ACH2002

Aula 7

Técnicas de Desenvolvimento de Algoritmos -

Divisão e Conquista

(adaptados dos slides de aula da Profa. Fátima L. S. Nunes)

EACH USP

Aula passada

- Conceitos de **recursividade**
- Quando usar e não usar recursividade
- Como provar corretude e analisar complexidade de algoritmos recursivos (utilizando **equações de recorrência** e **indução matemática**)

Indução Matemática

- Algoritmos recursivos podem ser definidos e estudados a partir da indução matemática.

- Seja *T* um teorema a ser provado

  - Consideremos ***T*** como possuindo um número natural como parâmetro (***n***)

  - Em vez de tentar provar que *T* é válido para todos valores de ***n***, basta provar duas condições:

1. *T* é válido para *n = 1 (ou outro número inicial);* base da indução

2. Para todo *n* > 1:

   - se *T* é válido para um dado *k* ⇒ *T* é válido para *k+1*

     passo da indução

Recursividade e indução

- Definindo o cálculo do fatorial usando indução:

```
int fatorial (int n)
 {
   if (n == 0)
     return 1;
   else
     return n*fatorial (n-1);
 }
```

  - Base da indução

    Se n=0, então o fatorial = 1

  - Passo indutivo:

    - Devemos expressar a solução para n > 0, supondo que já sabemos a solução para algum caso mais simples (**n-1**, por exemplo)

    n! = n * (n-1)!

Aula de hoje

- **Divisão e conquista** (um tipo de técnica muito comum em recursividade)
- EP 1

Novo algoritmo de ordenação: **MergeSort**

- O que é *merge*?

Novo algoritmo de ordenação: **MergeSort**

- O que é *merge*?
  - Combinar ou unir em uma única unidade

Novo algoritmo de ordenação: **MergeSort**

- O que é *merge*?
  - Combinar ou unir em uma única unidade

- Como poderia ser um algoritmo de ordenação baseado nessa ideia?

Novo algoritmo de ordenação: **MergeSort**

- O que é *merge*?
    - Combinar ou unir em uma única unidade

- Como poderia ser um algoritmo de ordenação baseado nessa ideia?
    - Divido os elementos em duas partes, ordeno cada uma das partes separadamente, e depois combino as partes ordenadas

https://youtu.be/3l2HvDlPF9U

Novo algoritmo de ordenação: **MergeSort**

- Ordenação por **intercalação** (que é a forma de combinar):
  - Dados um natural *n* e uma sequência de *n* elementos:
    - Divide o arranjo em duas subsequências de *n/2* elementos;
    - Ordena as duas subsequências da mesma forma, utilizando a própria ordenação por intercalação (ou seja, ?);
    - Combina os resultados fazendo a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

Novo algoritmo de ordenação: **MergeSort**

- Ordenação por **intercalação** (que é a forma de combinar):
  - Dados um natural *n* e uma sequência de *n* elementos:
    - Divide o arranjo em duas subsequências de *n/2* elementos;
    - Ordena as duas subsequências da mesma forma, utilizando a própria ordenação por intercalação (ou seja, recursivamente);
    - Combina os resultados fazendo a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

Novo algoritmo de ordenação: **MergeSort**

- Ordenação por **intercalação** (que é a forma de combinar):
  - Dados um natural *n* e uma sequência de *n* elementos:
    - Divide o arranjo em duas subsequências de *n/2* elementos;
    - Ordena as duas subsequências da mesma forma, utilizando a própria ordenação por intercalação (ou seja, recursivamente);
    - Combina os resultados fazendo a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

Como seria?

Arranjo inicial

5	**2**	**4**	**7**	**1**	**3**	**2**

MergeSort

- Quais os parâmetros? O que eu deveria fazer? Qual a base da recursão?

```
mergeSort (?)
```

MergeSort

- Quais os parâmetros? O que eu deveria fazer? Qual a base da recursão?
  - Base da recursão: quando a sequência a ser ordenada tem comprimento 0 ou 1: neste caso não há trabalho a ser feito.

```
mergeSort (A,i,f)
se i < f
    m ← ⌊(i+f)/2⌋
    mergeSort(A, i, m)
    mergeSort(A, m+1,f)
    merge(?)
```

MergeSort

- Dados A, i, f:
  - Preciso ordenar elementos do subarranjo A [i..f];
  - se f≥i, o subarranjo tem no máximo um elemento (já está ordenado);
  - caso contrário, faz a divisão: calcula um índice m que particiona A [i..f] em dois subarranjos: A[i..m] contendo n/2 elementos e A[m+1..f] contendo n/2 elementos (aproximadamente...).
  - Depois intercala os dois subarranjos. Quais? Quais parâmetros então devem ser passados à rotina merge?

MergeSort

- Dados A, i, f:
  - Preciso ordenar elementos do subarranjo A [i..f];
  - se f≥i, o subarranjo tem no máximo um elemento (já está ordenado);
  - caso contrário, faz a divisão: calcula um índice m que particiona A [i..f] em dois subarranjos: A[i..m] contendo n/2 elementos e A[m+1..f] contendo n/2 elementos (aproximadamente...).
  - Depois intercala os dois subarranjos. Quais? Quais parâmetros então devem ser passados à rotina merge?

A, i, m, f

MergeSort – **A intercalação...**

- Operação chave do algoritmo MergeSort:
    - intercalação de duas sequências **ordenadas**
- Algoritmo (analogia com baralho):
    1. há duas pilhas com cartas ordenadas;
    2. menor carta está com a face para cima em cada pilha;
    3. pegar menor delas e coloca em nova pilha, com face para baixo;
    4. repete 3 até terminar uma das pilhas;
    5. junta cartas da pilha restante.

MergeSort – **A intercalação**

- Algoritmo da parte de intercalação (ainda não é o alg. todo):

```
merge (A,i,m,f) // A=arranjo; i,m,f=índices, i ≤ m < f
   // A[i..m] e A [m+1..f] estão ordenados
   // intercala os subarranjos para formar novo arranjo A
```

?

MergeSort – **A intercalação**

- Algoritmo da parte de intercalação (ainda não é o alg. todo):

```
merge (A,i,m,f) // A=arranjo; i,m,f=índices, i ≤ m < f
   // A[i..m] e A [m+1..f] estão ordenados
   // intercala os subarranjos para formar novo arranjo A

   // define tamanhos dos subarranjos
   n1 ← m–i+1
   n2 ← f–m

   // preciso criar uma cópia dessas duas partes (subarranjos)
   // com sentinela (para evitar testar se chegou fim)
   criar arranjos L[1..n1+1] e R[1..n2+1]
   L[n1+1] ← ∞
   R[n2+1] ← ∞

?

   // continua...
```

Lembrando que em pseudocódigos vetores começam em 1

MergeSort – **A intercalação**

- Algoritmo da parte de intercalação (ainda não é o alg. todo):

```
merge (A,i,m,f) // A=arranjo; i,m,f=índices, i ≤ m < f
   // A[i..m] e A [m+1..f] estão ordenados
   // intercala os subarranjos para formar novo arranjo A

   // define subarranjos
   n1 ← m–i+1
   n2 ← f–m

   // preciso criar uma cópia dessas duas partes (subarranjos)
   // com sentinela (para evitar testar se chegou fim)
   criar arranjos L[1..n1+1] e R[1..n2+1]
   L[n1+1] ← ∞
   R[n2+1] ← ∞

   para j ← 1 até n1
       L[j] ← A[i+j-1]
   para j ← 1 até n2
       faça R[j] ← A[m+j]
       // continua...
```

MergeSort – **A intercalação**

- Algoritmo da parte de intercalação (ainda não é o alg. todo):

```
merge (A,i,m,f) // A=arranjo; i,m,f=índices, i ≤ m < f
   // ... continuação

   //  mesclar subarranjos
```

?

MergeSort – **A intercalação**

- Algoritmo da parte de intercalação (ainda não é o alg. todo):

```
merge (A,i,m,f) // A=arranjo; i,m,f=índices, i ≤ m < f
   // ... continuação

   //  mesclar subarranjos
   kL ← 1 // kL é o índice que percorre L
   kR ← 1 // kR é o índice que percorre R
   para k ← i até f   // k é o índice que percorre A
       se L[kL] ≤ R[kR]
            A[k] ← L[kL]
            kL ← kL + 1
       senão
            A[k] ← R[kR]
            kR ← kR + 1
       fim se
   fim para
```

MergeSort

```
mergeSort (A,i,f)
se i < f
      m ← ⌊(i+f)/2⌋
      mergeSort(A, i, m)
      mergeSort(A, m+1,f)
      merge(A,i,m,f)
```

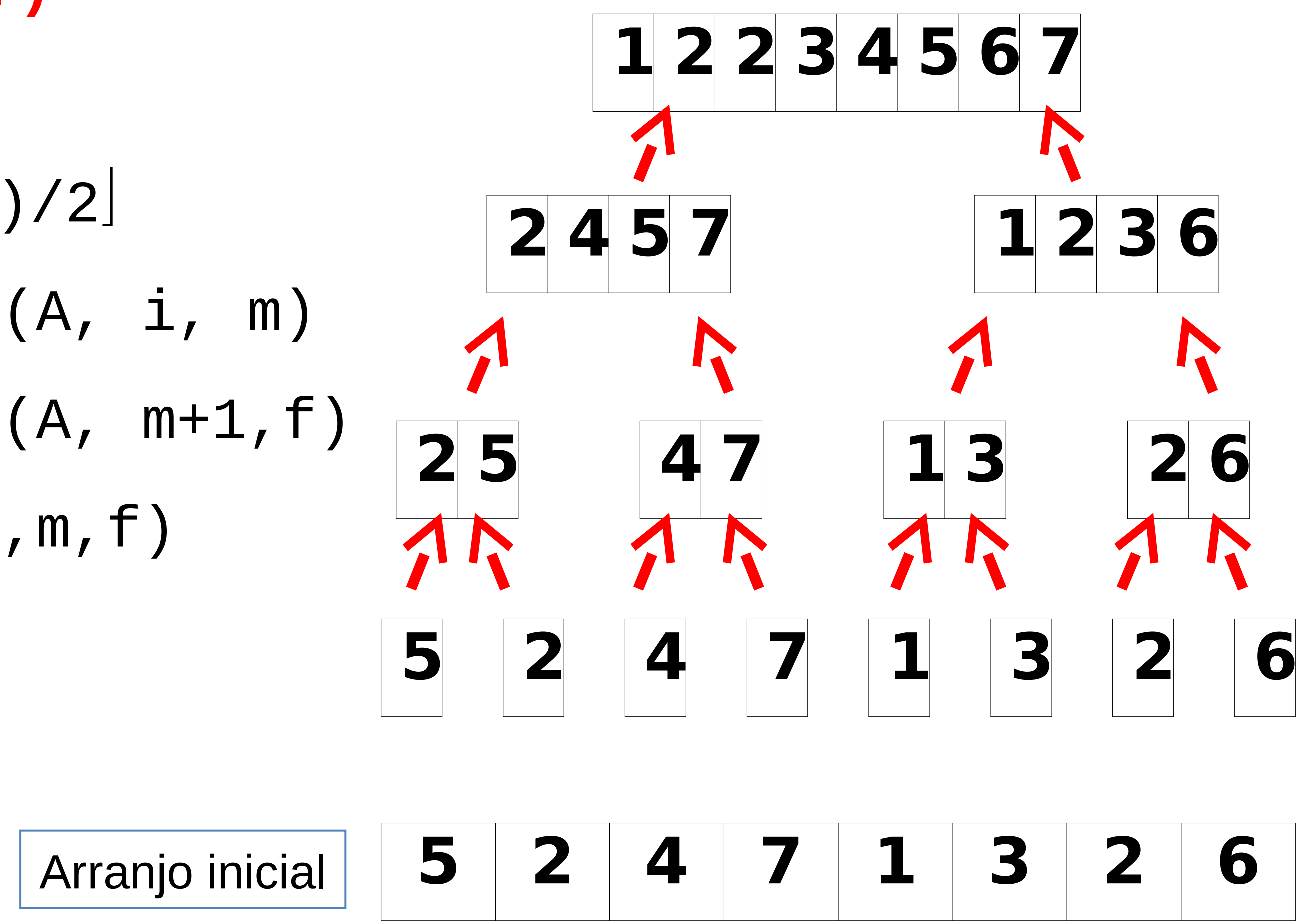
Arranjo inicial

5	**2**	**4**	**7**	**1**	**3**	**2**	**6**

Divide até obter subarranjos com tamanho 1.
Então, começa a mesclar...

MergeSort

```
mergeSort (A,i,f)
se i < f
        m ← ⌊(i+f)/2⌋
        mergeSort(A, i, m)
        mergeSort(A, m+1,f)
        merge(A,i,m,f)
```



Divide até obter subarranjos com tamanho 1.
Então, começa a mesclar...

Fazer uma simulacao como a feita para fatorial

MergeSort

mergeSort (A,i,f)

```
se i < f
        m ← ⌊(i+f)/2⌋
        mergeSort(A, i, m)
        mergeSort(A, m+1,f)
        merge(A,i,m,f)
```

fatorial (n)

```
se n < 2
   retorna 1
senão
   retorna n * fatorial (n-1)
 fim se
```

Quais as semelhanças e diferenças entre esse algoritmo do mergesort e o algoritmo de cálculo do fatorial?

MergeSort

```
mergeSort (A,i,f)
se i < f
      m ← ⌊(i+f)/2⌋
      mergeSort(A, i, m)
      mergeSort(A, m+1,f)
      merge(A,i,m,f)
```

```
fatorial (n)
   se n < 2
      retorna 1
   senão
      retorna n * fatorial (n-1)
    fim se
```

Quais as semelhanças e diferenças entre esse algoritmo do mergesort e o algoritmo de cálculo do fatorial?

Semelhanças:

- Usam recursão
- Dividem o problema em um (ou mais) problema(s) menor(es) e usa a(s) solução(ões) para resolver o problema original

Diferenças:

- O número de problemas menores para os quais o problema original foi dividido

MergeSort

mergeSort (A,i,f)

```
se i < f
        m ← ⌊(i+f)/2⌋
        mergeSort(A, i, m)
        mergeSort(A, m+1,f)
        merge(A,i,m,f)
```

insertionSort (A)

```
para j = 2 até tamanho[A] faça
  chave = A[j]
// ordenando elementos à esquerda
  i = j - 1
  enquanto i > 0 e A[i] > chave faça
     A[i+1] = A[i]
     i = i -1
   fim enquanto
   A[i+1] = chave
fim para
```

Contraste com o insertionSort que é **incremental**

MergeSort

- Ordenação por **intercalação** (que é a forma de combinar):
  - Dados um natural *n* e uma sequência de *n* elementos:
    - Dividir: divide o arranjo em duas subsequências de *n/2* elementos;
    - Conquistar: classifica as duas subsequências recursivamente, utilizando a própria ordenação por intercalação;
    - Combinar: faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

Paradigma Dividir e conquistar

- Em geral, algoritmos recursivos seguem a abordagem *dividir e conquistar:*
  - desmembram o problema em vários subproblemas semelhantes, mas menores em tamanho;
  - resolvem os subproblemas (recursivamente);
  - combinam soluções dos subproblemas para criar solução para o problema original.

Paradigma Dividir e conquistar

- Três passos em cada nível de recursão:
  - Dividir – o problema em um determinado número de subproblemas.
  - Conquistar – os subproblemas, resolvendo-os recursivamente. Se os tamanhos dos subproblemas forem pequenos o suficiente, resolvê-los diretamente.
  - Combinar – as soluções dadas aos subproblemas para formar solução procurada para problema original.

Exercício para casa: Como seria a versão recursiva do algoritmo de busca binária?

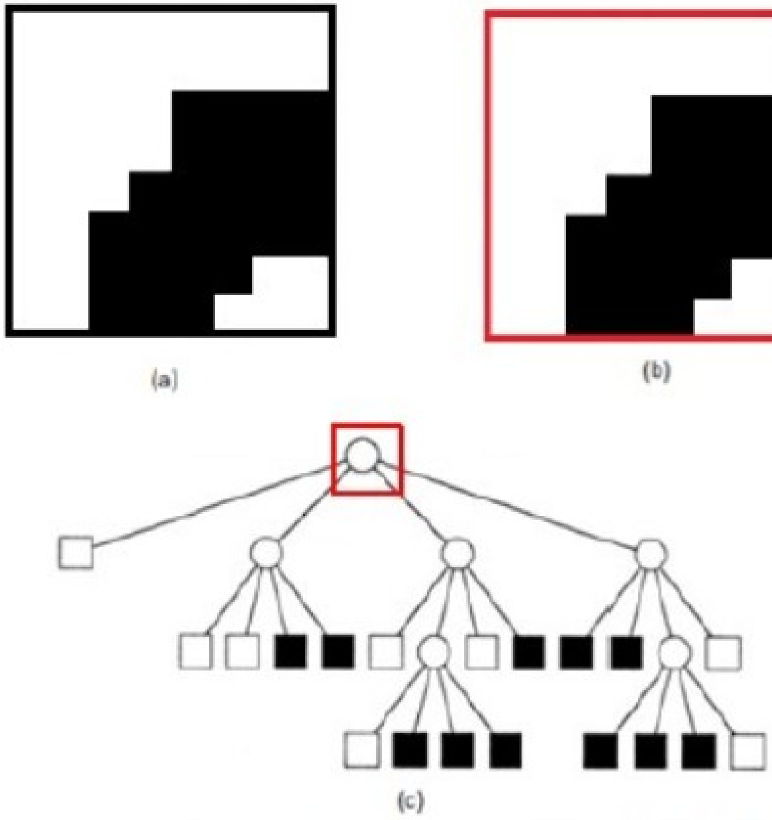
A solução é também do tipo "dividir e conquistar"

Exercício Programa 1

- Uso de recursão, e divisão e conquista

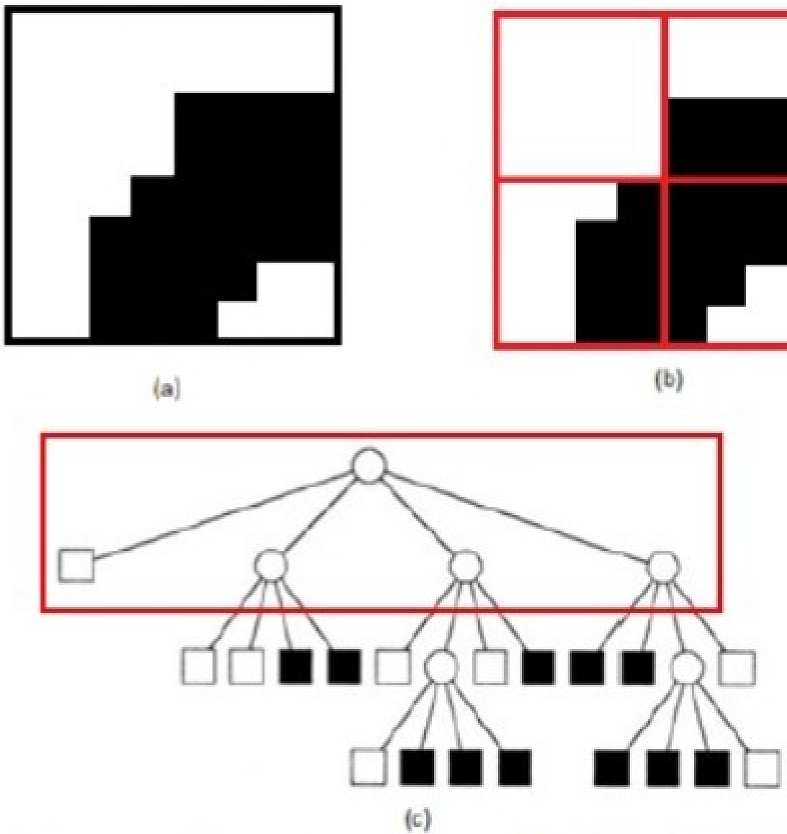
Imagens: representação por quadtrees

Descendo no nível de pixels



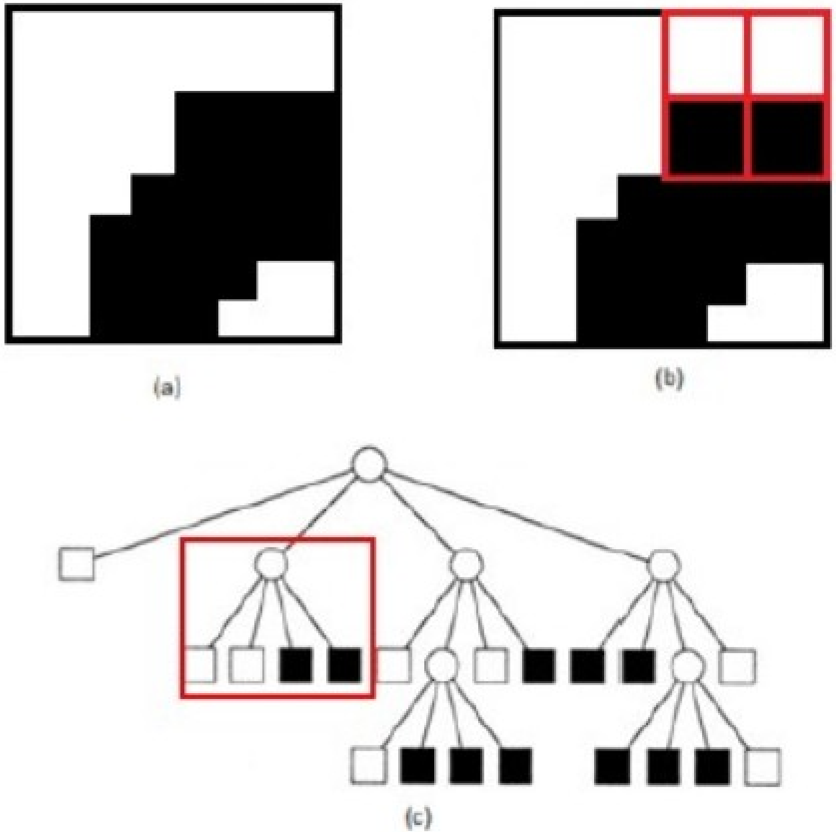
Estrutura de uma quadtree. Baseado em (AIZAWA; NAKAMURA, 1999)

Imagens: representação por quadtrees



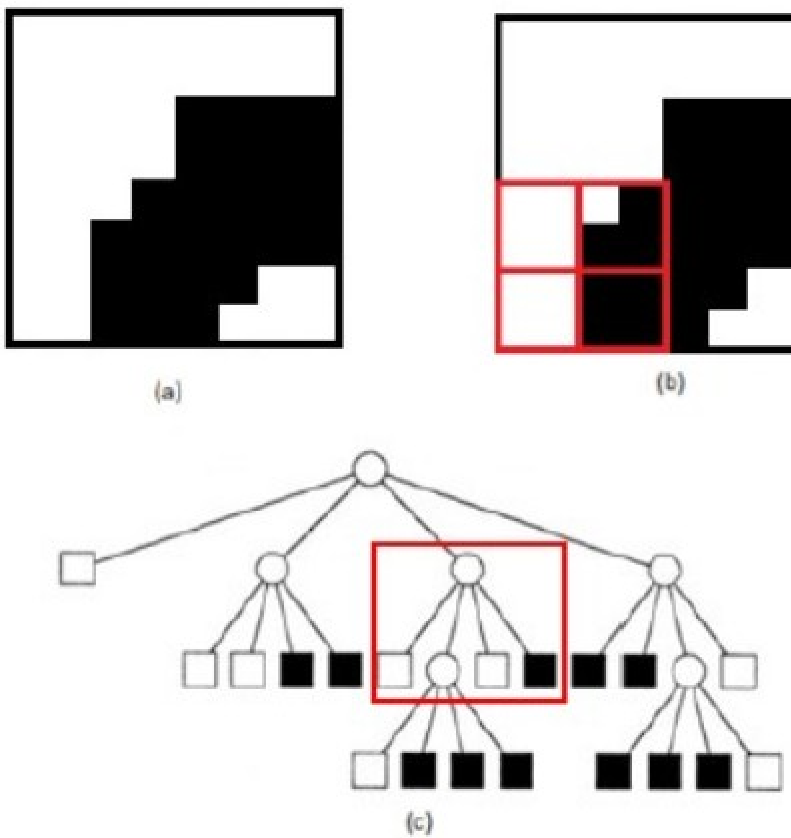
Estrutura de uma quadtree. Baseado em (AIZAWA; NAKAMURA, 1999)

Imagens: representação por quadtrees



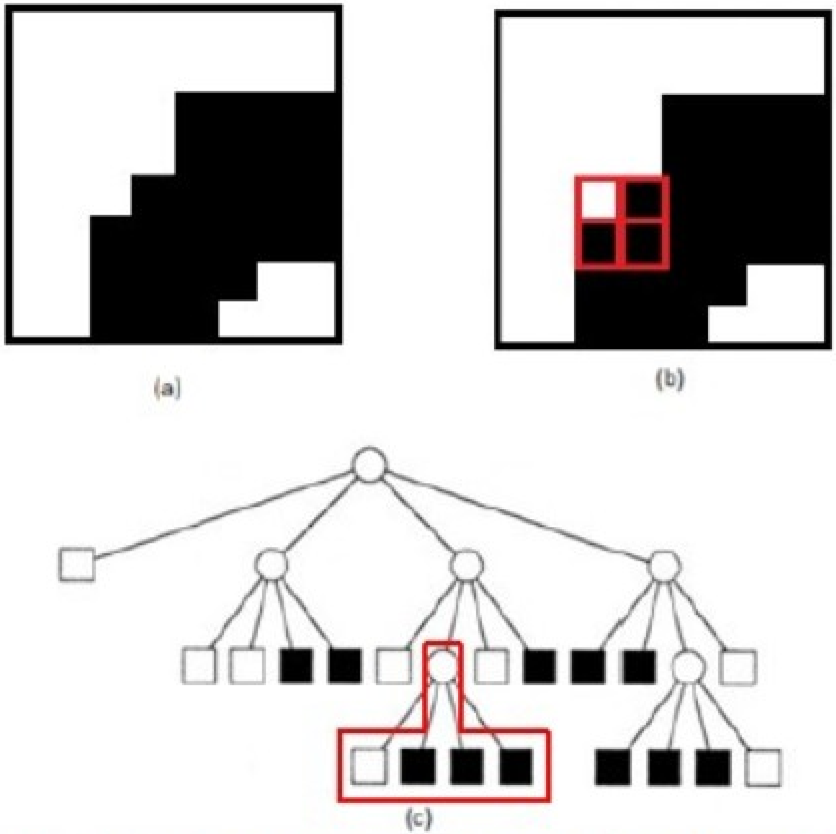
Estrutura de uma quadtree. Baseado em (AIZAWA; NAKAMURA, 1999)

Imagens: representação por quadtrees



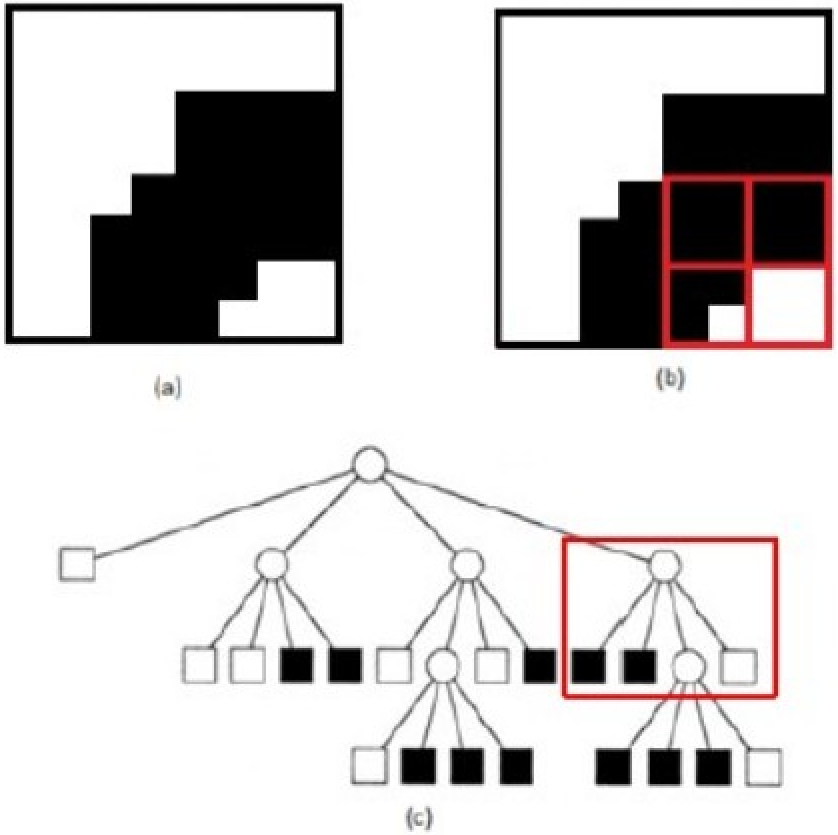
Estrutura de uma quadtree. Baseado em (AIZAWA; NAKAMURA, 1999)

Imagens: representação por quadtrees



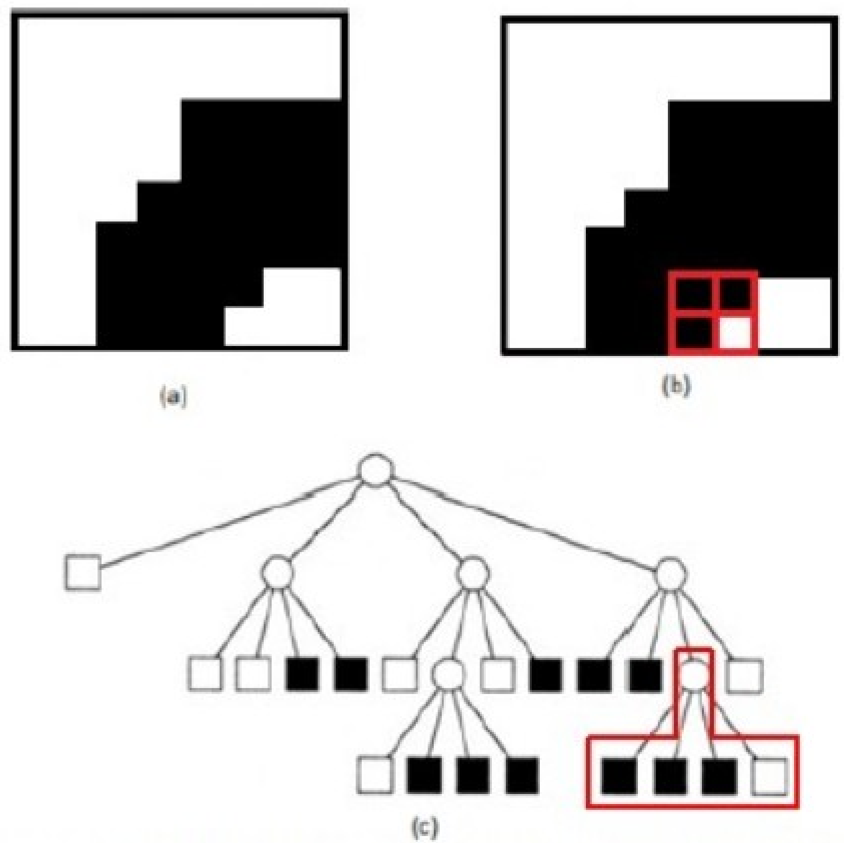
Estrutura de uma quadtree. Baseado em (AIZAWA; NAKAMURA, 1999)

Imagens: representação por quadtrees



Estrutura de uma quadtree. Baseado em (AIZAWA; NAKAMURA, 1999)

Imagens: representação por quadtrees



Estrutura de uma quadtree. Baseado em (AIZAWA; NAKAMURA, 1999)

Referências

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - Tradução da 2a. Edição Americana. Editora Campus, 2002 (Cap 2.3)
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 2a. Edição, 2004. Cap 2