustração



representação em vetor
|20|15|18|10|11|18|05|01|03|05|

Operações típicas sobre filas de prioridades

- pq_new: cria uma nova fila de prioridades.
- pq_delete: desaloca memória e destroi uma fila de prioridades.
- pq_insert: insere um novo elemento na fila na posição correspondente a sua prioridade.
- pq_deleteMax: remove o elemento mais prioritário na heap.
- pq_getMax: retorna o elemento mais prioritário na heap.

```
typedef int * heap_t;
typedef struct cell * element_t;
struct cell {/* ... */};

typedef struct  {
  int size, maxsize; // #elem. e tam. max. da fila
  heap_t  priority; // heap com prioridades
         element_t element; // outras info da aplicacao
} pqueue_t;
```

- O comportamento das filas de prioridades é determinado basicamente pelas operações sobre a heap. Apenas estas serão descritas neste módulo.

Operações típicas sobre heap máxima

Operações básicas:

- `h_insert`: insere um novo elemento na heap.
- `h_deleteMax`: remove o elemento mais prioritário na heap.
- `heapfy`: constrói a heap a partir de uma sequência.

Operações para manutenção da ordem da heap:

- `h_bubbleUp`: movimenta um elemento de sua posição em direção à raiz preservando a ordem da heap.
- `h_bubbleDown`: movimenta um elemento de sua posição em direção à folha preservando a ordem da heap.

Fila de prioridade – definições iniciais

```c
/* Macros uteis - indices dos filhos esquerdo e
   direito e do pai do noh de indice i */
#define LEFT(i)    2*i+1
#define RIGHT(i)   2*i+2
#define PARENT(i)  (i-1)/2

typedef int * heap_t; // heap como uma sequencia de int

int h_size;  // tamanho da heap
```

Inserção numa heap máxima

```c
/* troca elementos i e j */
void h_swap(int i, int j, heap_t h) {
  int k = h[i];
  h[i] = h[j];
  h[j] = k;
}

/* Move o elemento i para cima
e preserva a ordem da heap */
void h_bubbleUp(int i, heap_t h) {

  while(i > 0 && h[i] > h[PARENT(i)]) {
    h_swap(PARENT(i), i, h);
    i = PARENT(i);
  }
}

/* Insere o elemento x na heap aumentando seu tamanho */
void h_insert(int x, int *pn, heap_t h) {
  h[*pn] = x;
  h_bubbleUp(*pn, h);
  (*pn)++;
}
```

Remoção numa heap máxima

```
/* Retorna o indice do maior filho do noh i */
int maxChild(int i, int n, heap_t h) {
  int imax = LEFT(i);
  if (imax < n-1 && h[RIGHT(i)] > h[imax])
    imax++;

   return imax;
}

/* Move o noh i para baixo presercando a ordem da heap */
void h_bubbleDown(int i, int n, heap_t h) {

  int imax = maxChild(i,n,h);

  while((imax < n) && (h[i] < h[imax])) {
    h_swap(i, imax, h);
    i = imax;
    imax = maxChild(i,n,h);
  }
}

/* Remove o maior elemento da heap. O elemento removido eh
colocado no final do vetor, apos a heap. */
void h_deletemax(int *pn, heap_t h) {

  if (*pn > 0) {
    h_swap(0, (*pn)-1, h);
    (*pn)--;
    h_bubbleDown(0, *pn, h);
  }
}
```

Exercícios

1. Construa uma fila de prioridades para uma aplicação de agendamento de atendimento para um consultório médico. Cada paciente tem um número identificador e um nome. As prioridades para atendimento são estabelecidas de acordo com uma triagem. Há 10 classes de prioridades. Assuma que as mesmas já estão associadas aos pacientes. Execute uma simulação para $n = 100$ clientes num dia de atendimento.
2. Qual a complexidade de tempo de execução das funções h_bubbleUp, h_bubbleDown, `h_deletemax` e `h_insert`?
3. Construa versões recursivas para as funções h_bubbleUp e h_bubbleDown? Discuta as diferenças em termos de tempo de execução das versões iterativas e recursivas.

Ordenação de sequência de valores

O problema de ordenação

Ordenar uma dada sequência é encontrar uma permutação da mesma que satisfaz a ordem desejada. Assume-se aqui que os elementos da sequência são comparáveis pelas relações $<$ e $\leq$ e a ordem desejada é definida de acordo com estas relações.

- Gerar permutações da entrada e verificar se estas satisfazem a ordem desejada não é viável (qual seria a complexidade de tal algoritmo?).
- O problema de ordenação de uma sequência de n valores arbitrária não pode ser resolvido em menos que $O(n \log n)$.
- Existem algoritmos (ótimos) que resolvem o problema de ordenação em $O(n \log n)$ passos.

Usando heap máxima para ordenar

Ideia básica

Considerando que a sequência de n elementos a ser ordenada forma uma heap máxima:

- Retirar os elementos máximos da heap, um por vez. O primeiro a ser retirado é colocado na n-ésima posição da sequência ordenada, o segundo na posição posição $n - 1$ e assim por diante.
- Após retirar os n elementos da heap máxima, termina-se com uma sequência em ordem não-decrescente.

- Para a abordagem acima funcionar, precisa-se garantir que os elementos a serem ordenados já estejam numa heap máxima.

Construindo uma heap máxima inicial

- Estratégia 1. Inserir os n elementos a serem ordenados numa heap máxima de tamanho n.
  - Como cada inserção e cada remoção custa $O(\log n)$ passos, o algoritmo de ordenação teria complexidade de tempo $O(n \log n)$.
  - Pode-se reduzir as constantes: pode-se arrumar o vetor ordenado como heap máxima sem precisar de vetor auxiliar para servir de heap.
- Estratégia 2. Fazer a sequência de n elementos a serem ordenados ser uma heap realizando apenas algumas trocas.
  - Metade do vetor de entrada já é heap máxima! Algumas outras poucas operações de troca de elementos impõem a ordem da heap sobre todos os elementos do vetor.

Construindo uma heap máxima inicial (em tempo linear)

Operação `heapfy`:

```
/* Move cada noh interno para baixo se este violar a
   ordem da heap. Folhas jah sao heap maximas. */
void heapfy(int n, heap_t h) {

  for (int i = n/2; i > -1; i--)
    h_bubbleDown(i, n, h);
}
```

- Como `h_bubbleDown` executa em não mais que $O(\log n)$ e há $O(n)$ chamadas à `h_bubbleDown`, `heapfy` executa em $O(n \log n)$.
- Mas, é possível verificar que o limite assintótico de `heapfy` é $O(n)$!
- Não é necessário alocar outro espaço. O vetor de entrada é transformado em heap máxima!

Análise da complexidade de `heapfy`

Pior caso:

- para cada nó folha, `h_bubbleDown` itera 0 vezes (nem é chamada); há $\frac{n}{2}$ nós folhas;
- para cada nó com altura 1, `h_bubbleDown` itera no máximo 1 vez; há $\frac{n}{2^2}$ nós com altura 1;
- para cada nó com altura 2, `h_bubbleDown` itera no máximo 2 vezes; há $\frac{n}{2^3}$ nós no nível 2;
- para cada nó com altura l, `h_bubbleDown` itera l vezes; há $\frac{n}{2^{l+1}}$ nós no nível l;
- para o nó raiz (altura $\log_2 n$), `h_bubbleDown` itera $\log_2 n$ vezes.
- Portanto, o número total de iterações de `h_bubbleDown` será

$$\sum_{i=0}^{\log_2 n} i \frac{n}{2^{i+1}} = \frac{n}{2} \sum_{i=1}^{\log_2 n} \frac{i}{2^i} \leq \frac{n}{2} \sum_{i=1}^{\infty} \frac{i}{2^i}$$

Análise da complexidade de `heapfy`

Calculando...

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16} + \cdots$$

$$\begin{array}{ccccccccccccc}
1/2 & + & 1/4 & + & 1/4 & + & 1/16 & + & \cdots & = & 1 \\
 & & 1/4 & + & 1/8 & + & 1/16 & + & \cdots & = & 1/2 \\
 & & & & 1/8 & + & 1/16 & + & \cdots & = & 1/4 \\
 & & & & & & 1/16 & + & \cdots & = & 1/8 \\
 & & & & & & & & \cdots & = & \cdots
\end{array}$$

Então,

$$\frac{n}{2} \sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{n}{2} \left(1 + \sum_{i=1}^{\infty} \frac{1}{2^i} \right) = \frac{n}{2} \times 2 = n$$

Portanto, o custo de `heapfy` é $O(n)$.

Ordenação usando heap

Algoritmo de ordenação com tempo de execução $O(n \log n)$:

```
void heapsort(int n, int v[]) {

  heapfy(n,v); // O(n)
  int i = n;
  while(i) // itera n vezes
    h_deletemax(&i, heap_t v); // O(log n)
}
```

- Ordem não-decrescente com heap máxima.
- Ordem não-crescente com heap mínima.
- Algoritmo rápido de ordenação, usando o próprio espaço da sequência de entrada (ordenação *in place*).

Exercícios

- Modifique as funções usadas na manipulação de heap para versões recursivas e comente sobre possíveis diferenças em desempenho comparando-as com as versões iterativas apresentadas neste módulo. Para tanto, conduza experimentos para vários valores de n, medindo o tempo médio de execução destas versões.
- Construa as funções que operam sobre heap estudadas neste módulo para que as mesmas utilizem a representação explícita de árvores (com o uso de ponteiros).
- A árvore de pesquisa binária pode ser usada como implementação de filas de prioridades. Mostre as operações `insert` e `deletemax` sobre tal implementação alternativa. Qual a complexidade destas funções? Compare-as com as estudadas neste módulo.
- Derive um algoritmo para fornecer os $\sqrt{n}$ maiores elementos de uma sequência numérica de n valores. O tempo de execução deste algoritmo não deve ser superior a $O(n)$.