📜 História e Motivação

O conceito de quadtree foi introduzido por Raphael Finkel e J.L. Bentley em 1974 no artigo *"Quad Trees: A Data Structure for Retrieval on Composite Keys"* publicado na revista *Acta Informatica*.

A motivação principal era criar uma estrutura de dados eficiente para armazenar e recuperar informações espaciais em duas dimensões, como imagens e mapas, de forma hierárquica e adaptativa.

🔍 O Que Era Feito Antes

Antes das quadtrees, estruturas como matrizes bidimensionais eram utilizadas para representar dados espaciais. No entanto, essas abordagens apresentavam limitações significativas:

- **Ineficientes para dados esparsos**: Matrizes ocupam espaço uniforme, mesmo em regiões sem dados relevantes.
- **Operações de busca lentas**: A busca por elementos específicos exigia varreduras completas ou algoritmos complexos.
- **Pouca adaptabilidade**: Não se ajustavam bem a diferentes densidades de dados em regiões distintas.

🧠 Como Funciona a Quadtree

Uma quadtree é uma estrutura de árvore onde cada nó interno possui exatamente quatro filhos, correspondendo às subdivisões de uma região em quadrantes. O processo é recursivo:

1. **Divisão**: Se uma região contém mais de um ponto ou excede uma capacidade predefinida, ela é subdividida em quatro quadrantes.
2. **Atribuição**: Cada ponto é atribuído ao quadrante correspondente.
3. **Recursão**: O processo se repete para cada quadrante até que as condições de parada sejam satisfeitas.

Essa abordagem permite representar eficientemente áreas com diferentes densidades de dados, adaptando-se conforme necessário.

✅ Vantagens

1. **Eficiência Espacial**: Quadtrees representam dados esparsos de forma compacta, subdividindo apenas onde há necessidade.

2. **Melhoria no Desempenho de Buscas**: Operações como busca por vizinhos próximos ou interseções são otimizadas devido à estrutura hierárquica.

3. **Adaptabilidade**: A estrutura se ajusta dinamicamente à distribuição dos dados, sendo eficaz para dados com densidades variáveis.

4. **Aplicações Diversas**: Utilizadas em processamento de imagens, sistemas de informações geográficas (SIG), compressão de imagens e detecção de colisões em jogos.

❌ Desvantagens

1. **Sobrecarga de Subdivisão**: Em casos de dados altamente irregulares, pode haver subdivisões excessivas, aumentando o uso de memória.
2. **Complexidade na Deleção**: Remover elementos pode exigir a fusão de quadrantes, o que é complexo e pode desbalancear a árvore.
3. **Sensibilidade a Transformações**: Rotacionar ou transladar dados pode alterar significativamente a estrutura da quadtree, dificultando comparações.
4. **Desequilíbrio da Árvore**: Dados concentrados em regiões específicas podem gerar árvores desbalanceadas, afetando o desempenho.

📜 História e Motivação

O conceito de quadtree foi introduzido por Raphael Finkel e J.L. Bentley em 1974 no artigo *"Quad Trees: A Data Structure for Retrieval on Composite Keys"* publicado na revista *Acta Informatica*.

A motivação principal era criar uma estrutura de dados eficiente para armazenar e recuperar informações espaciais em duas dimensões, como imagens e mapas, de forma hierárquica e adaptativa.

🔍 O Que Era Feito Antes

Antes das quadtrees, estruturas como matrizes bidimensionais eram utilizadas para representar dados espaciais. No entanto, essas abordagens apresentavam limitações significativas:

- **Ineficientes para dados esparsos**: Matrizes ocupam espaço uniforme, mesmo em regiões sem dados relevantes.
- **Operações de busca lentas**: A busca por elementos específicos exigia varreduras completas ou algoritmos complexos.
- **Pouca adaptabilidade**: Não se ajustavam bem a diferentes densidades de dados em regiões distintas.

🧠 Como Funciona a Quadtree

Uma quadtree é uma estrutura de árvore onde cada nó interno possui exatamente quatro filhos, correspondendo às subdivisões de uma região em quadrantes. O processo é recursivo:

1. **Divisão**: Se uma região contém mais de um ponto ou excede uma capacidade predefinida, ela é subdividida em quatro quadrantes.
2. **Atribuição**: Cada ponto é atribuído ao quadrante correspondente.

3. **Recursão**: O processo se repete para cada quadrante até que as condições de parada sejam satisfeitas.

Essa abordagem permite representar eficientemente áreas com diferentes densidades de dados, adaptando-se conforme necessário.

✅ Vantagens

1. **Eficiência Espacial**: Quadtrees representam dados esparsos de forma compacta, subdividindo apenas onde há necessidade.
2. **Melhoria no Desempenho de Buscas**: Operações como busca por vizinhos próximos ou interseções são otimizadas devido à estrutura hierárquica.
3. **Adaptabilidade**: A estrutura se ajusta dinamicamente à distribuição dos dados, sendo eficaz para dados com densidades variáveis.
4. **Aplicações Diversas**: Utilizadas em processamento de imagens, sistemas de informações geográficas (SIG), compressão de imagens e detecção de colisões em jogos.

❌ Desvantagens

1. **Sobrecarga de Subdivisão**: Em casos de dados altamente irregulares, pode haver subdivisões excessivas, aumentando o uso de memória.
2. **Complexidade na Deleção**: Remover elementos pode exigir a fusão de quadrantes, o que é complexo e pode desbalancear a árvore.
3. **Sensibilidade a Transformações**: Rotacionar ou transladar dados pode alterar significativamente a estrutura da quadtree, dificultando comparações.
4. **Desequilíbrio da Árvore**: Dados concentrados em regiões específicas podem gerar árvores desbalanceadas, afetando o desempenho.

A seguir, apresento implementações do algoritmo **quadtree** tanto em **Python** quanto em **C**, com foco na inserção de pontos em uma estrutura de dados espacial. Essa implementação usa uma abordagem simplificada adequada para aprendizado, com suporte à inserção e visualização básica da árvore.

✅ Implementação em Python

```
class Point:
    def __init__(self, x, y, data=None):
        self.x = x
        self.y = y
        self.data = data

class Rectangle:
```

```
    def __init__(self, x, y, w, h):
        self.x = x  # centro x
        self.y = y  # centro y
        self.w = w  # largura / 2
        self.h = h  # altura / 2

    def contains(self, point):
        return (self.x - self.w <= point.x <= self.x + self.w and
                self.y - self.h <= point.y <= self.y + self.h)

class Quadtree:
    def __init__(self, boundary, capacity):
        self.boundary = boundary
        self.capacity = capacity
        self.points = []
        self.divided = False

    def subdivide(self):
        x, y, w, h = self.boundary.x, self.boundary.y, self.boundary.w,
self.boundary.h
        self.northeast = Quadtree(Rectangle(x + w/2, y - h/2, w/2, h/2),
self.capacity)
        self.northwest = Quadtree(Rectangle(x - w/2, y - h/2, w/2, h/2),
self.capacity)
        self.southeast = Quadtree(Rectangle(x + w/2, y + h/2, w/2, h/2),
self.capacity)
        self.southwest = Quadtree(Rectangle(x - w/2, y + h/2, w/2, h/2),
self.capacity)
        self.divided = True

    def insert(self, point):
        if not self.boundary.contains(point):
            return False

        if len(self.points) < self.capacity:
            self.points.append(point)
            return True
        else:
            if not self.divided:
                self.subdivide()
            return (self.northeast.insert(point) or
                    self.northwest.insert(point) or
                    self.southeast.insert(point) or
                    self.southwest.insert(point))

    def __str__(self, level=0):
        result = "  " * level + f"Quadtree ({len(self.points)}
points)\n"
        for p in self.points:
            result += "  " * (level + 1) + f"Point({p.x}, {p.y})\n"
        if self.divided:
            result += self.northwest.__str__(level + 1)
            result += self.northeast.__str__(level + 1)
```

```
            result += self.southwest.__str__(level + 1)
            result += self.southeast.__str__(level + 1)
        return result


# Exemplo de uso
boundary = Rectangle(0, 0, 10, 10)
qt = Quadtree(boundary, 2)

qt.insert(Point(1, 1))
qt.insert(Point(-2, -3))
qt.insert(Point(3, 4))
qt.insert(Point(8, -1))

print(qt)
```

✅ Implementação em C

> Essa versão usa estruturas e funções básicas para um quadtree com inserção. Não há lib gráfica nem malloc dinâmico para a lista de pontos, apenas divisão e inserção.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define CAPACITY 2

typedef struct {
    float x, y;
} Point;

typedef struct {
    float x, y, w, h;
} Rectangle;

typedef struct Quadtree {
    Rectangle boundary;
    Point points[CAPACITY];
    int count;
    bool divided;
    struct Quadtree *ne, *nw, *se, *sw;
} Quadtree;

bool contains(Rectangle rect, Point p) {
    return (p.x >= rect.x - rect.w && p.x <= rect.x + rect.w &&
            p.y >= rect.y - rect.h && p.y <= rect.y + rect.h);
}

Quadtree* create_quadtree(Rectangle boundary) {
```

```c
    Quadtree *qt = malloc(sizeof(Quadtree));
    qt->boundary = boundary;
    qt->count = 0;
    qt->divided = false;
    qt->ne = qt->nw = qt->se = qt->sw = NULL;
    return qt;
}

void subdivide(Quadtree *qt) {
    float x = qt->boundary.x;
    float y = qt->boundary.y;
    float w = qt->boundary.w / 2;
    float h = qt->boundary.h / 2;

    qt->ne = create_quadtree((Rectangle){x + w, y - h, w, h});
    qt->nw = create_quadtree((Rectangle){x - w, y - h, w, h});
    qt->se = create_quadtree((Rectangle){x + w, y + h, w, h});
    qt->sw = create_quadtree((Rectangle){x - w, y + h, w, h});
    qt->divided = true;
}

bool insert(Quadtree *qt, Point p) {
    if (!contains(qt->boundary, p)) return false;

    if (qt->count < CAPACITY) {
        qt->points[qt->count++] = p;
        return true;
    }

    if (!qt->divided) subdivide(qt);

    return insert(qt->ne, p) || insert(qt->nw, p) ||
           insert(qt->se, p) || insert(qt->sw, p);
}

void print_quadtree(Quadtree *qt, int level) {
    for (int i = 0; i < level; i++) printf("  ");
    printf("Node with %d points\n", qt->count);
    for (int i = 0; i < qt->count; i++) {
        for (int j = 0; j < level + 1; j++) printf("  ");
        printf("Point(%f, %f)\n", qt->points[i].x, qt->points[i].y);
    }
    if (qt->divided) {
        print_quadtree(qt->nw, level + 1);
        print_quadtree(qt->ne, level + 1);
        print_quadtree(qt->sw, level + 1);
        print_quadtree(qt->se, level + 1);
    }
}

int main() {
    Rectangle boundary = {0, 0, 10, 10};
    Quadtree *qt = create_quadtree(boundary);
```

```
    insert(qt, (Point){1, 1});
    insert(qt, (Point){-2, -3});
    insert(qt, (Point){3, 4});
    insert(qt, (Point){8, -1});

    print_quadtree(qt, 0);

    return 0;
}
```

✅ Código com Visualização usando `matplotlib`

```
import matplotlib.pyplot as plt

class Point:
    def __init__(self, x, y, data=None):
        self.x = x
        self.y = y
        self.data = data

class Rectangle:
    def __init__(self, x, y, w, h):
        self.x = x  # centro x
        self.y = y  # centro y
        self.w = w  # largura/2
        self.h = h  # altura/2

    def contains(self, point):
        return (self.x - self.w <= point.x <= self.x + self.w and
                self.y - self.h <= point.y <= self.y + self.h)

class Quadtree:
    def __init__(self, boundary, capacity):
        self.boundary = boundary
        self.capacity = capacity
        self.points = []
        self.divided = False

    def subdivide(self):
        x, y, w, h = self.boundary.x, self.boundary.y, self.boundary.w,
self.boundary.h
        self.northeast = Quadtree(Rectangle(x + w/2, y - h/2, w/2, h/2),
self.capacity)
        self.northwest = Quadtree(Rectangle(x - w/2, y - h/2, w/2, h/2),
self.capacity)
        self.southeast = Quadtree(Rectangle(x + w/2, y + h/2, w/2, h/2),
self.capacity)
        self.southwest = Quadtree(Rectangle(x - w/2, y + h/2, w/2, h/2),
```

```
self.capacity)
        self.divided = True

    def insert(self, point):
        if not self.boundary.contains(point):
            return False

        if len(self.points) < self.capacity:
            self.points.append(point)
            return True
        else:
            if not self.divided:
                self.subdivide()
            return (self.northeast.insert(point) or
                    self.northwest.insert(point) or
                    self.southeast.insert(point) or
                    self.southwest.insert(point))

    def draw(self, ax):
        # desenha o retângulo da região
        rect = self.boundary
        ax.plot([rect.x - rect.w, rect.x + rect.w], [rect.y - rect.h,
rect.y - rect.h], 'k-')
        ax.plot([rect.x - rect.w, rect.x + rect.w], [rect.y + rect.h,
rect.y + rect.h], 'k-')
        ax.plot([rect.x - rect.w, rect.x - rect.w], [rect.y - rect.h,
rect.y + rect.h], 'k-')
        ax.plot([rect.x + rect.w, rect.x + rect.w], [rect.y - rect.h,
rect.y + rect.h], 'k-')

        # desenha os pontos
        for p in self.points:
            ax.plot(p.x, p.y, 'ro')

        # desenha os filhos
        if self.divided:
            self.northeast.draw(ax)
            self.northwest.draw(ax)
            self.southeast.draw(ax)
            self.southwest.draw(ax)

# Exemplo de uso
boundary = Rectangle(0, 0, 10, 10)
qt = Quadtree(boundary, capacity=2)

# Inserindo pontos aleatórios
import random
for _ in range(30):
    pt = Point(random.uniform(-10, 10), random.uniform(-10, 10))
    qt.insert(pt)

# Visualizar com matplotlib
fig, ax = plt.subplots()
```

```
qt.draw(ax)
ax.set_aspect('equal')
ax.set_xlim(-11, 11)
ax.set_ylim(-11, 11)
plt.title("Visualização de Quadtree com Matplotlib")
plt.grid(True)
plt.show()
```

🧾 Conclusão

As quadtrees representam uma solução elegante e eficiente para o gerenciamento de dados espaciais bidimensionais, especialmente em contextos onde a distribuição dos dados é não uniforme. Sua capacidade de adaptação e eficiência em operações de busca as tornam valiosas em diversas aplicações, desde SIG até jogos e processamento de imagens.

No entanto, é essencial considerar suas limitações, especialmente em cenários com dados altamente irregulares ou quando são necessárias transformações espaciais frequentes. Nesses casos, outras estruturas, como k-d trees ou R-trees, podem ser mais adequadas.

Em resumo, as quadtrees são uma ferramenta poderosa no arsenal da ciência da computação, oferecendo soluções eficientes para problemas específicos de dados espaciais.
