Progetto algoritmi

Alessandro Ingrosso

Settembre 2021

1 Introduzione

I quadtree sono strutture dati ad albero che dispongono di quattro figli e la cui caratteristica è rappresentare dati in uno spazio di due dimensioni. Dividono lo spazio ricorsivamente in quattro parti e permettono di immagazzinare vari tipi di informazioni in modo efficiente essendo delle strutture arboree.

Lo spazio decomposto è una cella che contiene delle informazioni di vario tipo; se una cella contiene un numero finito di informazioni allora al suo riempimento si dividerà in altre quattro celle così da poter tenere più informazioni. I dati che contiene un nodo possono variare ed è per questo che esistono vari tipi di quadtree. Alcuni di essi lavorano con le immagini e quindi manipolano pixels, altri manipolano archi e in generale ogni variante del quadtree prende il nome dai dati con cui intende lavorare.

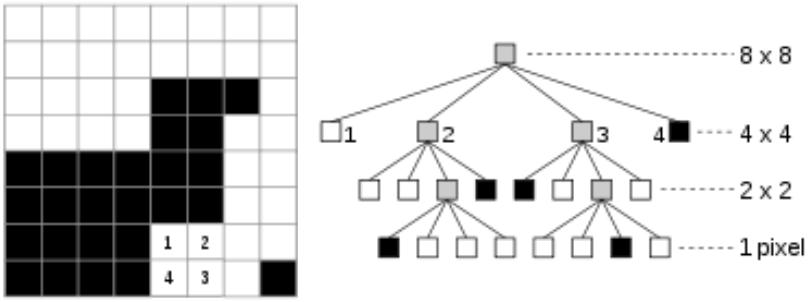

Figura 1: Quadtree per la rappresentazione delle immagini

I point quadtree sono usati per rappresentare in un piano dei punti in maniera efficiente, infatti le operazioni che vengono implementate impiegano tempo nel caso medio $\mathcal{O}(\log n)$. Essendo alberi, la struttura è fortemente ricorsiva e quindi i punti sono divisi tramite degli *split points* che vanno a formare una struttura dell'albero che può essere simile all'immagine 1. Sono di fatti superati dai k-d tree in vari campi di utilizzo perché rispetto ad essi, non scalano bene all'aumentare dei punti.

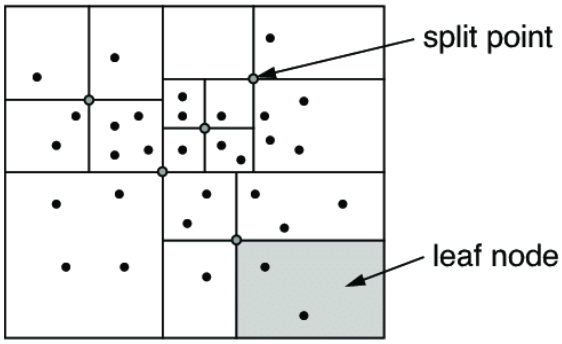

Figura 2: Point quadtree nel dettaglio

Il point quadtree dispone di diverse strutture dati al suo interno per lavorare. Tali strutture dati sono il *Point* ed il *Quadrant/Node*.

La prima dei due è la struttura dati adibita a segnare il punto con le rispettive coordinate. Implementa un metodo per confrontare se due punti sono uguali e dipendentemente dal risultato, restituisce un valore vero o falso a seconda che siano o meno gli stessi punti.

La seconda è la struttura dati adibita a tenere i vari punti e poter lavorare con essi. Al suo interno sono presenti quindi il punto ed eventuali dati aggiuntivi, come delle stringhe ad indicare il nome o dei numeri che possono indicare l'ordine di inserimento, l'importanza o un qualsiasi altro valore che abbia un'utilità nel riconoscere un punto. Può anche implementare una lista per andare a tenere i vari punti del sottopiano.

La struttura dati che utilizza le precedenti è di fatto il point quadtree che dispone al suo interno di quattro puntatori a dei figli. Essi sono chiamati in base alla locazione spaziale, quindi si hanno i point quadtree['NW'], point quadtree['NE'], point quadtree['SW'] e point quadtree['SE']. Implementa i metodi per inserire i vari punti, fare la ricerca dei nodi ed eliminarli.

2 Operazioni sulla struttura dati Point quadtree

Ci sono vari algoritmi dei point quadtree, di seguito i vari pseudo-codici e una breve descrizione del loro funzionamento.

2.1 Inserimento e suddivisione

L'algoritmo di inserimento nel point quadtree ha il compito di inserire dei punti negli appositi nodi andando a sfruttare le coordinate per gestire al meglio l'inserimento. L'inserimento può essere fatto in due modi dipendentemente se il nodo dispone di una lista o no.

Qualora il nodo non disponga di una lista l'inserimento si fa tenendo conto delle coordinate **x** e **y** e andando a inserire il punto nel nodo corretto. Quello che accade è che durante il processo si sceglie, dipendentemente dal valore delle coordinate, su quale sottoalbero andare a mettere il punto; grazie agli split points situati nei nodi dei sottoalberi, questa operazione può essere gestita in maniera molto più semplice ed efficiente perché si tratta di un banale confronto numerico(le coordinate dei punti), non diverso da quello che accade in un albero binario di interi, pertanto la complessità della procedura è la stessa di un albero binario.

Se invece si dispone di una lista, l'inserimento va fatto al suo interno e la sua dimensione va limitata ad un certo numero di elementi per evitare che la ricerca e l'inserimento ordinato dei punti impieghino tempo lineare; raggiunta la dimensione massima della lista viene scelto un punto che fa da split point per gli altri e assegna ai sottoalberi i rispettivi punti.

La scelta dello split point nel caso in cui si dispone di una lista, è fatta dall'algoritmo di **suddivisione** che prende il punto approssimativamente migliore tra quelli della lista per far si che lo spazio 2D dei figli sia il più equo possibile. In pratica si calcola un "peso" che è dato dalla somma delle differenze in valore assoluto, tra le coordinate **x** e **y** e la media dei punti **x** e **y** di tutti i punti della lista; il punto con il peso più basso è quello di split point. Si può anche eventualmente usare l'algoritmo di suddivisione randomico che sceglie a caso uno tra i punti della lista da usare come split point. La complessità del primo è pari alla lunghezza **k** della lista ($\mathcal{O}(k)$) in quanto ogni elemento della lista viene visitato un numero costante di volte mentre il secondo è costante ($\mathcal{O}(1)$).

Sono diverse le motivazioni che in pratica differenziano l'inserimento con lista da quello senza: l'algoritmo di suddivisione fornisce un altro approccio in cui viene scelto lo split point andando a vedere quale tra tutti i punti ha le coordinate più vicine alla media dei valori **x** ed **y** e andando a deallocare la lista una volta distribuiti i punti nei figli; questo è un approccio semplice per far si che lo spazio 2D sia ben bilanciato, contenga più punti in un nodo e la suddivisione del quadtree sia il più ottimale possibile.

```
1  Insert(x, y, string)
2  PointToAdd = (x, y)
3  NodeFather = Root
4  Node = Root
5  while Node ≠ NIL do
6      NodeFather = Node
7      if x ≥ Node.x
8          if y ≥ Node.y
9              Node = Node.NE
10         else
11             Node = Node.SE
12     else
13         if y ≥ Node.y
14             Node = Node.NW
15         else
16             Node = Node.SW
17 NodeInserted = PointToAdd
18 NodeInserted.key = string
19 NodeInserted.root = NodeFather
20 if x ≥ NodeFather.x
21     if y ≥ NodeFather.y
22         NodeFather.NE = NodeInserted
23     else
24         NodeFather.SE = NodeInserted
25 else
26     if y ≥ NodeFather.y
27         NodeFather.NW = NodeInserted
28     else
29         NodeFather.SW = NodeInserted
```

Algorithm 1: Algoritmo di inserimento senza lista

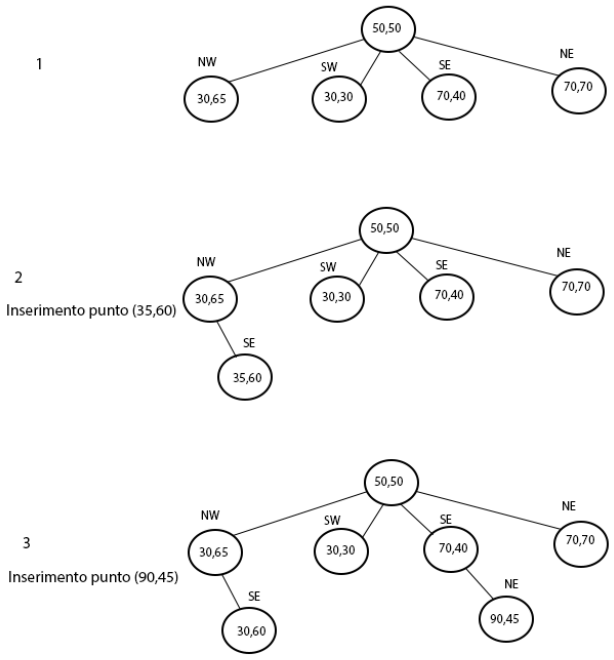

Figura 3: Esempio di inserimento senza lista

```
1  Subdivide(NodeToFix, PointToAdd)
2  for i = 0 to k do
3      mediax = NodeToFix.List.x
4      mediay = NodeToFix.List.y
5  mediax = mediax/k
6  mediay = mediay/k
7  for i = 0 to k do
8      Array[i] = (absolute(NodeToFix.List[i] − mediax)) +
         (absolute(NodeToFix.List[i] − mediay))
9  min = Array[0]
10 for i = 1 to k do
11     if min > Array[i]
12         min = Array[i]
13         index = i
14 bestPoint = NodeToFix.List[index]
15 NodeToFix.SplitPoint = bestPoint
16 NodeToFix.List[index] deleted
17 for i = 0 to k − 1 do
18     if NodeToFix.List[i].x ≥ bestPoint.x
19         if NodeToFix.List[i].y ≥ bestPoint.y
20             NodeToFix.NE.List = NodeToFix.List[i]
21         else
22             NodeToFix.SE.List = NodeToFix.List[i]
23     else
24         if NodeToFix.List[i].y ≥ bestPoint.y
25             NodeToFix.NW.List = NodeToFix.List[i]
26         else
27             NodeToFix.SW.List = NodeToFix.List[i]
28 if PointToAdd.x ≥ bestPoint.x
29     if PointToAdd.y ≥ bestPoint.y
30         NodeToFix.NE.List = PointToAdd
31     else
32         NodeToFix.SE.List = PointToAdd
33 else
34     if PointToAdd.y ≥ bestPoint.y
35         NodeToFix.NW.List = PointToAdd
36     else
37         NodeToFix.SW.List = PointToAdd
38 NodeToFix.List deallocated
```

Algorithm 2: Algoritmo di suddivisione

```
1  Insert(x, y)
2  PointToAdd = (x, y)
3  Node = Root
4  while Node.SplitPoint ≠ NIL do
5      if x ≥ Node.SplitPoint.x
6          if y ≥ Node.SplitPoint.y
7              Node = Node.NE
8          else
9              Node = Node.SE
10     else
11         if y ≥ SplitPoint.y
12             Node = Node.NW
13         else
14             Node = Node.SW
15 if Node.List.size == Node.listdim
16     Subdivide(Node, PointToAdd)
17 else
18     Node.List = PointToAdd
```

Algorithm 3: Algoritmo di inserimento con lista

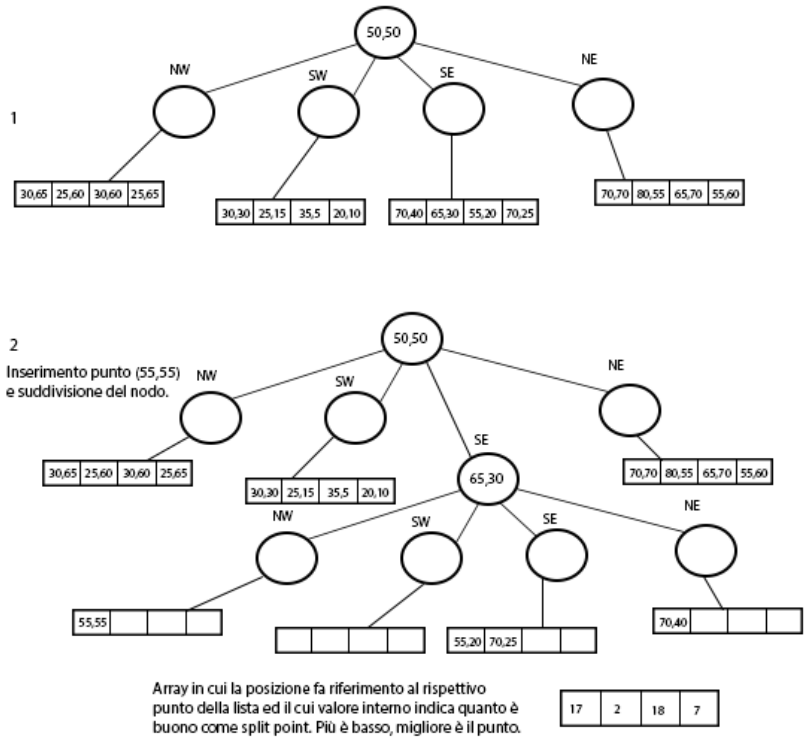

Figura 4: Esempio di inserimento con lista

2.2 Ricerca

La ricerca in un point quadtree differisce in base all'implementazione dei nodi, cioè se si usa una lista o no. Qualora non si abbia una lista per l'inserimento dei punti allora la ricerca ha complessità $\mathcal{O}(\log n)$ nel caso medio mentre è $\mathcal{O}(n)$ nel caso peggiore cioè se i punti vengono inseriti in modo tale che l'albero degeneri in una lista. Se nell'implementazione dei nodi si usa una lista per tenere i punti, allora la ricerca impiegherà nel caso medio e nel caso peggiore la stessa complessità appena vista per l'altra struttura che non usa la lista. La differenza tra i due approcci, sta nel fatto che la ricerca può essere più lenta

se la dimensione della lista è troppo elevata e quindi annulla i benefici ricavati dall'usare una struttura arborea. Dall'altra parte, come visto nell'inserimento, la procedura di suddivisione permette per un numero alto di punti, di gestire al meglio la suddivisione dello spazio evitando una degenerazione della struttura complessiva a casi lineari di ricerca.

Quello che accade in pratica nella ricerca in un quadtree, in assenza della lista di punti nel nodo, non è diverso dalla ricerca in un albero binario. Quindi, si passa da sottoalbero in sottoalbero dipendentemente dal valore del punto passato e si ritorna o il punto o un valore booleano che indica la presenza o l'assenza. Con una lista implementata nei nodi, ciò che accade è pressoché simile tranne per la ricerca nell'ultimo sottoalbero che può impiegare un tempo proporzionale alla grandezza della lista. Il valore restituito può essere un boolean true / false oppure il punto in questione o null dipendentemente se il punto si trova nel quadtree oppure no.

```
1  Search(x, y)
2  PointSearched = (x, y)
3  Node = Root
4  while Node ≠ NIL and Node not contains PointSearched do
5      if x ≥ Node.x
6          if y ≥ Node.y
7              Node = Node.NE
8          else
9              Node = Node.SE
10     else
11         if y ≥ Node.y
12             Node = Node.NW
13         else
14             Node = Node.SW
15 if Nodo == NIL
16     return false
17 else
18     return true
```

Algorithm 4: Algoritmo di ricerca senza lista

```
1  Search(x, y)
2  PointSearched = (x, y)
3  Node = Root
4  while Node.SplitPoint ≠ NIL do
5      if Node.SplitPoint == PointSearched
6          return true
7      if x ≥ Node.SplitPoint.x
8          if y ≥ Node.SplitPoint.y
9              Node = Node.NE
10         else
11             Node = Node.SE
12     else
13         if y ≥ Node.SplitPoint.y
14             Node = Node.NW
15         else
16             Node = Node.SW
17 if Node.List contains PointSearched
18     return true
19 else
20     return false
```

Algorithm 5: Algoritmo di ricerca con lista

2.3 Verifica della presenza di una chiave

L'operazione di verifica della presenza di una chiave è simile all'operazione di ricerca, come il valore restituito dal metodo che può essere true/false oppure può essere il punto in questione o null. Ciò che cambia dalla ricerca è l'oggetto cercato e i criteri con cui lo si cerca.

Ad ogni nodo può essere associata una chiave generata in maniera differente. Essendo le chiavi di diverso tipo si possono scrivere diversi tipi di algoritmi che lavorano in modo differente. Assumendo per semplicità che la chiave sia una stringa che viene generata casualmente o assegnata da un client, allora per verificare la sua presenza devono essere analizzati tutti i nodi avendo una complessità finale di $\mathcal{O}(n)$. Quindi in pratica, secondo questa implementazione, la chiave viene cercata usando una coda che contiene i figli di ogni nodo passato. Ad ogni ciclo si estrae un nodo dalla coda e vengono inseriti al suo interno i figli non null. Se si esce dal ciclo lo si fa perché il nodo estratto precedentemente contiene la chiave o perché la coda si svuota e quindi sono stati analizzati tutti i nodi. Il valore di ritorno è true o false dipendentemente se la chiave è stata trovata oppure no.

Quest'algoritmo viene fatto solo dal Point quadtree che non implementa la lista perché durante l'operazione di *Subdivide* un punto diventa non accessibile perché di split point e quindi la struttura perderebbe consistenza.

```
1  KeyPresence(key)
2  queue.insert(root)
3  while Node not contains key and queue ≠ empty do
4      Nodo = queue.front
5      queue.pop
6      if Nodo.NE ≠ NIL
7          queue.push(Nodo.NE)
8      if Nodo.NW ≠ NIL
9          queue.push(Nodo.NW)
10     if Nodo.SE ≠ NIL
11         queue.push(Nodo.SE)
12     if Nodo.SW ≠ NIL
13         queue.push(Nodo.SW)
14 if Nodo contains key
15     return true
16 else
17     return false
```

Algorithm 6: Algoritmo della verifica della presenza di una chiave

2.4 Cancellazione

La cancellazione di un punto è un'operazione possibile se e solo se il punto in questione è in un nodo foglia. Se così non fosse la cancellazione non fa nulla perché eliminare uno split point causerebbe inconsistenza nella struttura del quadtree. La complessità della procedura è la stessa della ricerca in quanto la chiave prima di essere cancellata deve essere trovata. Il tipo di ritorno per semplicità è true o false, ad indicare la corretta cancellazione o meno.

Questa procedura varia in base alla presenza o meno della lista nel nodo. Se una lista non è presente allora la procedura come nella ricerca controlla la presenza di un punto scorrendo i sottoalberi. Se il punto esiste e non è uno split point allora viene cancellato e si ritorna true; se invece esiste ed è uno split point oppure non esiste viene tornato false. Se nel nodo è implementata una lista allora va sempre fatta una operazione di ricerca con le stesse considerazioni precedenti; se è presente il punto in lista allora viene cancellato e si torna true, nel caso in cui non c'è si torna false. Questo accade perché il punto se è di split point non si trova nella lista.

```
1  Cancel(x, y)
2  PointCanc = (x, y)
3  Node = Root
4  while Node ≠ NIL and Node not contains PointCanc do
5      if x ≥ Node.x
6          if y ≥ Node.y
7              Node = Node.NE
8          else
9              Node = Node.SE
10     else
11         if y ≥ Node.y
12             Node = Node.NW
13         else
14             Node = Node.SW
15 if Nodo == NIL
16     return false
17 else
18     if Nodo leafs == NIL
19         Nodo.father = NIL
20         delete Nodo
21         return true
22     else
23         return false
```

Algorithm 7: Algoritmo di cancellazione senza lista

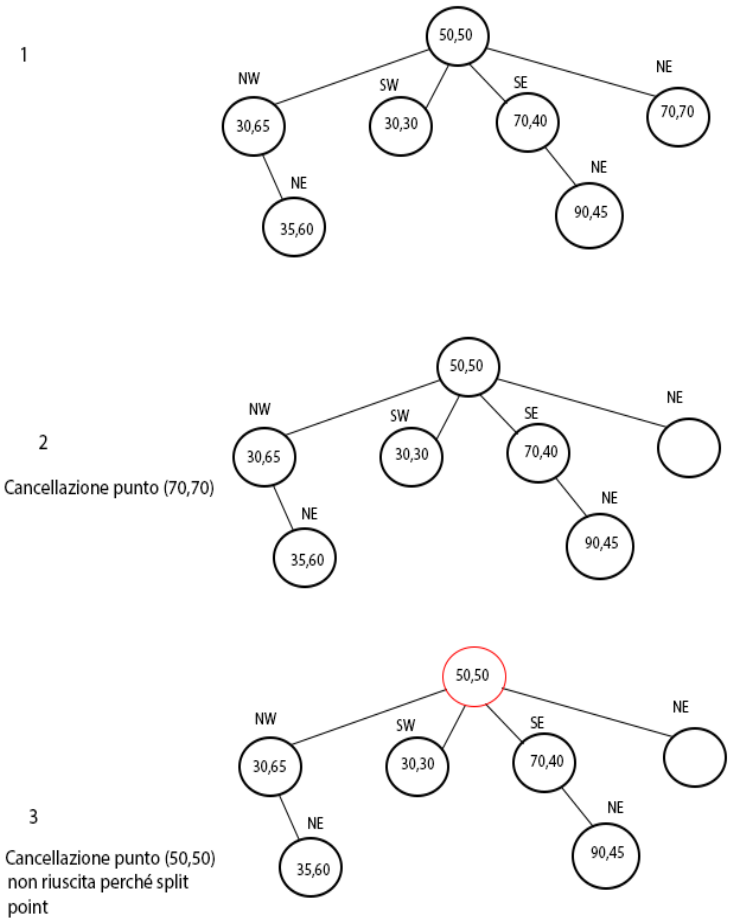

Figura 5: Esempio di cancellazione senza lista

```
1  Cancel(x, y)
2  PointCanc = (x, y)
3  Node = Root
4  while Node.SplitPoint ≠ NIL do
5      if Node.SplitPoint == PointCanc
6          return false
7      if x ≥ Node.SplitPoint.x
8          if y ≥ Node.SplitPoint.y
9              Node = Node.NE
10         else
11             Node = Node.SE
12     else
13         if y ≥ Node.SplitPoint.y
14             Node = Node.NW
15         else
16             Node = Node.SW
17 if Node.List contains PointCanc
18     Leave point from list
19     return true
20 else
21     return false
```

Algorithm 8: Algoritmo di cancellazione con lista

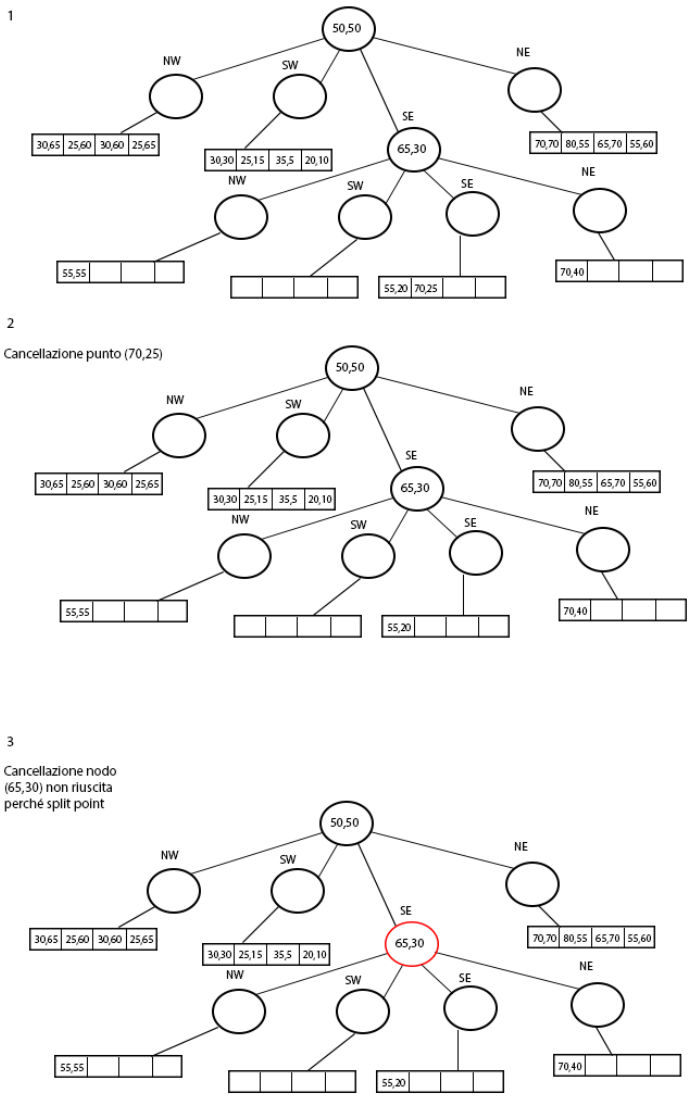

Figura 6: Esempio di cancellazione con lista

2.5 Tabelle con le rispettive complessità

Algoritmo	Complessità caso medio	Complessità caso pessimo
Inserimento	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
Ricerca	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
Ricerca chiave	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Cancellazione	$\mathcal{O}(logn)$	$\mathcal{O}(n)$

Tabella 1: Tabella sulla complessità delle procedure analizzate nel caso in cui non si usa una lista nel nodo

Algoritmo	Complessità caso medio	Complessità caso pessimo
Inserimento	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
Suddivisione	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Ricerca	$\mathcal{O}(logn)$	$\mathcal{O}(n)$
Cancellazione	$\mathcal{O}(logn)$	$\mathcal{O}(n)$

Tabella 2: Tabella sulla complessità delle procedure analizzate nel caso in cui si una lista nel nodo

3 Differenze con altre strutture dati e quadtree

3.1 Differenza con Point-Region quadtree

La differenza nel Point-Region quadtree sta nel fatto che un nodo rappresenta una regione del piano che è caratterizzata dall'avere un singolo valore che si applica a tutta l'area, pertanto il nodo ha un valore uniforme. Tuttavia si possono memorizzare un elenco di punti all'interno di una foglia. La dimensione del nodo è un dato utile in questa struttura. Le complessità delle operazioni sono le stesse del Point quadtree.

3.2 Differenza con k-d tree

Un k-d tree è anch'essa una struttura per immagazzinare dati in uno spazio di k-dimesioni, in questo caso dei punti. I nodi sono degli iperpiani che dividono lo spazio generando dei semispazi, ed in base ai valori dei punti si sceglie su quale dei semispazi collocare il punto. La differenza principale sta nello scaling e nella modificabilità. Un quad tree è più facile da modificare nel corso del tempo rispetto ai k-d tree perché la struttura di quest'ultimo, con semispazi ed iperpiani, è meno dinamica rispetto all'utilizzo di nodi che hanno al loro interno punti. Tuttavia il modo di dividere lo spazio dei k-d tree è nettamente più ottimale e di fatto scalano molto meglio rispetto ai quadtree con l'aumentare dei punti. Le complessità nel caso medio e nel caso pessimo sono le stesse, con la differenza però che nel k-d tree è meno probabile che si arrivi al caso pessimo.

3.3 Recap generale di pro e contro

Da quanto visto si può dedurre che un point quadtree è un'ottima struttura dati per immagazzinare i punti in un tempo medio logaritmico e far si che esso non degeneri in un tempo lineare. Scegliere questa struttura dati conviene quando si ha un numero di nodi non eccessivamente alto e che varia nel tempo poiché rispetto alle altre implementazioni, come visto, ha una maggiore semplicità a supportare i cambiamenti e ad immagazzinare diversi punti in un piano. I contro sono principalmente dalla scelta dell'implementazione del nodo, cioè se contiene o meno una lista. La grandezza della lista se scelta con attenzione non influisce nella complessità generale delle operazioni, ma se si sceglie un numero troppo alto di elementi si perde l'utilità di essa poiché la struttura degenera in ricerche lineari. Non scegliere di utilizzare la lista implica che la suddivisione dello spazio sia più propensa a non essere ottimale e di conseguenza, la struttura, come un albero binario, può degenerare in una lista.