Filas de prioridade

Algoritmos e Estruturas de Dados

2019/2020

FEUP

Filas de prioridade

- Uma fila de prioridade permite, pelo menos, duas operações sobre um conjunto de valores comparáveis:
  - *inserção* de um elemento
  - *remoção do menor/maior* elemento
- Operações adicionais facultativas:
  - diminuir, de um determinado valor, o elemento que se encontra numa determinada posição
  - aumentar, de um determinado valor, o elemento que se encontra numa determinada posição
  - remover o elemento de uma determinada posição
- Implementação:
  - Listas ligadas
  - Árvores binárias de pesquisa
  - *Heaps* binários

Filas de prioridade

Implementação por listas ligadas

- Características de uma implementação por listas ligadas:
  - inserção no início da lista: complexidade temporal *O(1)*
  - pesquisa, para obtenção do menor elemento: *O(N)*
- Alternativa: manter a lista ordenada
  - inserção : *O(N)*
  - obtenção do menor elemento : *O(1)*

Qual a melhor alternativa?

Filas de prioridade

Implementação baseada em árvores binárias de pesquisa

- Características de uma implementação baseada em árvores de pesquisa:
  - inserção : *O(log N)*
  - obtenção do menor elemento : *O(log N)* em média, e *O(N)* no pior caso

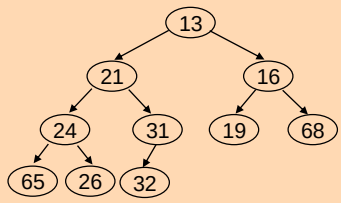
Ambas as operações podem ser realizadas em *O(log N)* no pior caso, usando árvores equilibradas.

Árvores binárias completas: todos os níveis estão completamente preenchidos, com possível exceção do último que estará preenchido a partir da esquerda.

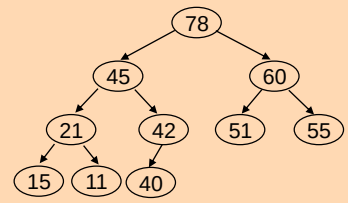
Uma árvore binária completa pode ser representada num vetor (e assim não é necessário usar apontadores)

Heaps binários

- **Propriedades do *heap* binário *(min-heap)***
  - É uma árvore binária completa
  - Para todos os nós, com exceção da raiz, o valor do pai é menor ou igual ao valor do nó.



min-heap



max-heap

- Vantagem em relação às árvores binárias:
  - acesso ao valor mínimo em tempo constante, *O(1)*: mínimo está sempre na raiz

Filas de prioridade: implementação

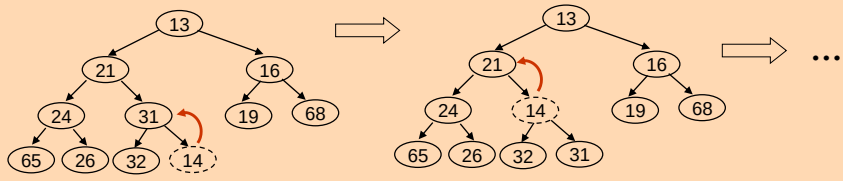
- Declaração da classe ***BinaryHeap***
  (uma implementação)

```
template <class Comparable>
class BinaryHeap {
public:
   explicit BinaryHeap(int capacity = 100);
   bool isEmpty() const;
   bool is Full() const;
   const Comparable & findMin() const;
   void insert(const Comparable & x);
   void deleteMin();
   void deleteMin(Comparable & minItem);
   void makeEmpty();
private:
   int currentSize;
   vector<Comparable> array;   // vector começa na posição 1
   void buildHeap();
   void percolateDown(int hole);
};
```

Filas de prioridade: implementação

- Heap – inserção de um elemento

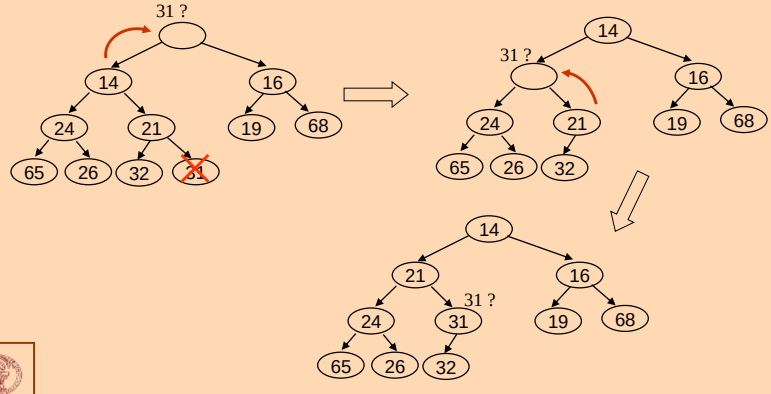
1. Inserir elemento na primeira posição livre
2. Enquanto não for respeitada a restrição de ordem: trocar elemento e seu pai



Filas de prioridade: implementação

- Heap – remoção do mínimo

1. Remover elemento da primeira posição (raiz), é o mínimo
2. Colocar na raiz o último elemento
3. Enquanto não for respeitada a restrição de ordem: trocar elemento e menor dos seus filhos



Filas de prioridade: implementação

- classe ***BinaryHeap*** : inserção de elementos , remoção do mínimo

```
template <class Comparable>
void BinaryHeap<Comparable>::insert(const Comparable & x)
{
   if ( isFull() ) throw Overflow();
   int hole = ++currentSize;
   for ( ; hole > 1 && x < array[hole/2]; hole/=2 )
      array[hole] = array[hole/2];
   array[hole] = x;
}
```

```
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin(Comparable & minItem)
{
   if ( isEmpty() ) throw Underflow();
   minItem = array[1];
   array[1] = array[currentSize--];
   percolateDown(1);
}
```


Filas de prioridade: implementação

- classe ***BinaryHeap***

```
template <class Comparable>
void BinaryHeap<Comparable>::percolateDown(int hole)
{
   int child;
   Comparable tmp = array[hole];
   for ( ; hole*2 <= currentSize; hole = child) {
      child = hole*2;
      if ( child != currentSize && array[child+1] < array[child] )
            child ++;
      if ( array[child] < tmp )
            array[hole] = array[child];
      else break;
   }
   array[hole] = tmp;
}
```

Filas de prioridade: implementação

- classe ***BinaryHeap***: construção
  - Uma inserção tem complexidade *O(log N)* no pior dos casos, mas apenas *O(1)* em média
  - Uma sequência de *N* inserções (sem remoções) permite construir um heap em *O(N log N)*, no pior dos casos
  - É possível construir um heap a partir de um vetor desordenado em tempo ***O(N)*** no pior dos casos (***verificar…***)

```
template < class Comparable>
void BinaryHeap<Comparable>::buildHeap()
{
   for ( int i = currentSize/2; i > 0; i-- )
      percolateDown(i);
}
```


Filas de prioridade

- **Ordenação** com *heaps*

  1. Criar um heap binário a partir de um vetor : *O(N)*
  2. Executar *N* operações *deleteMin()*, e guardar os elementos sucessivamente em um outro vetor. Os elementos são retirados por ordem crescente. Cada operação tem complexidade *O(log N)*

  O tempo total é portanto *O(N log N)*

  Problema (desvantagem): Necessidade de usar outro vetor.

  Solução: Usar o mesmo vetor. Quando se retira um elemento, o heap também liberta uma posição; essa posição pode ser usada para guardar o elemento retirado. O vetor fica ordenado por ordem decrescente.

Heapsort: implementação

- Heapsort : ordenação de vectores

```
template <class Comparable>
void heapsort(vector<Comparable> & a)
{
   for ( int i = a.size()/2; i >= 0; i--)
      percDown(a, i, a.size());
   for ( int j = a.size() - 1; j > 0; j--)
   {
      Comparable t = a[0];
      a[0] = a[j]; a[j] = t;
      percDown(a, 0, j);
   }
}
```


Heapsort: implementação

- Heapsort : ordenação de vectores

```
template <class Comparable>
void percDown(vector<Comparable> & a, int i, int n)
{
   int child;
   Comparable tmp;
   for ( tmp = a[i]; (2*i + 1) < n; i = child ) {
       child = 2 * i + 1;
       if ( child != n-1 && a[child] < a[child+1] )
           child ++;
       if ( tmp < a[child] )
           a[i] = a[child];
       else
           break;
      }
      a[i] = tmp;
   }
```

Filas de prioridade (Standard Template Library - STL)

- class ***priority_queue*** *(max-heap)*

- Alguns métodos:
  - bool empty() const
  - int size() const
  - const T & top() const
  - void push(const T &)
  - void pop()

Filas de prioridade : aplicação

- Alocação de recursos
  - Implementar um programa que distribui um conjunto de tarefas por diversas máquinas (todas iguais), de modo a minimizar o tempo que demora a executar todas as tarefas.

  - Estratégia LPT *("longest processing time first")*
    - As tarefas são alocadas às máquinas por ordem decrescente do seu tempo de processamento
    - As tarefas vão sendo alocadas às máquinas à medida que estas últimas ficam livres
    - Para determinar a primeira máquina livre, usa-se uma fila de prioridade, ordenada segundo o instante em que as máquinas ficam livres
    - A cada máquina retirada da fila é alocada a tarefa seguinte, e calculado o instante em que a máquina estará de novo livre. A máquina é então inserida de novo na fila de prioridade

Filas de prioridade: aplicação (usa a classe priority_queue – STL)

```
struct Maquina {
   int ID, disp;
   bool operator < (const Maquina & m) const
      { return disp > m.disp; }
   };

struct Tarefa {
   int ID, duracao;
   bool operator < (const Tarefa & t) const
      { return duracao < t.duracao; }
};
```

```
int main() {
   vector<Tarefa> tarefas;
   le_tarefas(tarefas);
   int nmaq;
   cout << "Numero de maquinas=: "; cin >> nmaq;
   LPT(tarefas, nmaq);
   return 1;
}
```


Filas de prioridade: aplicação (usa a classe priority_queue – STL)

```
template <class T> void LPT(vector<T> & a, int nm)
{
  heapsort(a);
  priority_queue<Maquina> h;
  Maquina m1;
  for ( int i = 1; i <= nm; i++ ) {
     m1.disp = 0; m1.ID = i;
     h.push(m1);
  }
  for ( int i = 0; i < a.size() ; i++ ) {
     m1 = h.top();
     h.pop();
     cout  << "Tarefa " << a[i].ID << " (dur= " << a[i].duracao
           << ") na máquina " << m1.ID  << " de " << m1.disp
           << " até " << (m1.disp+a[i].duracao) << endl;
     m1.disp += a[i].duracao;
     h.push(m1);
  }
}
```

vetor ordenado de tarefas (duração decrescente)

fila de prioridade de máquinas