Heap binária
Filas de prioridades
Heapsort

Prof. Tiago Massoni

Engenharia da Computação

Poli - UPE

Heapsort

- Possui o mesmo princípio de funcionamento da ordenação por seleção
- Algoritmo:

1. Selecione o menor Comparable do vetor.
2. Troque-o com o Comparable da primeira posição do vetor.
3. Repita estas operações com os n – 1 itens restantes, depois com os n – 2 itens, e assim sucessivamente.

- O custo para encontrar o menor (ou o maior) Comparable entre n itens é n – 1 comparações
- Isso pode ser reduzido utilizando uma fila de prioridades
  - Veremos esta estrutura antes de voltar ao algoritmo

Filas de prioridades

- Estrutura onde a chave de cada Comparable reflete sua habilidade de abandonar o conjunto de itens rapidamente
- Aplicações:
  - Sistemas operacionais - as chaves representam o tempo em que eventos devem ocorrer
  - Filas de impressão

Filas de prioridades TAD: operações

- Construir uma fila de prioridades a partir de um conjunto com n itens
- Informar maior Comparable do conjunto
- Retirar o Comparable com maior chave
- Insere um novo Comparable
- Aumentar o valor da chave do Comparable i para um novo valor que é maior que o valor atual da chave
- Alterar a prioridade de um Comparable
- Ajuntar duas filas de prioridades em uma única

Representação de filas de prioridades

- Lista ordenada:
  - Inserir é O(n)
  - Retirar é O(1)
  - Ajuntar é O(n)
- Lista não ordenada:
  - Inserir é O(1)
  - Retirar é O(n)
  - Ajunta é O(1) para ligada e O(n) para arrays

Representação de filas de prioridades

- A melhor representação é através de uma estrutura de dados chamada *heap*
  - Inserir, Retirar, Substituir são O(log n)

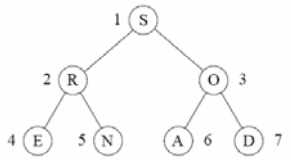
Heap (binária)

- É uma seqüência de itens com chaves c[1], c[2], . . . , c[n], tal que:

$$c[i] \geq c[2i]$$
$$c[i] \geq c[2i + 1]$$

para todo i = 1, 2, . . . , n/2.

- Definição pode ser visualizada como uma árvore binária

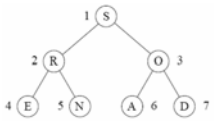


Heap

- Árvore binária completa
  - Os nós são numerados de 1 a n.
  - O primeiro nó é chamado raiz.
  - O nó $\lfloor k/2 \rfloor$ é o pai do nó k, para $1 < k \leq n$.
  - Os nós 2k e 2k + 1 são os filhos à esquerda e à direita do nó k, para $1 \leq k \leq \lfloor k/2 \rfloor$
- Propriedades
  - As chaves em cada nó s ao maiores do que as chaves em seus filhos
  - A chave no nó raiz é a maior chave do conjunto

Heap em array

- Uma árvore binária completa pode ser representada por um array
- A representação é extremamente compacta
  - Permite caminhar pelos nós da árvore facilmente
- Os filhos de um nó i estão nas posições 2i e 2i + 1
- O pai de um nó i está na posição i / 2



1	2	3	4	5	6	7
S	R	O	E	N	A	D

Classificação de heaps

- Max-heap
  - Maior elementos está acima dos outros na árvore
- Min-heap
  - Menor elemento está acima dos outros na árvore

Fila de prioridades com heaps

```
public class FPMaxHeap {
  private Comparable v[] ;
  private int n; //heap size
  public FPHeapMax (Comparable v[]){
    this.v= v;
    this.n = this.v.length - 1;
  }

  public void refaz (int i)
  public void constroi ()
  public Comparable max ()
  public Comparable retiraMax ()
  public void aumentaChave(int i,Comparable newK)
  public void insere (Comparable x)
}
```

Métodos para heap

```
public Comparable max () {
  return this.v[1] ;
}

public void refaz(int pos){
  int maior = 0;
  int esq = pos * 2;
  int dir = pos * 2 + 1;
  if (esq <= n && v[esq]>v[pos])
    maior = esq;
  else
    maior = pos;
  if (dir <= n && v[dir]>v[pos])
    maior = dir;
  if (maior != pos){
    exchange(v[pos],v[maior]);
    refaz(maior);
  }
}
```

Métodos para heap

```
//metodo para construir a heap a partir de um
//array preenchido
public void constroi(){
  for (int i= n/2; i<0; i++)
    this.refaz(i);
}
```

Depois de n/2, todos sao folhas (nao precisa refaz())

Métodos para heap

```
//metodo para retirar maior chave
public Comparable retiraMax(){
  Comparable maximo;
 if(this.n<1) //"Erro : heap vazio"
 else {
   maximo= this.v[1];
   this.v[1]= this.v[this.n--];
   refaz(1);
 }
 return maximo;
}
```

Coloca ultimo elemento na primeira e refaz tudo

Métodos para heap

```
//metodo para aumentar uma chave
public void aumentaChave(int i,Comparable newK){
  this.v[i] = newK;
  while((i>1)&&(newK.compareTo(this.v[i/2])>=0)){
    exchange(this.v[i],this.v[i/2]);
    i/= 2;
  }
}
```

Se a chave nova for maior que a do pai, sobe na heap

Algoritmo heapsort

1. Construir o *heap*
2. Troque o item na posição 1 do vetor (raiz do *heap*) com o item da posição n
3. Use refaz para reconstituir o *heap* para os itens v[1], v[2], . . . , v[n – 1]
4. Repita os passos 2 e 3 com os n – 1 itens restantes, depois com os n – 2, até que reste apenas um item

Heapsort

- O caminho seguido pelo refaz para reconstituir a condição do *heap* está em negrito
- Por exemplo, após a troca dos itens S e D na segunda linha, o item D volta para a posição 5, após passar pelas posições 1 e 2

1	2	3	4	5	6	7
S	R	O	E	N	A	D
R	**N**	O	E	**D**	A	**S**
O	N	**A**	E	D	R	
N	**E**	A	**D**	O		
E	**D**	A	N			
D	**A**	E				
A	D					

Heapsort

```
public static void heapsort(Comparable v[],int n){
  FPMaxHeap fpHeap= new FPMaxHeap(v);
  fpHeap.constroi();

  while (dir>1){
    Comparable x = v[1];
    exchange(v[1],v[n]);
    n--;
    fpHeap.refaz(1);
  }
}
```

Ordenação do vetor usando a heap – na verdade ordenação in-place (no próprio vetor)

Análise do heapsort

- O refaz gasta cerca de log(n) operações, no pior caso
- Logo, heapsort gasta um tempo de execução proporcional a O(nlog(n))
- Vantagens
  - O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
- Desvantagens
  - O anel interno do algoritmo é bastante complexo se comparado com dos outros algoritmos
- Recomendado
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap*