Estruturas de Dados Cinéticas

Marcos Siolin Martins

17 de setembro de 2021

1 Introdução

Em Geometria Computacional visa-se desenvolver algoritmos que resolvam problemas geométricos de maneira eficiente. Há problemas, em que esses algoritmos podem ser usados, que advêm de outras áreas como: tráfego aéreo, computação gráfica, telefonia celular, movimento de partículas, entre outras. Para o desenvolvimento de tais algoritmos, são utilizados resultados de áreas como: geometria euclidiana, teoria dos grafos, combinatória, estruturas de dados e análise de algoritmos.

Os objetos nos problemas podem representar entidades do mundo físico. Por exemplo, pontos podem representar pessoas, aviões, estabelecimentos, entre outras coisas, retas podem representar trajetórias.

Quando é dado um conjunto de objetos geométricos fixo, e deseja-se saber informações de um determinado atributo desses objetos (como, por exemplo, em um conjunto dado de pontos, qual par de pontos possui distância mínima) dizemos que esse é um problema *estático*.

O mesmo problema pode ser formulado, porém sobre um conjunto mutável. Por exemplo, pontos poderiam ser inseridos e removidos ao longo do tempo. Queremos calcular o atributo sem ter que resolver do zero a nova instância do problema estático. Chamamos esse tipo de problema de *dinâmico* ou *on-line*.

Em uma outra formulação, os pontos poderiam estar em movimento contínuo. Essa é a chamada versão *cinética* do problema. É nesse contexto que entram as chamadas *estruturas de dados cinéticas* (*KDS - Kinetic Data Structures*).

Essas estruturas nos permitem realizar consultas de um determinado atributo dos objetos, no instante atual. A garantia de que a estrutura permanece correta se dá através do uso de instrumentos chamados *certificados*. Os certificados estabelecem que uma relação entre um objeto da estrutura e outro se mantém verdadeira até o seu vencimento e devem ajudar na manutenção da estrutura para permitir as consultas desejadas. Poderíamos usar certificados por exemplo entre alguns pares de pontos que, considerando suas trajetórias

atuais, garantissem que, até o instante t, a ordenada de um dos pontos é maior que a ordenada do outro ponto. Ao atingirmos o instante t, o certificado vence, implicando em uma mudança estrutural no conjunto de pontos que pode afetar o resultado de futuras consultas, requerendo assim possíveis ajustes na estrutura de dados, e eventual cálculo de novos certificados.

A seguir estão alguns problemas e maneiras de resolvê-los eficientemente usando essas estruturas.

2 Ordenação Cinética

Considere o seguinte problema cinético. São dados n pares de valores. Cada par (x_0, v) representa um valor que está mudando linearmente com o tempo. Num instante arbitrário $t \geq 0$, o valor correspondente ao par (x_0, v) é $x_0 + tv$. O objetivo é responder consultas do tipo: para um certo i, com $1 \leq i \leq n$, quem é o i-ésimo maior valor da coleção no instante corrente.

Por exemplo, se tivermos quatro elementos na coleção, digamos $\left(6, -\frac{1}{2}\right)$, $(5, 0)$, $\left(3, \frac{1}{4}\right)$ e $\left(0, \frac{4}{3}\right)$, podemos representar essa coleção como na figura 1.

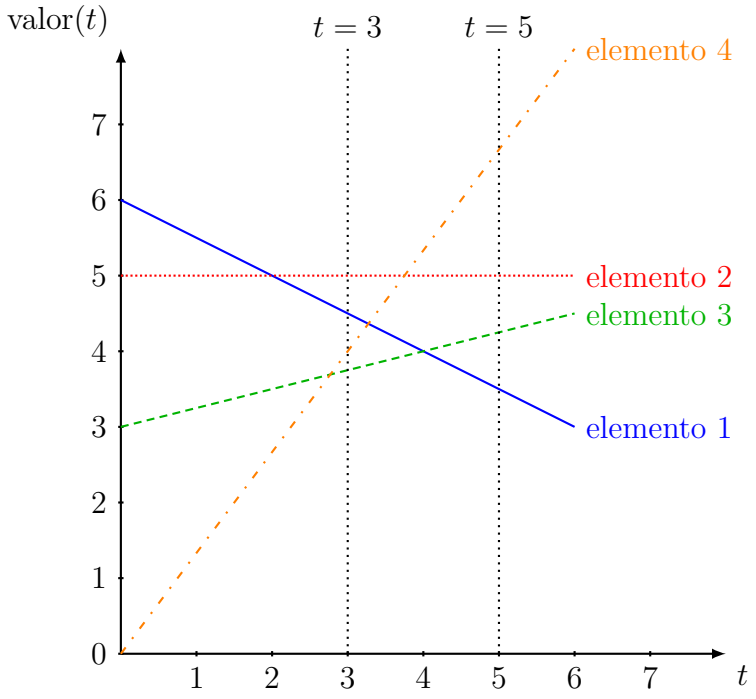


Figura 1: O segundo maior valor no instante $t = 0$ é do elemento 2, e no instante $t = 3$ é do elemento 1. O menor valor da coleção no instante $t = 3$ é do elemento 3, e no instante $t = 5$ é do elemento 1.

Queremos dar suporte às seguintes operações:

- ADVANCE(t) $\rightarrow$ avança o tempo corrente para t;
- CHANGE(j, v) $\rightarrow$ altera a velocidade do elemento j para v;
- QUERY_KTH(i) $\rightarrow$ devolve o elemento cujo valor é o i-ésimo maior no instante atual.

2.1 Lista Ordenada Cinética

Um jeito natural de resolver o problema da lista ordenada cinética é manter um vetor com os elementos da lista em ordem decrescente do valor no instante atual.

Inicialmente o vetor começa com os valores dos elementos no instante $t = 0$, ou seja, com o valor x_0 de cada elemento, e este vetor é ordenado em ordem decrescente. Na verdade, o vetor pode armazenar não os valores, mas os índices dos elementos, e fazemos ordenação indireta. No caso de empates nos valores dos elementos, o desempate será feito pela velocidade, ou seja, se dois elementos, digamos i e j, possuem o mesmo valor x_0, mas a velocidade de i é maior que a de j, então i será tratado como se possuísse maior valor que j no instante inicial. Esse mesmo critério de desempate será aplicado em todos os instantes e também em todos os problemas daqui em diante.

Uma vez de posse do vetor ordenado com os valores iniciais decrescentemente, construímos um certificado para cada par de elementos consecutivos no vetor. O i-ésimo certificado, denotado pelo par (i, t), se refere ao par das posições i e $i + 1$. O valor t consiste no instante de tempo em que o i-ésimo elemento deixará de ter um valor maior que o valor do $(i + 1)$-ésimo elemento do vetor, se esse instante for maior ou igual a 0, ou em geral ao instante atual. Do contrário, o valor t consiste em $+\infty$. O valor t do certificado é o seu prazo de validade.

Esses prazos de validade determinam os eventos que potencialmente causarão modificações no vetor que mantém os elementos ordenados pelo seu valor e consequentemente em alguns certificados.

Esses $n - 1$ certificados são colocados em uma fila com prioridades, com o prazo de validade determinando a prioridade. Estamos interessados nos certificados com menor prazo de validade. Ou seja, a fila com prioridades pode ser implementada com um heap de mínimo que usa os prazos de validade como chave.

Para descrever a implementação das três operações, precisamos estabelecer o nome das variáveis usadas. São elas:

1. n: o número de elementos dados;
2. x_0 e *speed*: vetores com o valor e a velocidade inicial de cada um dos n elementos;

3. *now*: instante atual;

4. *sorted*: vetor com os índices dos n elementos em ordem decrescente do seu valor no instante *now*;

5. *indS*: vetor de n posições; $indS[i]$ guarda a posição em *sorted* do elemento i;

6. *cert*: vetor com os $n-1$ certificados; $cert[i]$ guarda o certificado entre $sorted[i]$ e $sorted[i+1]$, para $1 \leq i < n$;

7. Q: fila com prioridades para os certificados.

A interface da fila com prioridades que utilizaremos inclui as duas seguintes operações:

1. MINPQ(Q): devolve o certificado (i, t) com chave t mínima em Q;

2. UPDATEPQ(Q, i, t): altera a chave do i-ésimo certificado para t e ajusta Q de acordo.

O vetor *indS* nos permite implementar a operação CHANGE de maneira eficiente, pois, dado um elemento j, precisamos saber a posição i do elemento j em *sorted* para recalcular os certificados relacionados com a posição i.

Para implementar a operação UPDATEPQ(Q, i, t) em tempo logarítmico no número de elementos na fila Q, é necessário utilizar um vetor adicional *indQ* que guarda em $indQ[i]$ a posição do i-ésimo certificado em Q.

Com isso, a operação ADVANCE(t) segue uma ideia bem simples: enquanto t for maior que o prazo de validade do próximo evento, avançamos *now* para esse prazo de validade e tratamos esse evento. Nos problemas seguintes, a operação ADVANCE(t) será sempre a mesma; as únicas mudanças ocorrerão no tratamento de um evento. Um evento está associado a um certificado (i, t) que expira quando $now = t$. O tratamento do evento correspondente ao certificado (i, t) consiste em trocar de lugar os índices das posições i e $i+1$ do vetor *sorted*, recalcular o prazo de validade do $(i-1)$-ésimo certificado se $i > 1$, e do $(i+1)$-ésimo certificado se $i < n-1$. O i-ésimo certificado também deve ser ajustado para $+\infty$. Finalmente, é necessário fazer ajustes em Q, alterando a chave dos certificados que sofreram alteração.

Algoritmo 2.1: Função ADVANCE.

```
1: function ADVANCE(t)
2:     if t < now :
3:         return
4:     while t ≥ cert[MINPQ(Q)] :
5:         now ← cert[MINPQ(Q)]
6:         EVENT()
7:     now ← t
```

Na implementação da operação EVENT, utilizaremos a rotina UPDATE(i) para calcular a nova validade t do i-ésimo certificado, se $1 \leq i < n$, e fazer os devidos ajustes em Q. Para calcular t, utilizaremos uma rotina chamada EXPIRE(i, j), que calcula a validade dos certificados entre os elementos i e j. A rotina auxiliar EXPIRE(i, j) não mudará para outros problemas, mantendo a mesma definição.

Algoritmo 2.2: Função UPDATE.

```
1: function UPDATE(i)
2:     if 1 ≤ i < n :
3:         t ← EXPIRE(i, i + 1)
4:         UPDATEPQ(Q, i, t)
```

Algoritmo 2.3: Função EVENT.

```
1: function EVENT()
2:     i ← MINPQ(Q)
3:     while cert[i] = now :
4:         sorted[i] ↔ sorted[i + 1]
5:         indS[sorted[i]] ↔ indS[sorted[i + 1]]
6:         UPDATE(i)
7:         UPDATE(i − 1)
8:         UPDATE(i + 1)
9:         i ← MINPQ(Q)
```

As figuras 2 e 3 ilustram o tratamento do evento de expiração do segundo certificado.

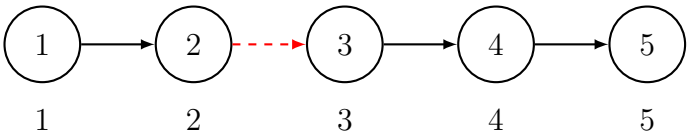


Figura 2: *cert*[2] expirou.

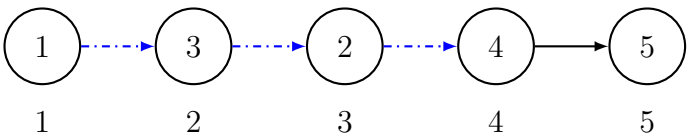


Figura 3: *sorted*[2] e *sorted*[3] foram trocados e *cert*[1], *cert*[2], *cert*[3] foram atualizados.

A operação QUERY_KTH(i) consiste em devolver *sorted*[i], enquanto que a operação CHANGE(j, v) consiste em alterar a posição $x_0[j]$ para $x_0[j] + (speed[j] - v) \cdot now$, a posição

speed[j] para v e recalcular os eventuais certificados de que j participa. O novo valor da posição $x_0[j]$ corresponde à posição inicial do elemento caso ele tivesse começado com essa velocidade e estivesse na posição atual agora. Para tanto, a partir da posição i em que j se encontra no vetor *sorted*, podemos recalcular $cert[i-1]$ se $i > 1$ e $cert[i]$ se $i < n$, como ilustrado na figura 4, acionando a rotina UPDATEPQ para fazer os devidos acertos em Q correspondentes a estas modificações.

Algoritmo 2.4: Função QUERY_KTH.

```
1: function QUERY_KTH(i)
2:     if 1 ≤ i ≤ n :
3:         return sorted[i]
4:     return −1
```

Algoritmo 2.5: Função CHANGE.

```
1: function CHANGE(j, v)
2:     x_0[j] ← x_0[j] + (speed[j] − v) · now;
3:     speed[j] ← v
4:     i ← indS[j]
5:     UPDATE(i)
6:     UPDATE(i − 1)
```

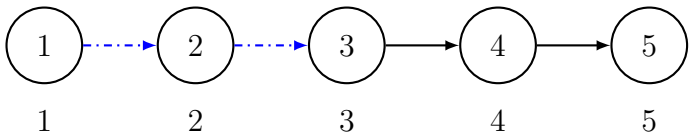


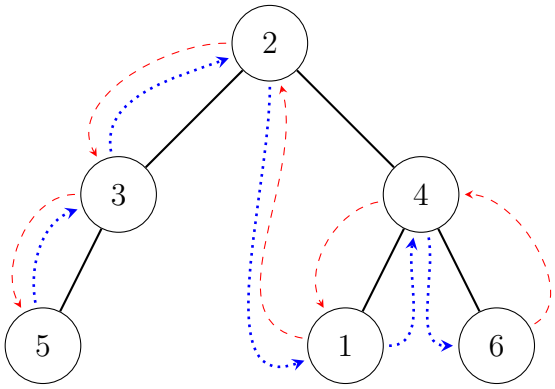
Figura 4: Após a mudança de trajetória do elemento 2, que se encontra em *sorted*[2], *cert*[1] e *cert*[2] foram atualizados.

2.2 Árvore binária de busca balanceada

Manter um vetor ordenado é uma boa maneira de resolver o problema da lista ordenada cinética dando suporte às operações ADVANCE(t), CHANGE(j, v) e QUERY_KTH(i). Poderíamos também querer dar suporte, além das operações citadas, às seguintes operações:

- INSERT(v, x_t) → insere um elemento com velocidade v e valor x_t no instante *now*;
- DELETE(i) → remove o elemento i no instante *now*.

Para inserir um elemento no vetor, antes teríamos de encontrar a posição que este deveria ocupar no vetor, digamos que seja a posição j. Após encontrar a posição, movemos todos



i	x_0
1	6
2	3
3	2
4	7
5	-2
6	14

Figura 5: Exemplo de árvore em que a ordem dos elementos, do menor para o maior no instante $now = 0$, é $5 - 3 - 2 - 1 - 4 - 6$. Os apontadores para o elemento anterior são representados pelas setas vermelhas tracejadas e os apontadores para o elemento posterior são representados pelas setas azuis pontilhadas.

os elementos, a partir da posição j, uma posição à frente e colocamos o elemento na posição j. Após isso, os certificados de j até $n - 1$ devem ser atualizados e um novo certificado será criado, o n-ésimo certificado, o total de elementos n deve ser mudado para $m = n + 1$. O novo certificado também deve ser inserido na fila de prioridade.

Essa sequência de operações pode se tornar pouco eficiente com uma grande quantidade de elementos sendo inseridos no começo do vetor, consumindo tempo linear por inserção. Como a remoção envolve uma sequência de operações parecida, da mesma maneira se torna pouco eficiente, também consumindo tempo linear.

Dessa forma, apesar da lista ordenada cinética implementada manipulando um vetor ser uma estrutura eficiente para a operação QUERY_KTH(i), com um consumo de tempo constante, o consumo de tempo para as operações INSERT(v, x_t) e DELETE(i) é, no pior caso, proporcional ao número de elementos, o que pode ser ruim para uma grande quantidade de elementos, inserções e remoções.

Podemos equilibrar o consumo de tempo das operações QUERY_KTH(i), INSERT(v, x_t) e DELETE(i) em tempo logarítmico no número de elementos, usando uma ABBB (árvore binária balanceada de busca). Os pontos serão armazenados na ABBB tendo o seu valor no instante *now* como chave.

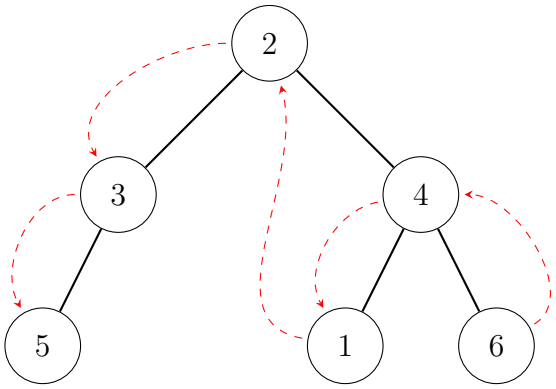
Além da ABBB, para garantirmos a eficiência das operações EVENT, CHANGE, INSERT e DELETE, cada elemento terá um apontador para o seu predecessor e um apontador para o seu sucessor, formando uma lista duplamente ligada ordenada pelo valor do elemento no instante *now*, veja a figura 5.

No que diz respeito aos certificados, antes um certificado estava associado a uma posição e, no vetor, ao inserirmos um elemento em uma determinada posição, teríamos que deslocar todos os certificados conseguintes àquela posição. Agora, para que consigamos

alterar apenas uma quantidade constante de certificados após uma inserção, os certificados não estarão mais associados a uma posição e sim aos elementos.

O certificado i se refere à relação estabelecida entre o elemento i e seu predecessor e consiste no instante de tempo em que o elemento i deixará de ter um valor maior que o valor do seu predecessor, se esse instante for maior que o instante atual. Do contrário, o certificado consiste em $+\infty$. Se o elemento i não possui predecessor, então o certificado também consiste em $+\infty$.

Esses n certificados também serão colocados em uma fila de prioridade, com o prazo de validade como chave. A fila de prioridade agora também deverá suportar operações como a inserção e remoção de certificados.



i	x_0	v	$cert[i]$
1	6	2	1
2	3	5	$+\infty$
3	2	1	2
4	7	4	$+\infty$
5	-2	3	$+\infty$
6	14	0.5	2

Figura 6: Certificados representados pelas setas vermelhas tracejadas. O certificado do elemento 5 vale $+\infty$.

Para descrever as implementações das operações, vamos estabelecer os nomes dos objetos, variáveis e rotinas auxiliares utilizados:

1. n: número de elementos no instante *now*;
2. *node*: objeto que compõe a árvore binária balanceada de busca, atributos:
   (a) *left*: aponta para a raiz da subárvore esquerda do nó;
   (b) *right*: aponta para a raiz da subárvore direita do nó;
   (c) *key*: aponta para um elemento;
   (d) *size*: quantidade de nós que as subárvores deste nó possuem. Este atributo será importante para a operação QUERY_KTH(i).
3. *root*: nó que é a raiz da árvore de busca binária balanceada;
4. *element*: objeto com os seguintes atributos:

(a) *id*: vem de *identifier* e é o atributo para identificar o objeto. Assim, daqui em diante, usaremos elemento i para se referir ao elemento cujo *id* é i;

(b) *speed*: a velocidade do elemento;

(c) x_0: é o valor que o elemento possuía no instante $t = 0$;

(d) *next*: é o atributo que aponta para o elemento imediatamente posterior a este na coleção, no instante *now*. O elemento imediatamente posterior a i é aquele que possui o menor valor dentre a coleção de elementos que possuem valor maior que o elemento i;

(e) *prev*: vem de *previous* e é o atributo que aponta para o elemento imediatamente anterior a este na coleção, no instante *now*;

(f) *pqpos*: vem de *priority queue position* e é o atributo que aponta para a posição do certificado associado ao elemento na fila de prioridade;

(g) *cert*: vem de *certificate* e é o tempo de validade do certificado entre este elemento e o elemento apontado por *prev*; se *prev* não aponta para ninguém, *cert* vale $+\infty$.

(h) *node*: aponta para o nó da árvore binária de busca em que o elemento se encontra.

5. Q: fila de prioridade que contém os elementos; o elemento com certificado de menor valor estará à frente da fila;

6. INSERTKEY$(root, e) \rightarrow$ insere e, um elemento, na árvore binária balanceada de busca com raiz *root* e retorna a, possivelmente nova, raiz da árvore. No processo também atualiza a lista ligada de elementos;

7. DELETEKEY$(root, e) \rightarrow$ remove e, um elemento, da árvore binária balanceada de busca com raiz *root* e retorna a, possivelmente nova, raiz da árvore. No processo também atualiza a lista ligada de elementos;

Para a implementação das operações CHANGE(j, v) e DELETE(i), precisamos de alguma maneira recuperar um elemento baseado no seu *id*. Para tal, podemos utilizar uma tabela de símbolos, implementada por uma árvore binária balanceada de busca ou uma tabela de dispersão. A seguir estão três operações que nos ajudarão a recuperar os elementos:

1. GETOBJECT$(i) \rightarrow$ retorna o elemento i;

2. INSERTOBJECT$(e) \rightarrow$ insere e, que é um elemento, na estrutura;

3. DELETEOBJECT$(e) \rightarrow$ remove e, que é um elemento, da estrutura.

Para permitir a inserção e remoção de certificados, a interface da fila de prioridade será reformulada, contando com duas operações extras:

1. INSERTPQ$(Q, e) \rightarrow$ insere e na fila de prioridade Q;
2. DELETEPQ$(Q, e) \rightarrow$ remove e da fila de prioridade Q;
3. UPDATEPQ$(Q, e, t) \rightarrow$ muda o valor do certificado de e para t e atualiza a fila de prioridade Q;
4. MINPQ$(Q) \rightarrow$ devolve o elemento com o certificado de menor valor da fila de prioridade Q.

A operação UPDATEPQ(Q, e, t) pode ser implementada em tempo logarítmico graças ao atributo *pqpos* dos elementos.

Um evento está associado a um certificado (e, t) que expira no instante t. O tratamento do evento correspondente ao certificado (e, t) consiste em trocar de lugar o elemento e e seu predecessor, digamos e', na árvore binária de busca e na lista ligada, e recalcular o prazo de validade de até 3 certificados:

- O certificado de e;
- O certificado de e';
- O certificado do novo sucessor de e'.

Na implementação da operação EVENT, utilizaremos a rotina UPDATE(e) que calcula a nova validade t do certificado do elemento e, e chama a rotina UPDATEPQ(Q, e, t).

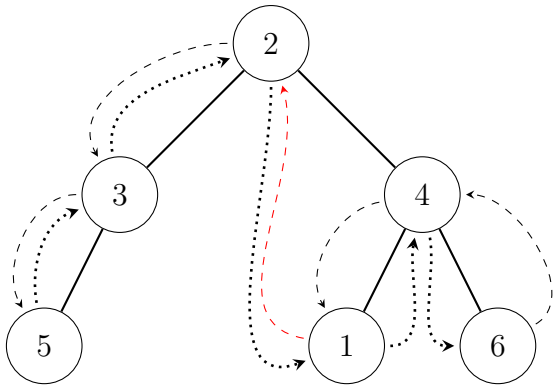
Algoritmo 2.6: Função UPDATE.

```
1: function UPDATE(e)
2:     if e ≠ NULL :
3:         e' ← e.next
4:         t ← EXPIRE(e, e')
5:         UPDATEPQ(Q, e, t)
   ▷ EXPIRE(e, e') calcula a validade do certificado entre os elementos e e e', se e' é NULL
   retorna +∞
```

No algoritmo abaixo a função SWAP(e_1, e_2) troca a posição de e_1 e e_2 na árvore binária balanceada de busca e na lista ligada.

Algoritmo 2.7: Função EVENT.

```
1: function EVENT()
2:     e_1 ← MINPQ(Q)
3:     while e_1.cert = now :
4:         e_2 ← e_1.prev
5:         SWAP(e_1, e_2)
6:         UPDATE(e_1)
7:         UPDATE(e_2)
8:         UPDATE(e_2.next)
9:         e_1 ← MINPQ(Q)
```



i	x_0	v	$now = 0$ $cert[i]$	$now = 1$ $cert[i]$
1	6	2	1	$+\infty$
2	3	5	$+\infty$	$+\infty$
3	2	1	2	2
4	7	4	$+\infty$	4
5	-2	3	$+\infty$	$+\infty$
6	14	0.5	2	2

Figura 7: Certificado do elemento 1 expirou em $now = 1$.

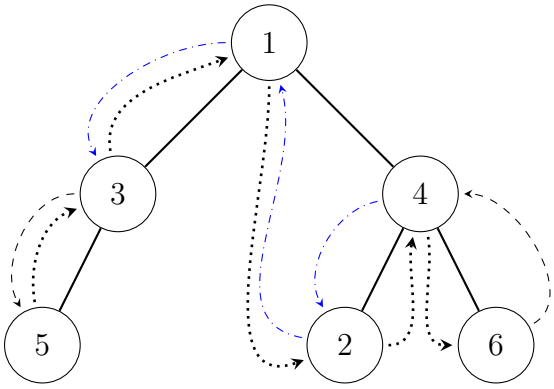


Figura 8: Em $now = 1$, o elemento 1 e o elemento 2 foram trocados e os certificados do elemento 1, elemento 2 e elemento 4 foram atualizados.

A operação QUERY_KTH(i) consiste em devolver o i-ésimo maior elemento da lista ligada. Para tal, usaremos a árvore binária balanceada de busca. Estando em um determinado nó *node* da árvore, sabemos que todos os nós na subárvore direita tem valor maior

que o nó atual e que os nós da subárvore esquerda. Portanto, se $i \leq node.right.size$, a nossa resposta com certeza está na subárvore direita do nó. Caso contrário temos duas opções: *node* é a resposta ou a resposta está na subárvore esquerda. Para checar se *node* é a resposta devemos perceber que *node* tem valor maior que os nós de sua subárvore esquerda, então se $i = node.right.size+1$, *node* é a resposta. Se $i > node.right.size+1$, então a nossa resposta está na subárvore esquerda e queremos o $[i-(node.right.size+1)]-$ésimo elemento da subárvore esquerda. Podemos repetir esse processo até encontrar a nossa resposta. O algoritmo 2.8 utiliza a rotina auxiliar RSIZE(r), que devolve o valor de *size* de $r.right$ caso este seja não nulo, caso contrário devolve 0.

Algoritmo 2.8: Função QUERY_KTH.

```
1:  function QUERY_KTH(i)
2:      node ← root
3:      r ← RSIZE(node)
4:      while i ≠ r + 1 :
5:          if i ≤ r :
6:              node ← node.right
7:          else
8:              node ← node.left
9:              i ← i − (r + 1)
10:         r ← RSIZE(node)
11:     return node.key
```

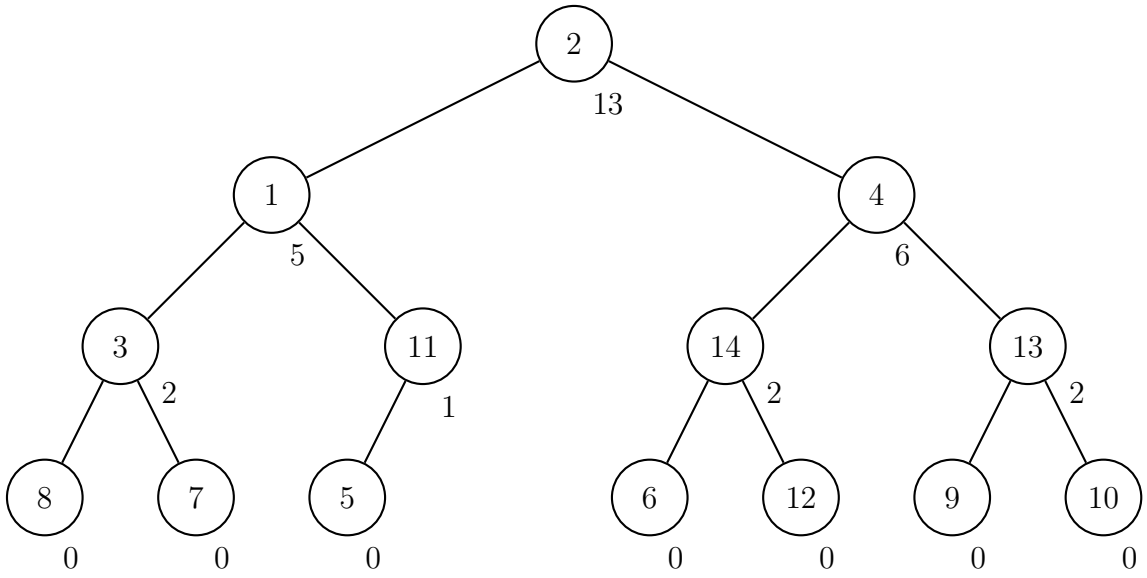


Figura 9: Se quiséssemos encontrar o 7° elemento na árvore acima, $i = 7$, desviamos para a direita, pois a quantidade de nós da subárvore direita da raiz é $\leq i$. Depois, desviamos para a esquerda e redefinimos i como 3, pois existem 4 elementos à frente. Como o nó do elemento 14 só tem um nó em sua subárvore direita, desviamos para a esquerda de novo e redefinimos i como 1. Como a quantidade de nós da subárvore direita do nó do elemento 6 somada a 1 é igual a i, o elemento 6 é o 7° elemento da coleção.

A operação CHANGE(j, v) consiste em recuperar o elemento e com identificador j, alterar seu atributo x_0 para $x_0 + (speed - v) \cdot now$, *speed* para v e recalcular os eventuais certificados de que j participa, que seriam $e.cert$ e $e.next.cert$, se $e.next$ existe.

Algoritmo 2.9: Função CHANGE.

```
1: function CHANGE(j, v)
2:     e ← GETOBJECT(j)
3:     e.x_0 ← e.x_0 + (speed[j] − v) · now;
4:     e.speed ← v
5:     UPDATE(e)
6:     UPDATE(e.next)
```

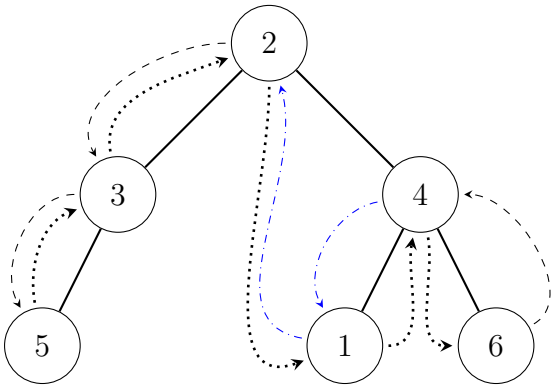


Figura 10: Após chamar CHANGE$(1, 5)$ a velocidade do elemento 1 foi alterada para 5 e os certificados do elemento 1 e do elemento 4 foram atualizados.

A operação INSERT(v, x_t) consiste em criar um novo elemento, inicializando seus atributos com os devidos valores, inseri-lo na árvore binária balanceada de busca e na estrutura que usamos para recuperá-lo depois, calcular o seu certificado e inseri-lo na fila de prioridade e, por fim, atualizar o certificado de seu sucessor, caso exista. Uma importante o bservação é que se $now \neq 0$, então $x_t \neq x_0$. Para calcular x_0, podemos utilizar a relação $x_t = now \cdot speed + x_0 \Rightarrow x_0 = x_t - speed \cdot now$.

Algoritmo 2.10: Função INSERT.

```
1: function INSERT(v, x_t)
2:     e.speed ← v
3:     e.x_0 ← x_t − now · v
4:     root ← INSERTKEY(root, e)
5:     INSERTOBJECT(e)
6:     e.cert ← EXPIRE(e, e.prev)
7:     INSERTPQ(Q, e)
8:     UPDATE(e.next)
```

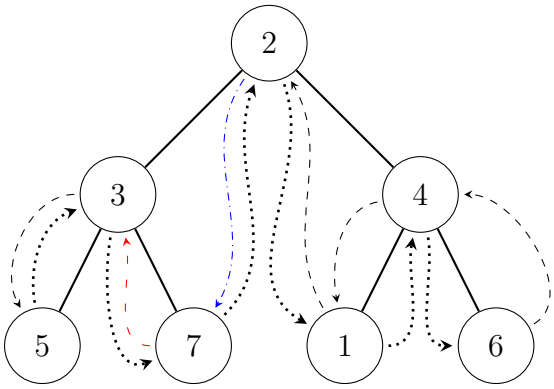


Figura 11: Após chamar INSERT$(1, 3)$, no instante 2, o elemento 7 foi inserido na árvore. O certificado do elemento 7 foi criado e o certificado do seu sucessor, o elemento 2, atualizado.

A operação DELETE(i) consiste em recuperar o elemento i, removê-lo da árvore binária balanceada de busca e da estrutura que usamos para recuperá-lo, e depois removê-lo da fila de prioridade. Após isso, basta atualizar o certificado de seu sucessor, caso exista.

Algoritmo 2.11: Função DELETE.

```
1: function DELETE(i)
2:     e ← GETOBJECT(i)
3:     e' ← e.next
4:     root ← DELETEKEY(root, e)
5:     DELETEOBJECT(e)
6:     DELETEPQ(Q, e)
7:     UPDATE(e')
```

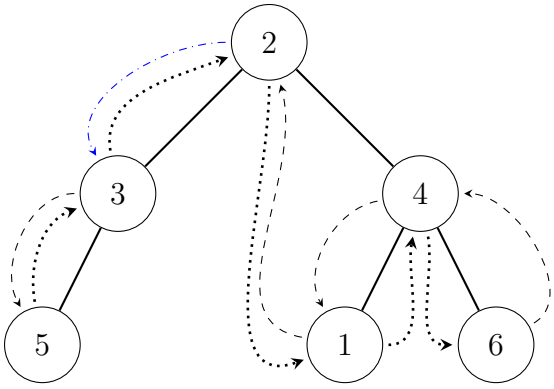


Figura 12: Após chamar DELETE(7), o elemento 7 foi retirado da árvore, a lista ligada foi ajustada e o certificado do seu sucessor, elemento 2, foi atualizado.

3 Máximo Cinético

Agora, considere o seguinte problema cinético. São dados n pares de valores em que cada par (x_0, v) representa um valor que está mudando linearmente com o tempo, assim como na lista cinética. Num instante arbitrário $t \geq 0$, o valor correspondente ao par (x_0, v) é $x_0 + tv$. Desta vez, o objetivo é responder consultas mais simples, do tipo: quem é o elemento com maior valor da coleção no instante corrente.

Utilizando o mesmo exemplo da lista, se tivermos quatro elementos na coleção, digamos $\left(6, -\frac{1}{2}\right)$, $(5, 0)$, $\left(3, \frac{1}{4}\right)$ e $\left(0, \frac{4}{3}\right)$, podemos representar essa coleção da seguinte maneira:

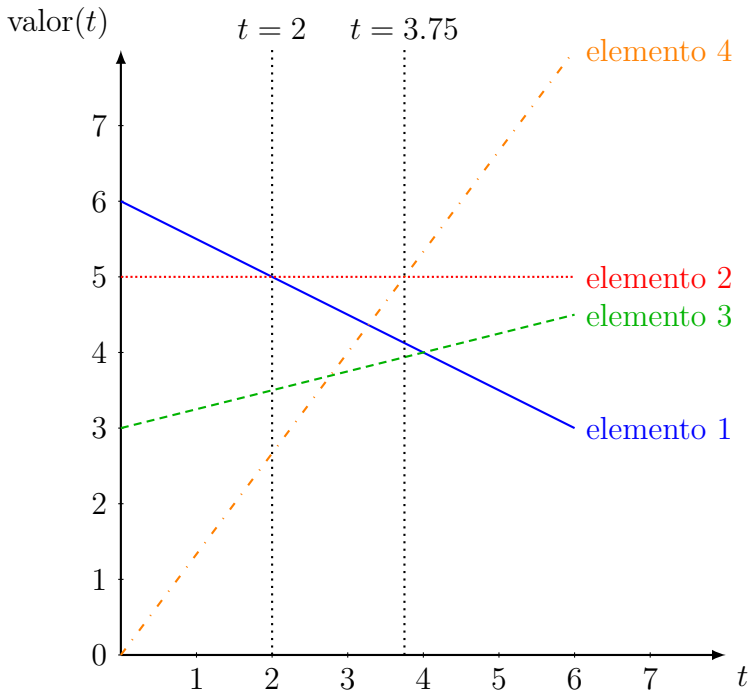


Figura 13: O maior valor até o instante $t = 2$ é o do elemento 1. Em seguida, o maior valor passa a ser o do elemento 2 até o instante $t = 3.75$, que é quando o elemento 4 passa a superar todos os outros.

Agora, além das operações ADVANCE(t) e CHANGE(j, v), queremos dar suporte à nova operação:

- QUERY_MAX() $\rightarrow$ devolve o elemento cujo valor é o maior no instante atual.

3.1 Heap Cinético

Um bom jeito de resolver o problema do máximo cinético é manter uma fila de prioridades com os elementos da coleção. Dessa maneira, o elemento que se encontra na raiz da fila

será o que possui o maior valor da coleção. Para implementar a heap utilizaremos um vetor.

Inicialmente o vetor começa com os índices dos elementos e construímos um heap de acordo com o valor de cada elemento no instante $t = 0$, ou seja, com o valor x_0 de cada elemento.

Uma vez de posse do heap montado, construímos um certificado para cada par (filho, pai) no heap. O i-ésimo certificado, que se refere ao par das posições i e $\left\lfloor \frac{i}{2} \right\rfloor$, consiste no instante de tempo em que o i-ésimo elemento passará a ter um valor maior que o valor do $\left\lfloor \frac{i}{2} \right\rfloor$-ésimo elemento do vetor, se esse instante for maior que o instante atual. Do contrário, o certificado consiste em $+\infty$.

Esses $n - 1$ certificados são colocados em uma fila de prioridade Q, com o prazo de validade como chave. Estamos interessados nos certificados com menor prazo de validade.

Para descrever a implementação das três operações, precisamos estabelecer o nome das novas variáveis usadas. São elas:

1. *heap*: vetor com os índices dos n elementos formando um heap de acordo com o seu valor no instante *now*;
2. *cert*: vetor com os certificados, onde *cert*$[i]$ guarda o certificado entre i e $\left\lfloor \frac{i}{2} \right\rfloor$, para $1 < i \leq n$.

A interface da fila de prioridade que utilizaremos não se altera.

Um evento está associado a um certificado (i, t) que expira no instante t. O tratamento do evento correspondente ao certificado (i, t) consiste em trocar de lugar os índices armazenados nas posições i e $\left\lfloor \frac{i}{2} \right\rfloor$ do vetor *heap*, e recalcular o prazo de validade de até cinco certificados:

- do $\left\lfloor \frac{i}{2} \right\rfloor$-ésimo certificado, se $i > 1$;
- do j-ésimo certificado, se $i > 1$ e $j \leq n$, onde $j = 2 \cdot \left\lfloor \frac{i}{2} \right\rfloor + ((i + 1) \mod 2)$ é o irmão de i;
- do $(2i)$-ésimo certificado, se $2i \leq n$;
- do $(2i + 1)$-ésimo certificado, se $2i + 1 \leq n$.

O i-ésimo certificado também deve ser ajustado para $+\infty$. Finalmente, é necessário fazer ajustes em Q, alterando a chave dos certificados que sofreram alteração.

Novamente, na implementação da operação EVENT, utilizaremos a rotina UPDATE(i) que calcula a nova validade t do i-ésimo certificado, se $1 < i \leq n$, e chama a rotina UPDATEPQ(Q, i, t).

Algoritmo 3.12: Função UPDATE.

```
1: function UPDATE(i)
2:     if 1 < i ≤ n :
3:         t ← EXPIRE(i, ⌊i/2⌋)
4:         UPDATEPQ(Q, i, t)
```

Algoritmo 3.13: Função EVENT.

```
1: function EVENT
2:     i ← MINPQ(Q)
3:     while cert[i] = now :
4:         heap[i] ↔ heap[⌊i/2⌋]
5:         UPDATE(i)
6:         UPDATE(⌊i/2⌋)
7:         UPDATE(2 · ⌊i/2⌋ + ((i + 1) mod 2))
8:         UPDATE(2i)
9:         UPDATE(2i + 1)
10:        i ← MINPQ(Q)
```

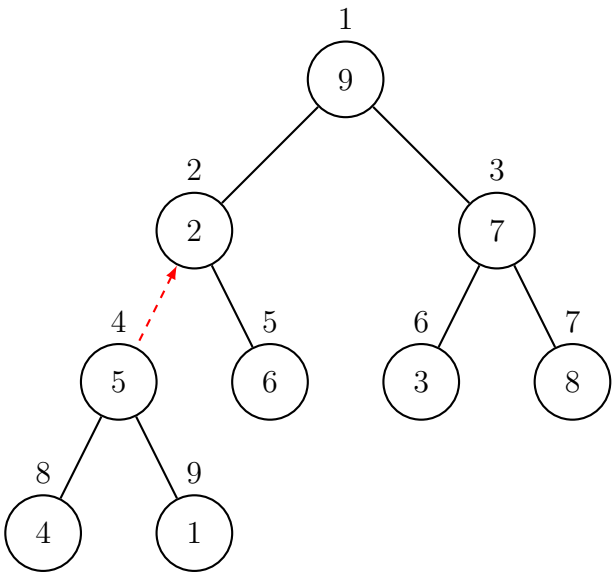


Figura 14: *cert*[4] expirou.

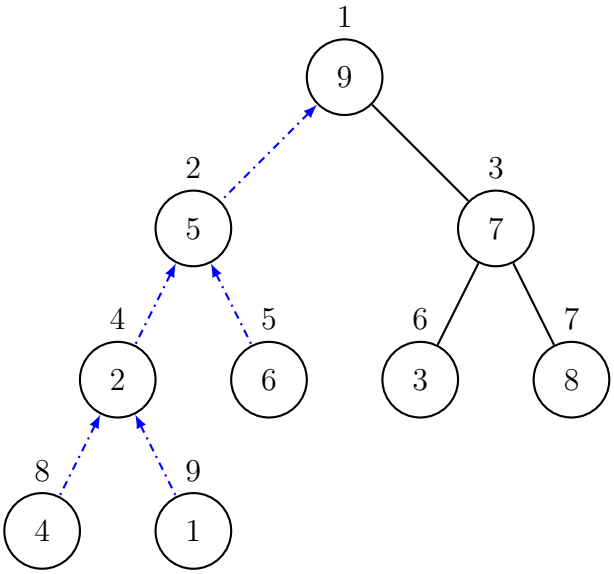


Figura 15: *heap*[4] e *heap*[2] foram trocados e *cert*[2], *cert*[4], *cert*[5], *cert*[8] e *cert*[9] foram atualizados.

A operação QUERY_MAX() consiste em devolver *heap*[1], enquanto que a operação CHANGE(j, v) consiste em alterar a posição $x_0[j]$ para $x_0[j] + (speed[j] - v) \cdot now$, a posição *speed*[j] para v e recalcular os eventuais certificados de que j participa. Para tanto, a partir da posição i em que j se encontra no vetor *heap*, podemos recalcular *cert*$[i]$ se $i > 1$, *cert*$[2i]$ se $2i \leq n$ e *cert*$[2i + 1]$ se $2i + 1 \leq n$, acionando a rotina UPDATEPQ para fazer os devidos acertos em Q correspondentes a estas modificações.

Algoritmo 3.14: Função QUERY_MAX.

```
1: function QUERY_MAX
2:     return heap[1]
```

Algoritmo 3.15: Função CHANGE.

```
1: function CHANGE(j, v)
2:     x_0[j] ← x_0[j] + (speed[j] - v) · now;
3:     speed[j] ← v
4:     UPDATE(i)
5:     UPDATE(2i)
6:     UPDATE(2i + 1)
```

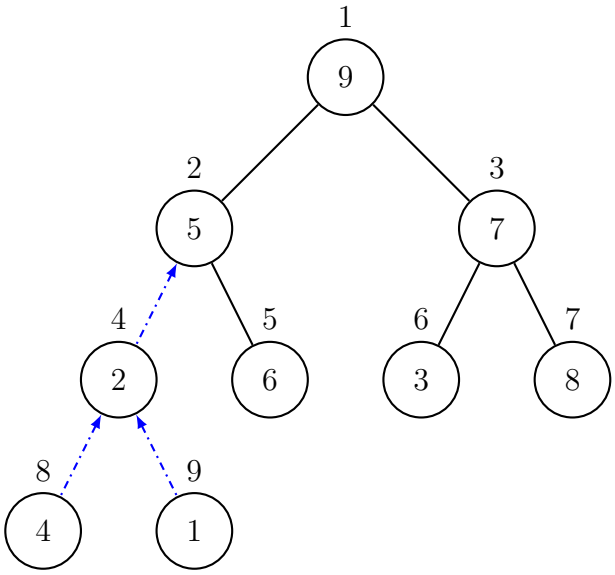


Figura 16: Após a mudança de velocidade do elemento 2, que se encontra em $heap[4]$, $cert[4]$, $cert[8]$ e $cert[9]$ foram atualizados.

3.2 Torneio Cinético

Considere o seguinte algoritmo para achar o valor máximo em um conjunto de n elementos: aloque um vetor $tourn$ com $2n-1$ posições. Inicializamos as últimas n posições com os valores dos n elementos e uma variável i com o valor da última posição, $i = 2n-1$. Repita o seguinte processo até que i seja igual a 1: se $tourn[i] > tourn[i-1]$, então $tourn\left[\left\lfloor \frac{i}{2} \right\rfloor\right] = tourn[i]$, caso contrário $tourn\left[\left\lfloor \frac{i}{2} \right\rfloor\right] = tourn[i-1]$, e, por fim, subtraia 2 de i. Dessa maneira, ao fim da execução do algoritmo, em $tourn[1]$ estará o maior valor da coleção. Na verdade, podemos fazer a comparação de maneira indireta, e guardar os índices dos elementos no vetor $tourn$ e não seus valores.

Podemos pensar nessas comparações entre $tourn[i]$ e $tourn[i-1]$ como sendo partidas de um torneio classificatório, por isso o nome torneio. Chamaremos de "partida" as comparações entre $tourn[i]$ e $tourn[i-1]$ e diremos que o elemento j "vence" o elemento k quando os elementos j e k disputaram uma partida entre si e o elemento j possuía maior valor nesse instante. Utilizaremos esse "torneio" para resolver o problema do máximo cinético e, para implementá-lo, será utilizado um vetor como o citado no algoritmo acima.

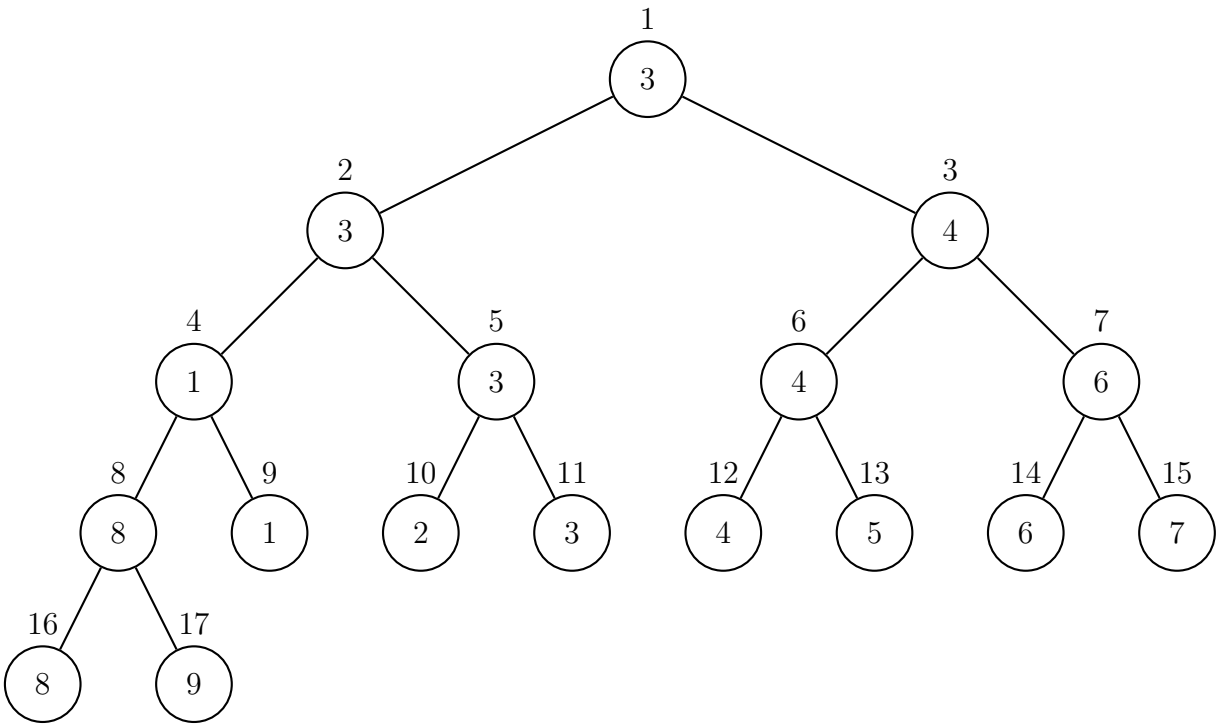


Figura 17: Torneio com 9 elementos em que 3 é o elemento com valor máximo.

Inicialmente o vetor começa com os índices dos elementos ocupando as últimas posições e construímos o torneio de acordo com o valor de cada elemento no instante $t = 0$, ou seja, com o valor x_0 de cada elemento.

Uma vez de posse do torneio montado, construímos um certificado para cada elemento no torneio. O i-ésimo certificado, que se refere ao par formado pelo i-ésimo elemento da entrada e quem o venceu na última partida que disputou, consiste no instante de tempo em que o i-ésimo elemento passará a ter um valor maior que o valor do elemento que o venceu anteriormente, se esse instante for maior que o instante atual. Do contrário, o certificado consiste em $+\infty$.

É importante observar que o elemento que se encontra na primeira posição do torneio não é vencido por ninguém no instante *now*. Dessa forma, sendo i o elemento que ocupa a primeira posição do torneio, associamos ao i-ésimo certificado a chave $+\infty$.

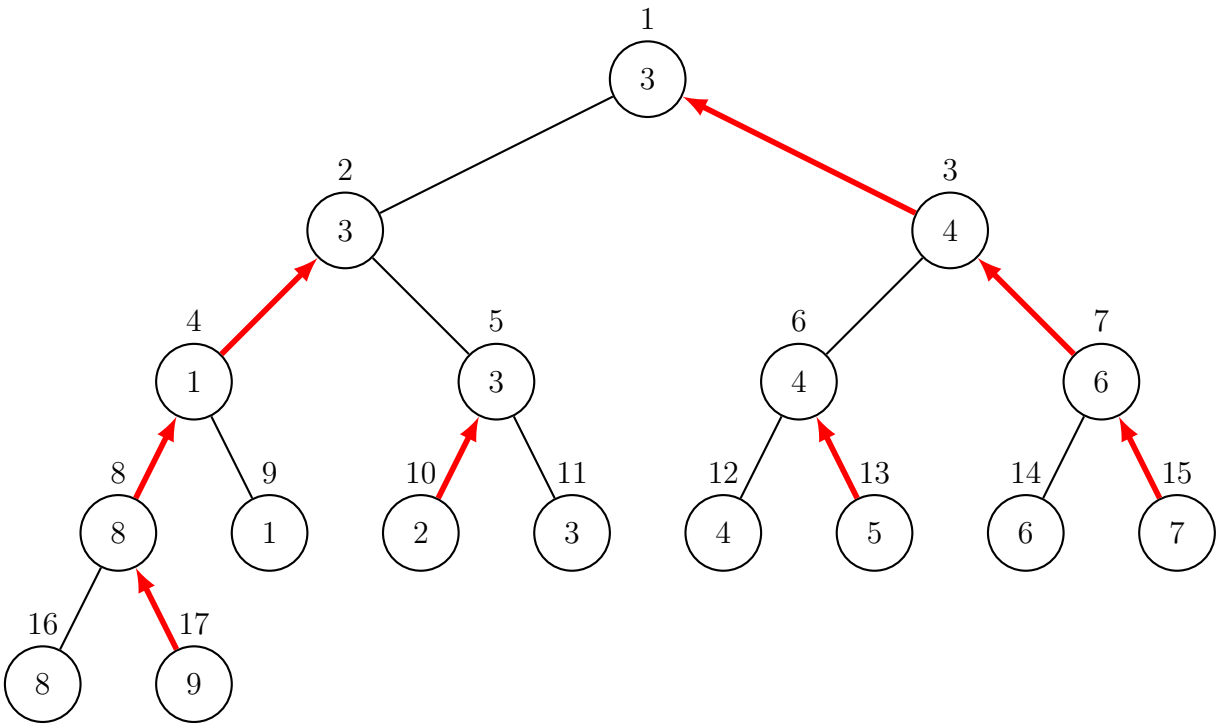


Figura 18: Torneio com 9 elementos e os certificados visualmente representados pelas setas vermelhas mais grossas. O certificado correspondente ao elemento 3 terá a chave $+\infty$.

Esses n certificados são colocados em uma fila de prioridade, com o prazo de validade como chave. Estamos interessados nos certificados com menor prazo de validade.

Para descrever a implementação das três operações, precisamos estabelecer o nome das novas variáveis usadas. São elas:

1. *tourn*: vetor, de $2n-1$ posições, com os índices dos n elementos formando um torneio de acordo com o seu valor no instante *now*;

2. *cert*: vetor com os certificados; $cert[i]$ guarda o certificado entre o elemento i e quem o venceu na última partida que disputou, para $1 \leq i \leq n$;

3. $indT$: vetor de n posições; $indT[i]$ guarda a posição em *tourn* em que i perde uma partida, com $1 \leq i \leq n$. Se i não perde nenhuma partida, $indT[i]$ é igual a -1.

A interface da fila de prioridade que utilizaremos não se altera.

Na implementação da operação EVENT, utilizaremos a rotina UPDATE(i) que calcula a nova validade t do elemento j que se encontra na i-ésima posição de *tourn*, isto é, $j = tourn[i]$ certificado, se $1 \leq i \leq 2n-1$, e chama a rotina UPDATEPQ(Q, i, t).

Algoritmo 3.16: Função UPDATE.

```
1: function UPDATE(i)
2:     if i = 1 :
3:         t ← ∞
4:     if 1 < i ≤ 2n − 1 :
5:         t ← EXPIRE(tourn[⌊i/2⌋], tourn[i])
6:     j ← tourn[i]
7:     UPDATEPQ(Q, j, t)
```

Algoritmo 3.17: Função EVENT.

```
1: function EVENT()
2:     i ← MINPQ(Q)
3:     while cert[i] = now :
4:         j ← indT[i]
5:         k ← 2 · ⌊j/2⌋ + ((j + 1) mod 2)                    ▷ adversário
6:         while j > 1 and VALUE(j) ≥ VALUE(k) :
7:             tourn[⌊j/2⌋] ← tourn[j]
8:             indT[tourn[k]] ← k
9:             UPDATE(k)
10:            j ← ⌊j/2⌋
11:            k ← 2 · ⌊j/2⌋ + ((j + 1) mod 2)
12:        indT[tourn[j]] ← j
13:        UPDATE(j)
14:        i ← MINPQ(Q)
   ▷ VALUE(j) retorna speed[j] · now+ x_0[j]
```

No trecho das linhas 5 - 11 do código 3.17, o resultado da partida entre o elemento j e seu adversário que se encontra na posição k de *tourn* é recalculado, e o certificado correspondente é atualizado. Caso o resultado da partida tenha sido alterado, a verificação se propaga para o nível de cima.

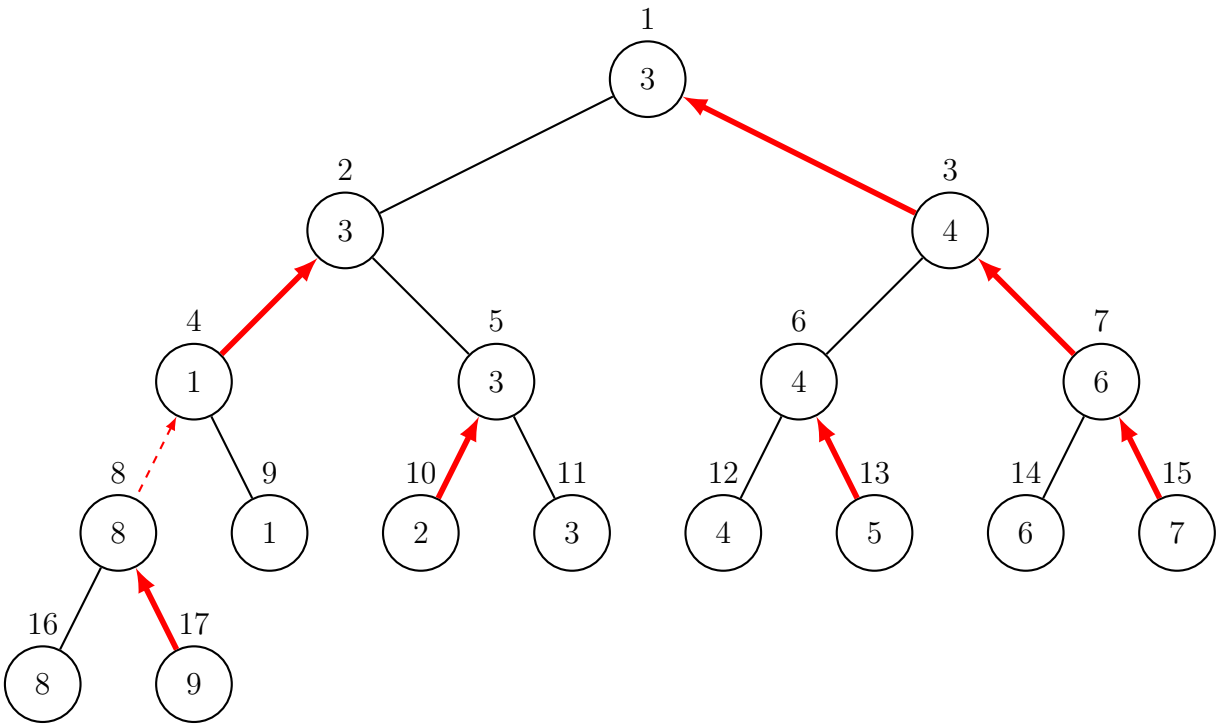


Figura 19: cert[8] expirou.

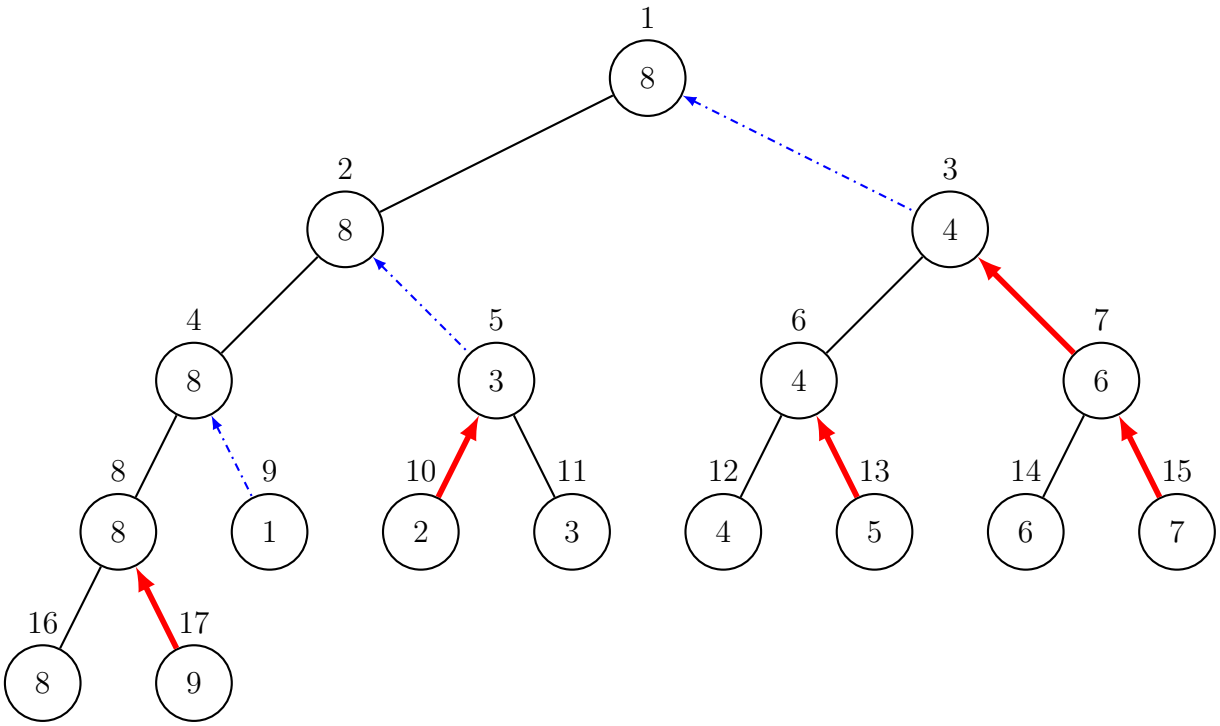


Figura 20: Após *cert*[8] ter expirado, o elemento 8 passou a vencer o elemento 1 e *tourn*[4] foi atualizado, o elemento 8 também venceu dos elementos 3 e 4 e *tourn*[2] e *tourn*[1] foram atualizados. Os certificados *cert*[1], *cert*[3] e *cert*[4], indicados pelas setas azuis, também foram atualizados

A operação QUERY_MAX() consiste em devolver $tourn[1]$, enquanto que a operação CHANGE(j, v) consiste em alterar a posição $x_0[j]$ para $x_0[j] + (speed[j] - v) \cdot now$, a posição $speed[j]$ para v e recalcular os eventuais certificados de que j participa. Para tanto, a partir da posição i em que j se encontra no vetor $tourn$, podemos recalcular $cert[j]$ e então continuamos visitando as partidas em que j participou para atualizar os certificados daqueles que perderam de j, acionando a rotina UPDATEPQ para fazer os devidos acertos em Q correspondentes a estas modificações.

Algoritmo 3.18: Função QUERY_MAX.

```
1: function QUERY_MAX()
2:     return tourn[1]
```

Algoritmo 3.19: Função CHANGE.

```
1: function CHANGE(j, v)
2:     x_0[j] ← x_0[j] + (speed[j] − v) · now;
3:     speed[j] ← v
4:     i ← indT[j]
5:     UPDATE(i)
6:     while i < n :
7:         if tourn[i] = tourn[2i] :
8:             i ← 2i
9:         else
10:            i ← 2i + 1
11:        k ← 2 · ⌊i/2⌋ + ((i + 1) mod 2)                    ▷ adversário
12:        UPDATE(k)
```

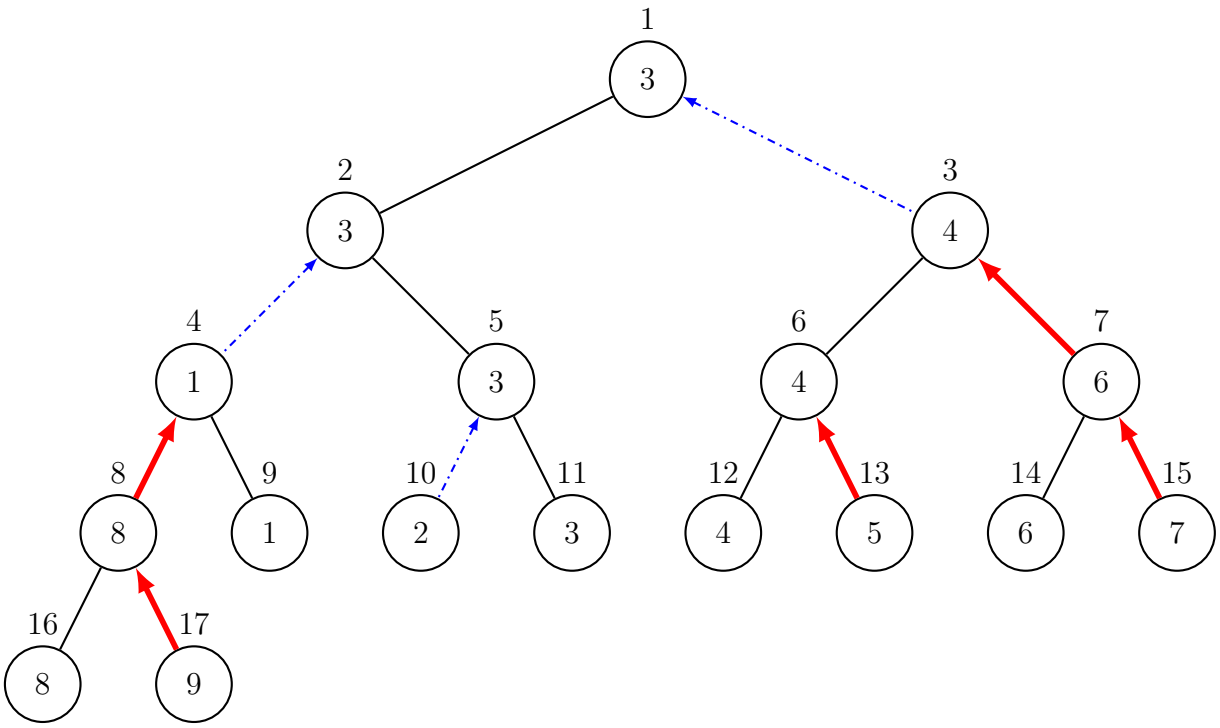


Figura 21: Após atualizar a velocidade do elemento 3, *cert*[2], *cert*[1] e *cert*[4] foram atualizados porque disputaram partidas com 3.

3.2.1 Inserção e remoção em torneio

Assim como fizemos na seção 2.2, além das operações ADVANCE(t), CHANGE(j, v) e QUERY_MAX() poderíamos querer dar suporte a operações como:

- INSERT(v, x_t) → insere um elemento com velocidade v e valor x_t no instante *now*;
- DELETE(i) → remove o elemento i no instante *now*.

Agora, diferentemente da seção 2.2, não utilizaremos uma nova estrutura para dar suporte à essas operações, pois um torneio já suporta operações como inserção e remoção de elementos em tempo logarítmico. Porém, da maneira como se encontra a interface, poderíamos ter problemas como espaços de memória ociosos após várias remoções ou um gasto elevado de tempo redimensionando vetores para que suportem a inserção de novos elementos. Dessa forma, descreveremos a seguir alterações a serem feitas na interface para evitar problemas como os citados.

Inicialmente o vetor que guarda o torneio começa com os elementos ocupando as suas últimas posições e construímos o torneio de acordo com o valor de cada elemento no instante $t = 0$.

Uma vez de posse do torneio montado, construímos um certificado para cada elemento no torneio. Agora os certificados não serão mais mantidos em um vetor; serão mantidos

junto dos elementos para facilitar a inserção e remoção de certificados, já que estas vêm junto com a inserção e remoção de elementos. O certificado de um elemento e se refere à relação estabelecida entre o elemento e e o elemento k, que é o elemento que venceu e na última partida que e disputou, consiste no instante de tempo em que o elemento e passará a ter um valor maior que o valor do elemento k, se esse instante for maior que o instante atual. Do contrário, o certificado consiste em $+\infty$.

Note que o elemento que está na primeira posição do torneio não é vencido por ninguém no instante *now*. Portanto, daremos o valor $+\infty$ para o seu certificado.

Esses n certificados serão colocados em uma fila de prioridade, com o prazo de validade como chave. O certificado com menor prazo de validade estará ocupando a primeira posição da fila. Na verdade, como os certificados estarão diretamente ligados aos elementos, colocaremos os elementos na fila de prioridade. Além disso, a interface da fila de prioridade passará a suportar as operações INSERTPQ(Q, e) e DELETEPQ(Q, e).

Para descrever as implementações das operações, vamos estabelecer os nomes dos objetos, variáveis e rotinas auxiliares utilizados:

1. n: número de elementos no instante *now*;
2. *element*: objeto com os seguintes atributos:
   (a) *id*: atributo para identificar o objeto. Daqui em diante, usaremos elemento i para se referir ao elemento cujo *id* é i;
   (b) *speed*: a velocidade do elemento;
   (c) x_0: é o valor que o elemento possuía no instante $t = 0$;
   (d) *cert*: o tempo de validade do certificado do elemento;
   (e) *pqpos*: atributo que aponta para a posição do elemento na fila de prioridade;
   (f) *lastMatch*: atributo que aponta para a posição do vetor *tourn* em que o elemento disputou sua última partida.
3. *tourn*: vetor, de $2n-1$ posições, que guarda apontadores para os elementos formando um torneio de acordo com seus valores no instante *now*;
4. Q: fila de prioridade que contém os elementos, o elemento com certificado de menor valor estará a frente da fila;
5. INSERTTOURN$(e) \rightarrow$ insere e, que é um elemento, no torneio e atualiza os certificados necessários no processo;
6. DELETETOURN$(e) \rightarrow$ remove e, que é um elemento, do torneio e atualiza os certificados necessários no processo.

Para a implementação das operações CHANGE(j, v) e DELETE(i) precisamos de alguma maneira recuperar um elemento baseado no seu *id*. Para tal, podemos utilizar estruturas como uma árvore de busca binária balanceada ou uma tabela de dispersão. A seguir estão três operações que nos ajudarão a recuperar os elementos:

1. GETOBJECT$(i) \rightarrow$ retorna o elemento i;
2. INSERTOBJECT$(e) \rightarrow$ insere e, que é um elemento, na estrutura;
3. DELETEOBJECT$(e) \rightarrow$ remove e, que é um elemento, da estrutura.

Para permitir a inserção e remoção de certificados, a interface da fila de prioridade será reformulada, contando com duas operações extras:

1. INSERTPQ$(Q, e) \rightarrow$ insere e na fila de prioridade Q;
2. DELETEPQ$(Q, e) \rightarrow$ remove e da fila de prioridade Q;
3. UPDATEPQ$(Q, e, t) \rightarrow$ muda o valor do certificado de e para t e atualiza a fila de prioridade Q;
4. MINPQ$(Q) \rightarrow$ devolve o elemento com o certificado de menor valor da fila de prioridade Q.

A operação UPDATEPQ(Q, e, t) pode ser implementada em tempo logarítmico graças ao atributo *pqpos* dos elementos.

Um evento está associado a um certificado (e, t) que expira no instante t. Na implementação da operação EVENT, utilizaremos a rotina UPDATE(e), do algoritmo 3.20, que calcula a nova validade t do certificado do elemento e, e chama a rotina UPDATEPQ(Q, e, t).

Algoritmo 3.20: Função UPDATE.

```
1: function UPDATE(e)
2:     if e ≠ NULL :
3:         e' ← tourn[(e.lastMatch)/2]
4:         t ← EXPIRE(e, e')
5:         UPDATEPQ(Q, e, t)
```

▷ Em expire(e, e'), e' pode ser nulo e nesse caso o retorno é $+\infty$.

No trecho das linhas 5 - 11 do código 3.21, o resultado da partida entre o elemento j e seu adversário que se encontra na posição k de *tourn* é recalculado, e o certificado correspondente é atualizado. Caso o resultado da partida tenha sido alterado, a verificação se propaga para o nível de cima.

Algoritmo 3.21: Função EVENT.

```
1:  function EVENT()
2:      e ← MINPQ(Q)
3:      while e.cert = now :
4:          j ← e.lastMatch
5:          k ← 2 · ⌊j/2⌋ + ((j + 1) mod 2)                    ▷ adversário
6:          while j > 1 and COMPARE(j, k) :
7:              tourn[⌊j/2⌋] ← tourn[j]
8:              tourn[k].lastMatch ← k
9:              UPDATE(tourn[k])
10:             j ← ⌊j/2⌋
11:             k ← 2 · ⌊j/2⌋ + ((j + 1) mod 2)                ▷ adversário
12:         tourn[j].lastMatch ← j
13:         UPDATE(tourn[j])
14:         e ← MINPQ(Q)
```

▷ COMPARE(i, j) retorna se o valor de i é maior que o valor de j.

A operação QUERY_MAX(), no algoritmo 3.22, consiste em devolver $tourn[1]$, enquanto que a operação CHANGE(j, v) consiste em recuperar o elemento j, alterar seu atributo x_0 para $x_0 + (speed - v) \cdot now$, $speed$ para v e recalcular os eventuais certificados de que j participa. Para tanto, a partir da posição i mais alta em que j se encontra no vetor $tourn$, podemos recalcular $cert[j]$ e então continuamos visitando as partidas em que j participou para atualizar os certificados daqueles que perderam de j, acionando a rotina UPDATEPQ para fazer os devidos acertos em Q correspondentes a estas modificações.

Algoritmo 3.22: Função QUERY_MAX.

```
1:  function QUERY_MAX()
2:      return tourn[1]
```

A operação INSERT(v, x_t) consiste em criar um novo elemento, inicializando seus atributos com os devidos valores, inseri-lo no torneio e na estrutura que usamos para recuperá-lo, calcular o seu certificado e inseri-lo na fila de prioridade. Uma importante observação é que se $now \neq 0$, então $x_t \neq x_0$. Para calcular x_0, podemos utilizar a relação $x_t = now \cdot speed + x_0 \Rightarrow x_0 = x_t - speed \cdot now$.

A operação DELETE(i) consiste em recuperar o elemento i, removê-lo da fila de prioridade, do torneio e da estrutura que usamos para recuperá-lo depois.

A função auxiliar INSERTTOURN(e), do algoritmo 3.24, consiste de criar uma nova partida, usando o elemento que está na posição n para completar a partida, depois subimos

Algoritmo 3.23: Função CHANGE.

```
1: function CHANGE(j, v)
2:     e ← GETOBJECT(j)
3:     e.x_0 ← e.x_0 + (e.speed − v) · now;
4:     e.speed ← v
5:     i ← e.lastMatch
6:     UPDATE(e)
7:     while i < n :
8:         if tourn[i] = tourn[2i] :
9:             i ← 2i
10:        else
11:            i ← 2i + 1
12:        k ← 2 · ⌊i/2⌋ + ((i + 1) mod 2)                    ▷ adversário
13:        UPDATE(tourn[k])
```

Algoritmo 3.24: Função INSERT.

```
1: function INSERT(v, x_t)
2:     e.speed ← v
3:     e.x_0 ← x_t − now · v
4:     root ← INSERTOBJECT(root, e)
5:     INSERTTOURN(e)
6:     NEWCERT(e)
7:     INSERTPQ(Q, e)
```

Algoritmo 3.25: Função DELETE.

```
1: function DELETE(i)
2:     e ← GETOBJECT(i)
3:     DELETEPQ(Q, e)
4:     DELETETOURN(e)
5:     DELETEOBJECT(e)
```

para o nível de cima no torneio, corrigindo os vencedores das partidas e atualizando os certificados correspondentes. O certificado do elemento inserido não será calculado nessa função, será calculado posteriormente. Na implementação do algoritmo 3.26, RESIZE() checa se *tourn* é capaz de suportar a inserção de novos elementos e, se não for, redimensiona *tourn*.

Algoritmo 3.26: Função INSERTTOURN.

```
1:  function INSERTTOURN(e)
2:      RESIZE()
3:      n ← n + 1
4:      i ← 2n − 1
5:      tourn[i] ← e
6:      tourn[i − 1] ← tourn[⌊i/2⌋]
7:      k ← i − 1
8:      while i > 1  and COMPARE(i, k) :
9:          tourn[⌊i/2⌋] ← tourn[i]
10:         tourn[k].lastMatch ← k
11:         UPDATE(tourn[k])
12:         i ← ⌊i/2⌋
13:         k ← 2 · ⌊i/2⌋ + ((i + 1)  mod 2)                ▷ adversário
14:     tourn[1].lastMatch ← 1
    ▷ COMPARE(i, j) retorna se o valor de i é maior que o valor de j.
```

A função auxiliar DELETETOURN(e), do algoritmo 3.25, consiste de usar o perdedor da partida travada entre os elementos que estão nas duas últimas posições de *tourn* para substituir o elemento e. Além disso, desfazemos essa partida para que os n elementos continuem a ocupar as $2n - 1$ primeiras posições do torneio após a remoção de e. O perdedor substituíra o elemento e na posição da primeira partida de que e participou. Todas as partidas desde essa posição, se propagando para o nível de cima no caminho até a primeira posição, serão recalculadas com os devidos certificados atualizados. Essa propagação até a primeira posição é importante para que não hajam resquícios do elemento deletado no torneio, veja o exemplo da figura 24. Na implementação, no algoritmo 3.27, SUBSTITUTE(e) faz a substituição citada retornando a posição da primeira partida de que e participou.

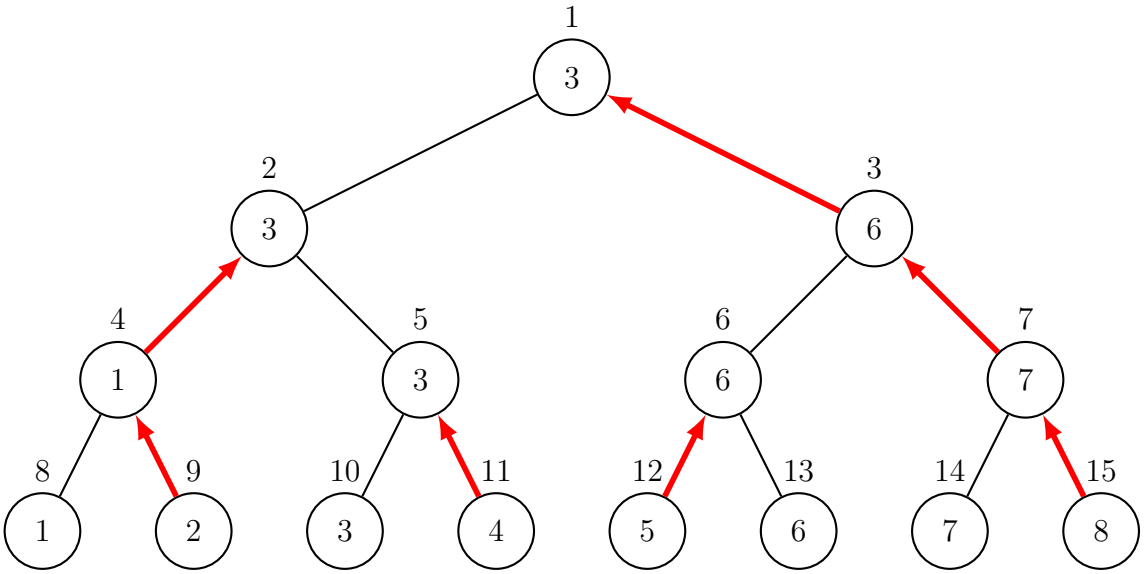


Figura 22: A figura mostra um torneio com 8 elementos e seus certificados representados pelas setas vermelhas, mais grossas; cada posição guarda um apontador para um elemento, mas na figura apenas os *id*'s são mostrados.

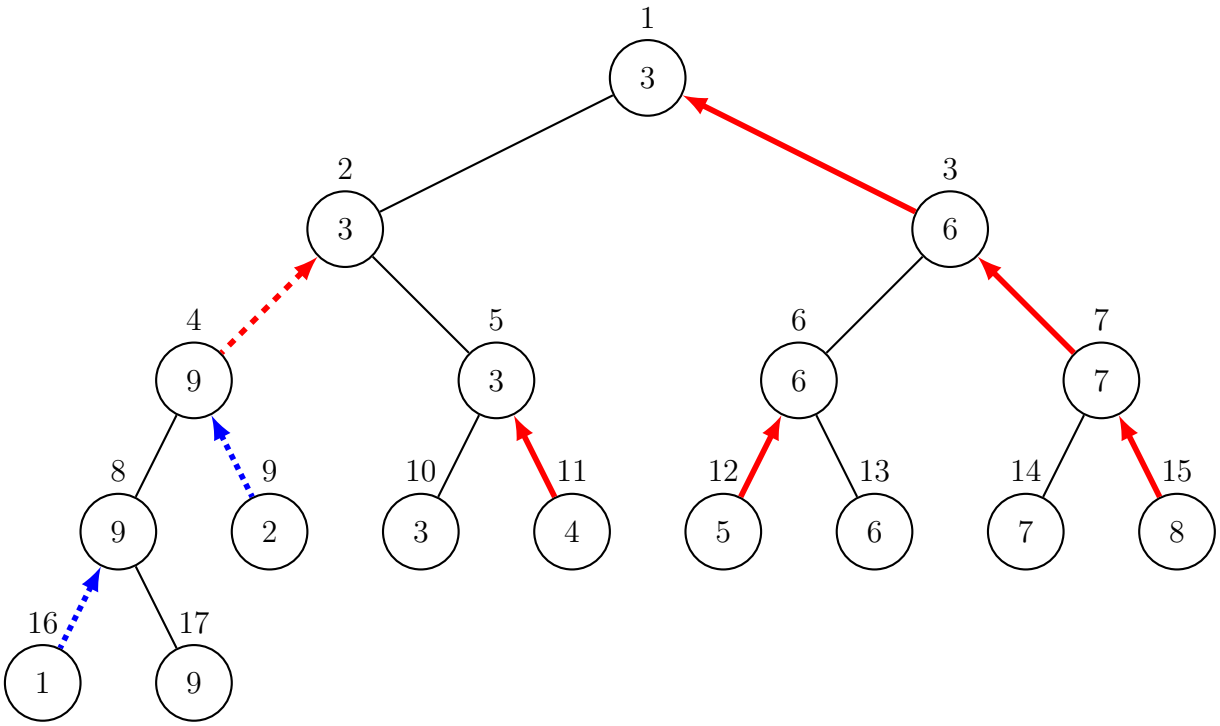


Figura 23: A figura mostra o torneio da figura 22 após a inserção de um elemento com *id* 9. A seta tracejada representa o certificado criado e as setas azuis pontilhadas representam os certificados atualizados.

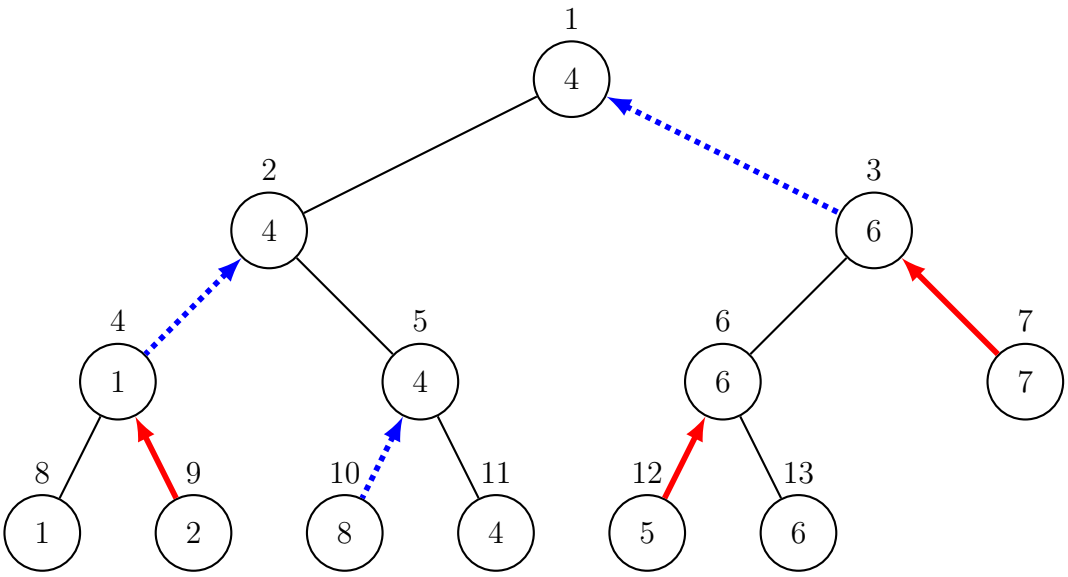


Figura 24: A figura mostra o torneio da figura 22 após a remoção do elemento com *id* 3. O elemento 8 que era o perdedor da partida entre os elementos que ocupam a última posição foi usado para substituir a posição que o elemento 3 ocupava. Todas partidas até o topo foram recalculadas com os devidos certificados atualizados. As setas azuis pontilhadas representam os certificados atualizados.

Algoritmo 3.27: Função DELETETOURN.

```
1:  function DELETETOURN(e)
2:      i ← SUBSTITUTE(e)
3:      k ← 2 · ⌊i/2⌋ + ((i + 1) mod 2)
4:      while i > 1 :
5:          if COMPARE(k, i) :
6:              i ↔ k
7:          tourn[⌊i/2⌋] ← tourn[i]
8:          tourn[k].lastMatch ← k
9:          UPDATE(tourn[k])
10:         i ← ⌊i/2⌋
11:         k ← 2 · ⌊i/2⌋ + ((i + 1) mod 2)          ▷ adversário
12:     tourn[1].lastMatch ← 1
13:     UPDATE(tourn[1])
```

▷ COMPARE(i, j) retorna se o valor de i é maior que o valor de j.

4 Par mais próximo

Considere o seguinte problema cinético. São dados n pontos movendo-se linearmente no plano. Cada ponto é representado por um par $(s_0, \vec{v})$ onde $s_0 = (x_0, y_0)$ é a sua posição inicial e $\vec{v} = (v_x, v_y)$ um vetor velocidade. A posição de um determinado ponto p, num instante arbitrário $t \geq 0$, é $s_p = (x_p, y_p) = (x_0, y_0) \ + \ t \cdot \vec{v}$. Queremos saber o par (p, q) cuja distância $d(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$ é mínima, num instante arbitrário $t \geq 0$.

Por exemplo, se tivermos 5 pontos na coleção, representados na figura 25: $((1, 0), (2, 1))$, $((5, -1), (-1, 2))$, $((0, 2), (1, -1))$, $((3, 2), (1, -2))$ e $((3, 1), (-1, 0))$.

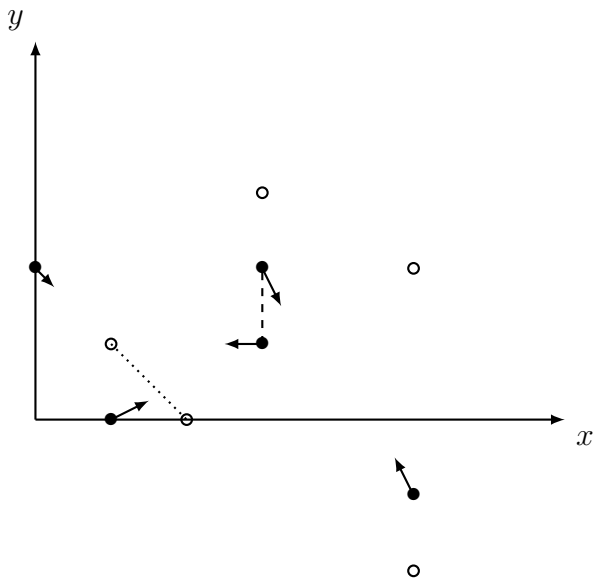


Figura 25: Os pontos preenchidos em preto representam a coleção no instante $t = 0$. Os pontos não preenchidos representam a coleção no instante $t = 2$. As setas representam a direção e sentido do vetor velocidade de cada ponto. A linha tracejada representa a distância entre o par mais próximo no instante $t = 0$, enquanto a linha pontilhada representa a distância entre o par mais próximo no instante $t = 2$.

Queremos dar suporte às seguintes operações:

- ADVANCE(t) $\rightarrow$ avança o tempo corrente para t;
- CHANGE$(j, \vec{v})$ $\rightarrow$ altera a velocidade do ponto j para $\vec{v}$;
- QUERY_CLOSEST$()$ $\rightarrow$ devolve os pontos que formam o par mais próximo no instante atual.

4.1 Algoritmo Estático

O algoritmo que será aqui apresentado foi proposto por Basch, Guibas e Hershberger e admite uma boa cinetização, usando a ideia de linha de varredura.

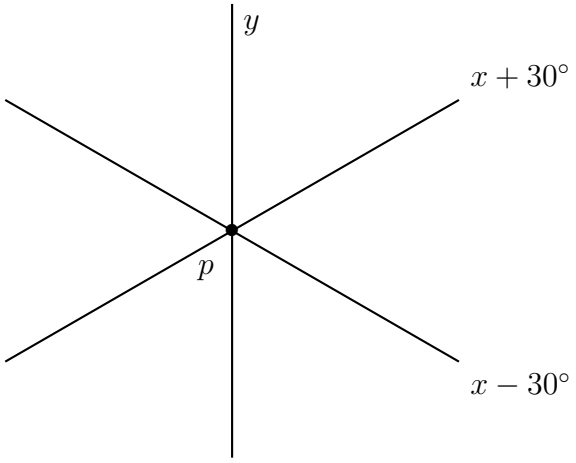


Figura 26: A reta paralela ao eixo y que passa por p e as retas $x \pm 30°$.

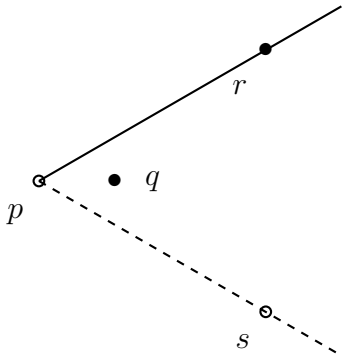


Figura 27: Os pontos q e r pertencem a $Dom(p)$, mas o ponto s não.

O algoritmo é baseado na ideia de dividir o plano, para cada ponto, em seis cones iguais. Os cones são delimitados pela reta paralela ao eixo y que passa pelo ponto e pelas retas $x \pm 30°$, isto é, as retas que passam pelo ponto e formam $\pm 30°$ com o eixo x como mostra a figura 26.

Tendo dividido o plano em cones, a ideia é achar o ponto mais próximo de p dentro de cada um desses cones. Se assim o fizermos para todos os pontos, um desses pares possui a menor distância entre si e será o par mais próximo que buscamos.

Se (p, q) formam um par mais próximo, então (q, p) também serão um par mais próximo, na verdade, serão o mesmo par. Dessa maneira, não precisamos dos seis cones para buscar os pares, somente de três deles. Para uma varredura da direita para a esquerda, apenas buscaremos os pares mais próximos nos três cones à direita de p.

Vamos começar analisando o cone cujo eixo central é paralelo ao eixo x. Chamaremos esse cone de *dominância de p* e o representaremos por $Dom(p)$. Consideraremos que um ponto em cima da linha $x + 30°$ pertence a $Dom(p)$ e um ponto em cima de $x - 30°$ não pertence a $Dom(p)$ como mostra a figura 27. O mesmo algoritmo poderá ser aplicado aos outros dois cones se rotacionarmos o sistema de coordenadas $\pm 60°$.

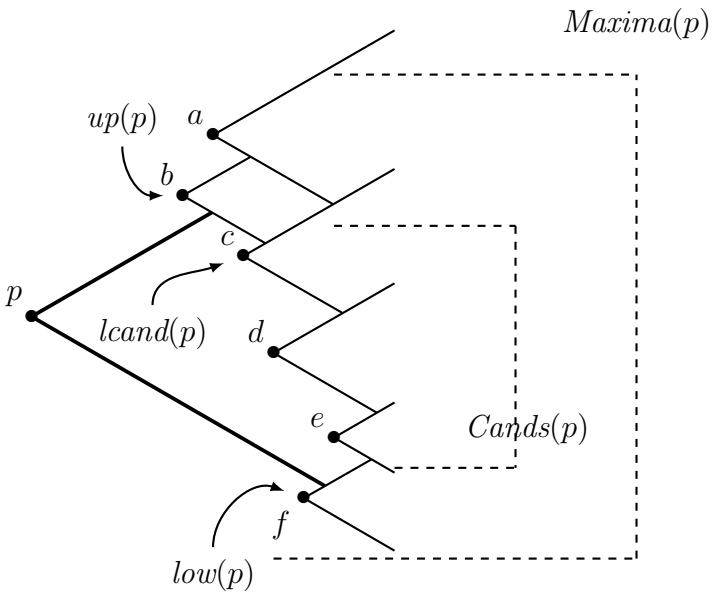


Figura 28: Os pontos c, d e e pertencem a $Cands(p)$, e todos os pontos exceto p pertencem a $Maxima(p)$. O ponto b é $up(p)$ e o ponto f é $low(p)$. O ponto c é $lcand(p)$.

Definiremos como $Maxima(p)$ o conjunto dos pontos à direita de p que não pertencem a *dominância* de nenhum ponto à direita de p. Isso nos permite definir o conjunto de *candidatos* de p representado por $Cands(p)$: $Cands(p) = Dom(p) \cap Maxima(p)$, ou seja, os *candidatos* de p são aqueles pontos à direita de p que não pertencem a *dominância* de nenhum ponto à direita de p e pertencem a *dominância* de p. Chamaremos o ponto de *Maxima* de menor ordenada que está acima de $Dom(p)$ de $up(p)$ e chamaremos o ponto de *Maxima* de maior ordenada que está abaixo de $Dom(p)$ de $low(p)$. Caso não existam tais pontos $up(p)$ e $low(p)$ são *NULL*. Os pontos estritamente entre $low(p)$ e $up(p)$ são justamente os de $Cands(p)$. Dentre os *candidatos* de p, chamaremos o ponto com menor coordenada x de $lcand(p)$. A figura 28 mostra um exemplo dos conjuntos definidos.

Consideraremos apenas os pares $(p, lcand(p))$ como possíveis candidatos a par mais próximo. Caso, para algum p, mais de um ponto atenda à condição de ser $lcand(p)$ poderemos escolher qualquer um deles como $lcand(p)$, pois, em um caso que há mais de um possível $lcand(p)$, esses pontos formarão um par mais próximo entre si do que o par $(p, lcand(p))$, como por exemplo na figura 29.

O algoritmo 4.28 descreve a sequência de operações a serem feitas para achar o par mais próximo em alguma das ordens $(-60°, 0°, 60°)$ representadas pelo ângulo θ, dado em radianos. Antes da rotina ser chamada os pontos devem ser ordenados de acordo com a sua coordenada x. No algoritmo, a e b são os pontos que representam o par mais próximo. Se p ou q são nulos, $d(p, q)$ retorna $+\infty$.

A cada iteração do algoritmo 4.28, *Maxima* é igual a $Maxima(p)$. Na nossa implementação, *Maxima* estará armazenado em uma árvore binária de busca, mais especifi-

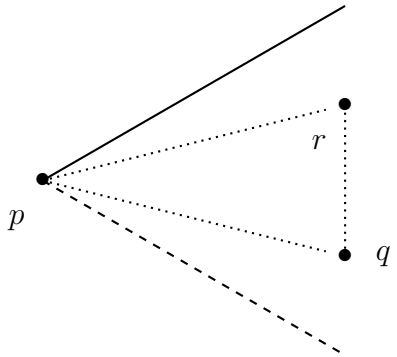


Figura 29: A distância de r até q é menor do que a distância p até r e do que a distância p até q.

camente em uma *splay tree* cuja chave é a coordenada y dos pontos. Com isso, podemos buscar por $up(p)$ e $low(p)$ em tempo logarítmico, bem como podemos retirar $Cands(p)$ de $Maxima$ em tempo logarítmico, isto é, atualizar $Maxima$ de maneira que $Maxima = Maxima \setminus Cands(p)$.

Algoritmo 4.28: Função CLOSEST_PAIR(p, n, θ).

```
1:  function CLOSEST_PAIR(p, n, θ)
2:      (a, b) ← (NULL, NULL)
3:      HEAPSORT(p, n, θ)                          ▷ p.x[1] > ··· > p.x[n]
4:      Maxima ← ∅
5:      for i ← 1 to n :
6:          Cands ← Maxima ∩ Dom(p_i)
7:          Maxima ← (Maxima \ Cands) ∪ {p_i}
8:          lcand ← MIN_X(Cands)
9:          if d(p_i, lcand) < d(a, b) :
10:             (a, b) ← (p_i, lcand)
11:     return (a, b)
```

Para descrever a implementação do algoritmo, já considerando as versões rotacionadas dele, iremos antes precisar estabelecer os nomes das variáveis e rotinas auxiliares utilizadas. São elas:

1. n: o número de pontos dados;
2. *point*: um ponto com os seguintes atributos:
   (a) x: coordenada x do ponto;
   (b) y: coordenada y do ponto.

3. *root*: raiz da splay tree;

4. *node*: objeto que compõe a árvore de busca binária, atributos:

   (a) *left*: aponta para a raiz da subárvore esquerda do nó. A subárvore esquerda é composta apenas por pontos que possuem *key* com menor ordenada que a *key* do nó;

   (b) *right*: aponta para a raiz da subárvore direita do nó. A subárvore direita é composta apenas por pontos que possuem *key* com ordenada maior ou igual que a *key* do nó;

   (c) *parent*: aponta para o nó que é pai deste nó;

   (d) *key*: aponta para um ponto.

5. *angle*: ângulo de rotação do sistema de coordenadas;

6. *points*: vetor de n posições que guarda os pontos;

7. GETX(p) $\rightarrow$ retorna a coordenada x de um ponto p baseada no ângulo de rotação *angle*;

8. GETY(p) $\rightarrow$ retorna a coordenada y de um ponto p baseada no ângulo de rotação *angle*;

9. HEAPSORT() $\rightarrow$ ordena o vetor *points*, utilizando o algoritmo *heapsort*, de acordo com a coordenada x de cada ponto cujo valor é retornado pela rotina GETX(p).

Para um ponto (r, ϕ) em coordenadas polares $x = rcos(\phi)$ e $y = rsen(\phi)$.

Rotacionar o sistema de coordenadas por θ é o mesmo que transformar ϕ em $\phi - \theta$, veja a figura 30. Isso significa que agora as novas coordenadas são descritas como:

$$x^* = rcos(\phi - \theta) = rcos(\phi)cos(\theta) + rsen(\phi)sen(\theta) = xcos(\theta) + ysen(\theta)$$
$$y^* = rsen(\phi - \theta) = rsen(\phi)cos(\theta) - rcos(\phi)sen(\theta) = ycos(\theta) - xsen(\theta)$$

Os valores x^* e y^* são os valores, respectivamente, retornados por GETX(p) e GETY(p) para $\theta = angle$.

A interface da *splay tree*, cuja chave é a coordenada y do ponto, contará com as seguintes operações além das usuais:

1. SUCCESSOR(p) $\rightarrow$ busca pelo nó cuja chave é $up(p)$ na *splay tree*. Esse nó corresponde ao sucessor de p na árvore;

2. PREDECESSOR(p) $\rightarrow$ busca pelo nó cuja chave é $low(p)$ na *splay tree*. Esse nó corresponde ao predecessor de p na árvore;

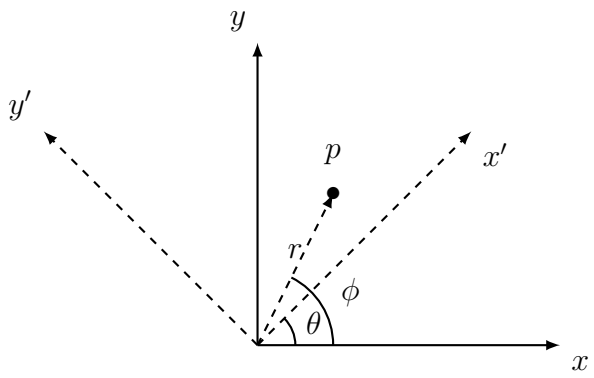


Figura 30: O ponto p está numa inclinação de $\phi - \theta$ graus em relação a reta que passa pela origem e por x'.

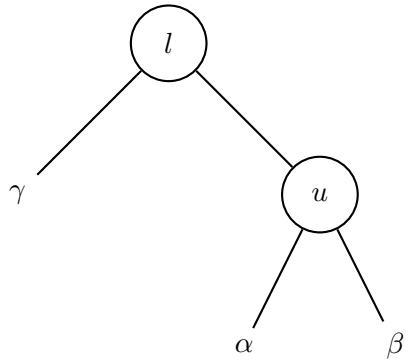


Figura 31: Na figura, l é $low(p)$ e u é $up(p)$. A subárvore α contém todos os pontos que estão entre $low(p)$ e $up(p)$ e, portanto, corresponde ao conjunto $Cands(p)$.

3. LCAND(p) $\rightarrow$ calcula $Cands(p)$, remove da *splay tree* e determina $lcand(p)$, que pode ser $NULL$;

No algoritmo 4.29 e no algoritmo 4.30 a rotina checkLine(p, q, θ) retorna se o ponto q está à esquerda, sobre ou à direita da reta r. A reta r é a reta que passa por p e faz um ângulo de θ radianos com o eixo x. Para q à esquerda de r o retorno é 1, para q sobre r o retorno é 0 e para q à direita de r, o retorno é -1.

O algoritmo 4.31 implementa a função LCAND(p). Sabemos que $lcand(p)$ é o elemento de $Cands(p)$ com menor coordenada x e que $Cands(p)$ é limitado por $low(p)$ e $up(p)$. A ideia para retirar $Cands(p)$ da árvore é reorganizá-la de modo que $low(p)$ seja a raiz e $up(p)$ o filho direito da raiz, dessa forma $Cands(p)$ é a subárvore esquerda do filho direito da raiz, veja a figura 31. Nem sempre tal configuração é possível, mas o algoritmo tratará dos casos de borda que são quando $low(p)$, ou $up(p)$, ou ambos, não existem. A rotina SPLIT(x) separa a subárvore de raiz x da *splay tree* e retorna a raiz dessa nova árvore. A rotina MIN_X(z) retorna o ponto com menor coordenada x da árvore de raiz z.

O algoritmo 4.32 implementa a função QUERY_CLOSEST que retorna o par (a, b) que

Algoritmo 4.29: Função SUCCESSOR(p).

```
1: function SUCCESSOR(p)
2:     x ← root
3:     up ← NULL
4:     while x ≠ NULL :
5:         y ← x
6:         if CHECKLINE(p, x.key, π/6) = −1 :
7:             x ← x.right
8:         else
9:             up ← x
10:            x ← x.left
11:    if y ≠ NULL :                    ▷ dá SPLAY no último nó visitado
12:        SPLAY(y)
13:    return up
```

Algoritmo 4.30: Função PREDECESSOR(p).

```
1: function PREDECESSOR(p)
2:     x ← root
3:     low ← NULL
4:     while x ≠ NULL :
5:         y ← x
6:         if CHECKLINE(p, x.key, −π/6) ≤ 0 :
7:             x ← x.left
8:         else
9:             low ← x
10:            x ← x.right
11:    if y ≠ NULL :                    ▷ dá SPLAY no último nó visitado
12:        SPLAY(y)
13:    return low
```

Algoritmo 4.31: Função lcand(p).

```
1:  function LCAND(p)
2:      r ← root
3:      low ← PREDECESSOR(p)
4:      if low ≠ NULL :
5:          SPLAY(low)
6:          r ← SPLIT(low.right)
7:      up ← SUCCESSOR(p)
8:      if up ≠ NULL :
9:          SPLAY(up)
10:         r ← SPLIT(up.left)
11:     if up ≠ NULL  and low ≠ NULL :
12:         low.right ← up
13:         up.parent ← low
14:     return MIN_X(r)
```

possui distância mínima em *points*.

Algoritmo 4.32: Função QUERY_CLOSEST.

```
1:  function QUERY_CLOSEST()
2:      (m, n) ← (NULL, NULL)
3:      angle ← −π/3
4:      while angle ≤ π/3 :
5:          HEAPSORT(points, n, θ)                ▷ points.x[1] > ··· > points.x[n]
6:          for i ← 1 to n :
7:              p ← points[i]
8:              lcand ← LCAND(p)
9:              INSERT(p)
10:             if d(p, lcand) < d(a, b) :
11:                 (a, b) ← (p, lcand)
12:         angle ← angle + π/3
13:         CLEARALL()                            ▷ esvazia a splay tree
14:     return (m, n)
```

4.2 Algoritmo Cinético

Para "cinetizar" o algoritmo estático utilizaremos certificados, para assegurar que as nossas estruturas permanecerão corretas. Primeiramente, teremos os certificados das três *listas ordenadas cinéticas*, seção 2.1, que guardarão a ordem dos pontos de acordo com os eixos x, $x + 60°$ e $x - 60°$.

Para garantir qual, dentre os pares $(p, lcand(p))$, é o par mais próximo usaremos um *torneio cinético com inserção e remoção*, seção 3.2.1, com respeito ao mínimo em vez de ao máximo. Temos um total de $3n$ pares, pois consideraremos também os pares $(p, lcand(p))$ em que $lcand(p)$ é nulo e os certificados destes serão $+\infty$.

Também precisaremos manter informação guardada para atualizar com eficiência mudanças provocadas por trocas na ordem dos pontos em relação a um dos três eixos. Por exemplo, uma troca na ordem dos pontos pode acarretar na mudança nos conjuntos $Cands(p)$ e $Cands(q)$. Mudanças nesses conjuntos ocorrerão quando $q = up(p)$, $q = low(p)$ ou $q \in Cands(p)$. Portanto, para que consigamos manter $lcand(p)$ de maneira eficiente cada ponto terá três árvores binárias de busca associadas a ele com os conjuntos $Cands(p)$, $Hits_{up}(p)$ e $Hits_{low}(p)$. A árvore $Cands(p)$ guarda os pontos que pertencem ao conjunto $Cands(p)$ ordenados pela coordenada y. A árvore $Hits_{up}(p)$ guarda os pontos q tais que $up(q) = p$, ordenados pela coordenada x. Similarmente, a árvore $Hits_{low}(p)$ guarda os pontos q tais que $low(q) = p$, ordenados pela coordenada x. Utilizaremos uma árvore $Cands(p)$, $Hits_{up}(p)$ e $Hits_{low}(p)$ para cada um dos eixos, logo, para cada ponto p, haverão nove *splay trees* no total.

Cada uma das três árvores têm sua raiz apontando para o nó p, e cada nó das árvores aponta para o seu nó pai. Na árvore $Cands(p)$ cada nó deve apontar para o descendente que contém o ponto mais à esquerda na ordenação horizontal. Na nossa implementação as árvores serão *splay trees*. Essas estruturas contém toda a informação necessária para que mantenhamos nossas estruturas atualizadas e, consequentemente, o par mais próximo do conjunto.

Na implementação do algoritmo inicialmente inserimos os pontos nas três listas ordenadas. Uma vez que as listas estejam montadas, percorremos os pontos da direita para a esquerda preenchendo as estruturas $Cands(p)$, $Hits_{up}(p)$ e $Hits_{low}(p)$ para cada ponto p e para cada um dos eixos. Esta etapa é feita da mesma forma que foi apresentada na seção sobre o algoritmo estático.

A medida que as estruturas $Cands(p)$ são inicializadas inserimos o par $(p, lcand(p))$ no torneio. Quando todos os pares forem inseridos no torneio realizamos as partidas e calculamos os certificados, o par (p, q) da partida que possuir menor distância é considerado o vencedor.

Todos os certificados são colocados em uma fila de prioridade Q. Os certificados inseridos na fila possuem quatro informações:

- t $\rightarrow$ instante de tempo em que o certificado expira. É utilizado como chave para a fila de prioridade. Desempates são tratados de maneira especial e serão explicados mais adiante;

- p $\rightarrow$ um dos pontos envolvidos no evento representado pelo certificado. Caso seja um certificado de troca na ordenação, p é o ponto mais a direita naquela ordenação;

- q $\rightarrow$ o outro ponto envolvido no evento representado pelo certificado. Caso seja um certificado de troca na ordenação, q é o ponto mais a esquerda naquela ordenação;

- tipo $\rightarrow$ o tipo de evento que o certificado representa. Pode representar uma troca em uma das três ordenações, denominadas por H (horizontal $= 0°$-ordem), U (up $= +60°$-ordem) e D (down $= -60°$-ordem) ou pode representar a vitória do par (p, q) em uma partida do torneio.

Vamos agora falar de um evento em que ocorre uma mudança na ordem horizontal. No primeiro caso, p se encontra à esquerda e abaixo de q, veja a figura 32. O caso em que q está à esquerda de p será tratado de maneira parecida. O algoritmo 4.33 implementa a sequência de operações referentes à esse tipo de evento.

Se p está em $Hits_{up}(q)$, como demonstrado na figura 32, então parte de $Cands(q)$ terá de passar para $Cands(p)$. Para tal, buscamos pelo novo $t = up(p)$ em $Cands(q)$ e chamamos a rotina *splay* para o nó que contém t. Após o *splay*, chamamos um *split*

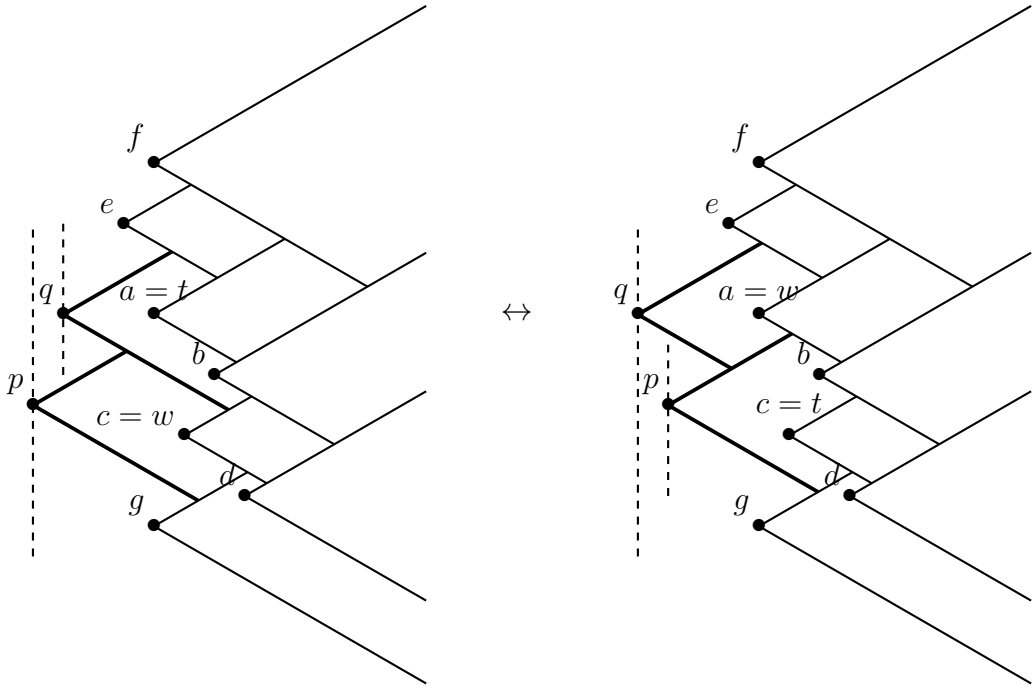


Figura 32: Da esquerda para direita, o caso em que p está em $Hits_{up}(q)$. Da direita para esquerda, o caso em que q está em $Hits_{low}(p)$.

na subárvore esquerda desse nó e unimos ela a $Cands(p)$. Se t não for encontrado em $Cands(q)$, então $t = up(q)$ e todos os pontos de $Cands(q)$ devem ser transferidos para $Cands(p)$. Não podemos esquecer de remover p de $Hits_{up}(q)$ e adicioná-lo em $Hits_{up}(t)$, além de remover q de $Hits_{low}(w)$ e adicioná-lo em $Hits_{low}(p)$. Se p não está em $Hits_{up}(q)$, então não haverão mudanças, veja a figura 33.

Similarmente, se q está em $Hits_{low}(p)$, como demonstrado na figura 32, parte de $Cands(p)$ passará para $Cands(q)$. Para realizar tal operação, buscamos pelo novo $t = low(q)$ em $Cands(p)$, damos um *splay* no nó que o contém, separamos a subárvore direita desse nó e unimos ela à $Cands(q)$. Se t não for encontrado em $Cands(p)$, então $t = low(p)$ e todos os pontos de $Cands(p)$ devem ser passados para $Cands(q)$. Devemos também remover q de $Hits_{low}(p)$ e inseri-lo em $Hits_{low}(t)$, além de remover p de $Hits_{up}(w)$ e adicioná-lo em $Hits_{up}(q)$. Se q não está em $Hits_{low}(p)$, então não haverão mudanças, veja a figura 33.

No caso de um evento em que ocorre uma mudança na 60°-ordem, que é a ordem dos pontos projetados no eixo $x + 60°$, vamos assumir que p é o ponto que está à esquerda e acima de q. O evento pode provocar a entrada ou saída do ponto q de $Cands(p)$, veja a figura 34. O algoritmo 4.34 implementa a sequência de operações referentes a este evento.

Se p está em $Hits_{low}(q)$, ou seja, q está entrando em $Dom(p)$ como demonstrado na figura 34 da esquerda para direita, então a troca na 60°-ordem afetará o ponto v tal que q está em $Cands(v)$. Achamos esse ponto subindo em $Cands(v)$, a partir do nó que contém

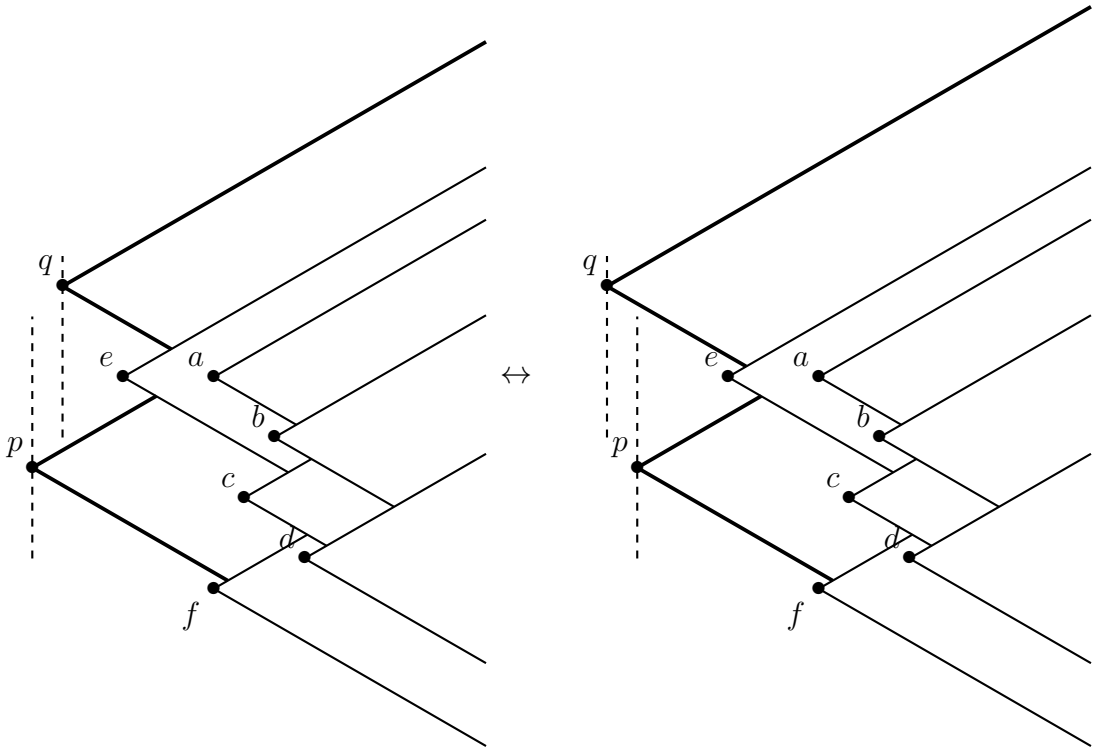


Figura 33: Se p não está em $Hits_{up}(q)$, ou se q não está em $Hits_{low}(p)$, nada acontece.

q, até a raiz que aponta para v. Devemos então remover q de $Cands(v)$ e inseri-lo em $Cands(p)$. A mudança também afetará todos os pontos que estão à esquerda de p e estão em $Hits_{up}(q)$. Para achar esses pontos, buscamos o ponto t em $Hits_{up}(q)$ mais à esquerda que está à direita de p. Chamamos *splay* para o nó que contém t e chamamos *split* para a subárvore esquerda desse nó e juntamos essa árvore em $Hits_{up}(p)$, pois são todos os pontos à esquerda de p que tinham q como *up* e agora seu novo *up* é p. Se esse ponto t não existe, todos os pontos de $Hits_{up}(q)$ devem ser transferidos para $Hits_{up}(p)$, veja a figura 35, e buscamos pelo ponto t tal que q está em $Hits_{low}(t)$. Por fim, removemos o ponto p de $Hits_{low}(q)$ e o inserimos em $Hits_{low}(t)$. Se p não está em $Hits_{low}(q)$ não haverão mudanças.

Se q está em $Cands(p)$, ou seja, q está saindo de $Dom(p)$ como demonstrado na figura 34 da direita para esquerda, então a troca afetará o ponto t tal que p está em $Hits_{low}(t)$. Se o ponto t existe, removemos p de $Hits_{low}(t)$. Devemos agora inserir p em $Hits_{low}(q)$, já que q é o novo $low(p)$. A mudança também afetará os pontos de $Hits_{up}(p)$ que agora deverão estar em $Hits_{up}(q)$. Para achar esses pontos, buscamos pelo ponto v em $Hits_{up}(p)$ mais à direita que não deveria estar em $Hits_{up}(q)$, chamamos *splay* para o nó que contém v e um *split* para sua subárvore direita, essa nova árvore deve ser incorporada a $Hits_{up}(q)$. Se tal ponto v não existe, todos os nós de $Hits_{up}(p)$ devem ser passados para $Hits_{up}(q)$. Por fim, devemos achar o novo ponto u tal que q deve estar em $Cands(u)$. Se o ponto v descrito anteriormente existe, então $u = v$. Se v não existe, então u é o

ponto tal que p está em $Cands(u)$. Dessa forma, retiramos q de $Cands(p)$ e o inserimos em $Cands(u)$. Se q não está em $Cands(p)$, não haverão mudanças.

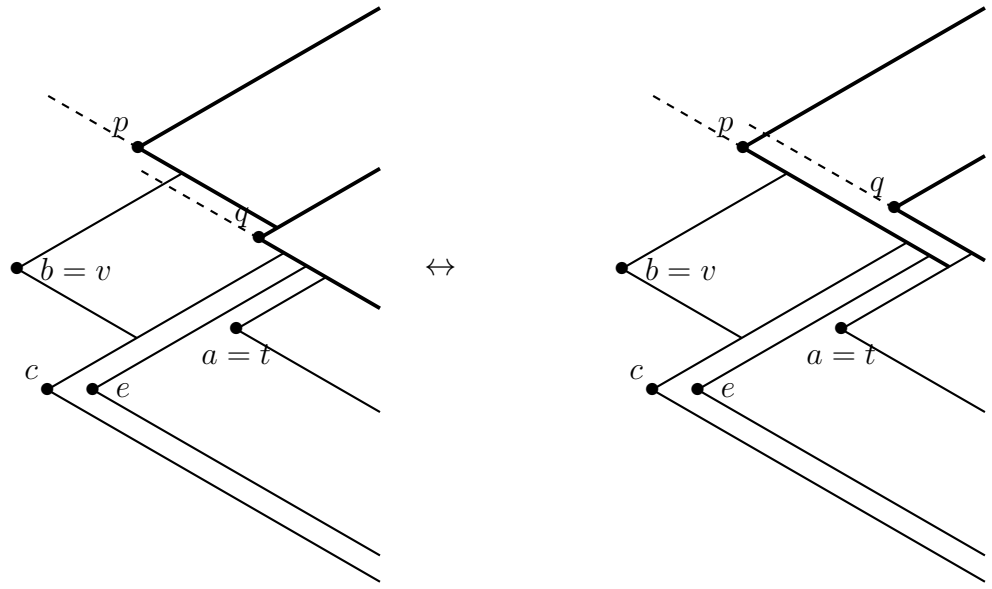


Figura 34: Da esquerda para direita, o caso em que p está em $Hits_{low}(q)$, ou seja, q está entrando em $Dom(p)$. Da direita para esquerda, o caso em que q está em $Cands(p)$, saindo de $Dom(p)$.

Um evento em que ocorre uma mudança na $-60°$-ordem, a ordem dos pontos projetados no eixo $x - 60°$, é simétrico a um evento na $60°$-ordem. Os pontos envolvidos no evento serão p e q e vamos assumir que p é o ponto mais à esquerda e abaixo de q. O evento pode provocar a entrada ou saída do ponto q de $Cands(p)$, veja a figura 36. O algoritmo 36 implementa a sequência de operações referentes a esse evento.

Se p está em $Hits_{up}(q)$ (q está entrando em $Dom(p)$), como demonstrado na figura 36, então a troca na $-60°$-ordem afetará o ponto v tal que q está em $Cands(v)$. Achamos esse ponto subindo em $Cands(v)$, a partir do nó que contém q, até a raiz que aponta para v. Devemos então remover q de $Cands(v)$ e inseri-lo em $Cands(p)$. A mudança também afetará todos os pontos que estão à esquerda de p e estão em $Hits_{low}(q)$. Para achar esses pontos buscamos o ponto t em $Hits_{low}(q)$ mais à esquerda que está a direita de p. Chamamos *splay* para o nó que contém t e *split* para a nova subárvore esquerda desse nó e juntamos essa árvore em $Hits_{low}(p)$, pois são todos pontos à esquerda de p que tinham q como *low* e agora seu novo *low* é p. Se esse ponto t não existe, veja a figura 37, então buscamos pelo ponto t tal que q está em $Hits_{up}(t)$. Por fim, removemos o ponto p de $Hits_{up}(q)$ e o inserimos em $Hits_{up}(t)$. Se p não está em $Hits_{up}(q)$ não haverão mudanças.

Se q está em $Cands(p)$ (q está saindo de $Dom(p)$), como demonstrado na figura 36, então a troca afetará o ponto t tal que p está em $Hits_{up}(t)$. Se o ponto t existe, removemos p de $Hits_{up}(t)$. Devemos agora inserir p em $Hits_{up}(q)$, já que q é o novo $up(p)$. A mudança

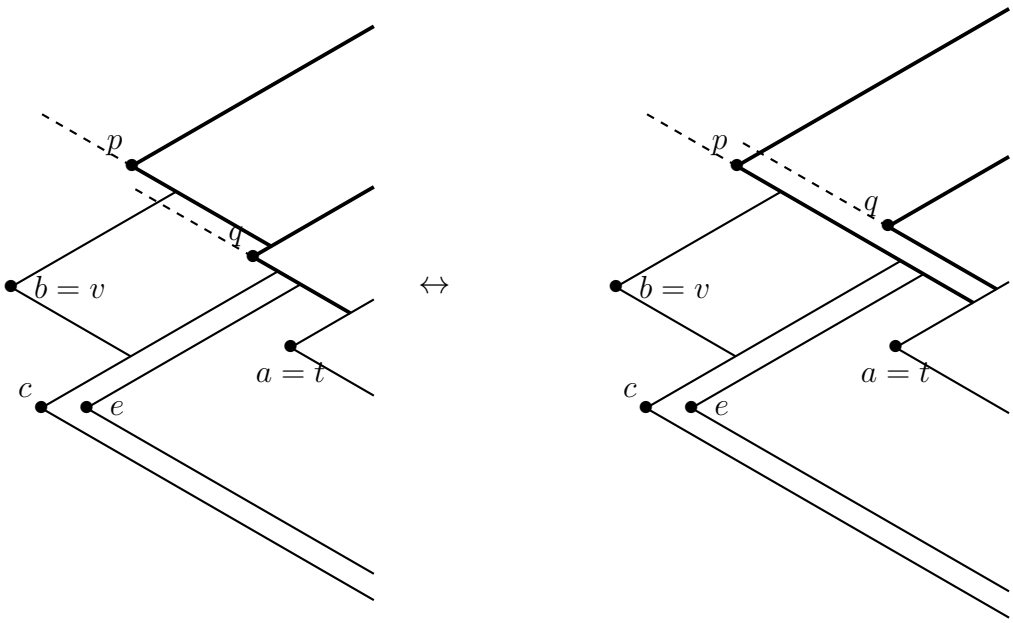


Figura 35: Da esquerda para a direita, todos os pontos de $Hits_{up}(q)$ são transferidos para $Hits_{up}(p)$ e p, que está em $Hits_{low}(q)$, é transferido para $Hits_{low}(t)$. Da direita para esquerda, os pontos em $Hits_{up}(p)$ são transferidos para $Hits_{up}(q)$ e p, que está em $Hits_{low}(t)$, é transferido para $Hits_{low}(q)$.

também afetará os pontos de $Hits_{low}(p)$ que agora atingem $Dom(q)$. Para achar esses pontos, buscamos pelo ponto v em $Hits_{low}(p)$ mais à direita que não atinge $Dom(q)$, chamamos *splay* para o nó que contém v e um *split* para sua subárvore direita. Essa nova árvore deve ser incorporada a $Hits_{low}(q)$. Se tal ponto v não existe, todos os nós de $Hits_{low}(p)$ devem ser passados para $Hits_{low}(q)$. Por fim, devemos achar o novo ponto u tal que q deve estar em $Cands(u)$. Se o ponto v descrito anteriormente existe, então $u = v$. Se v não existe, então u é o ponto tal que p está em $Cands(u)$. Dessa forma, retiramos q de $Cands(p)$ e o inserimos em $Cands(u)$. Se q não está em $Cands(p)$, não haverão mudanças.

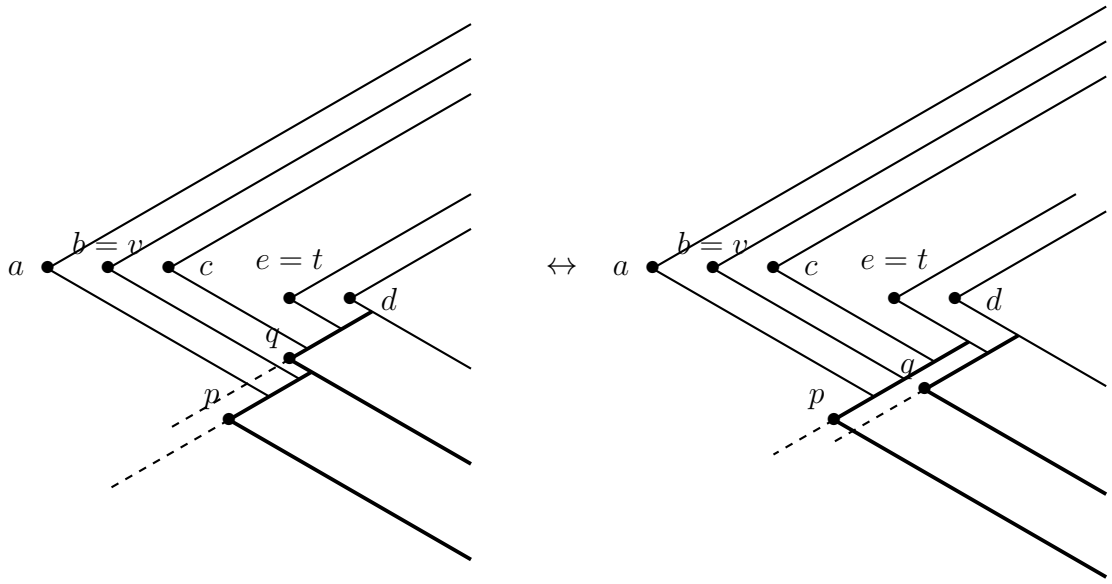


Figura 36: Da esquerda para direita, o caso em que p está em $Hits_{up}(q)$, ou seja, q está entrando em $Dom(p)$. Da direita para esquerda, o caso em que q está em $Cands(p)$, saindo de $Dom(p)$.

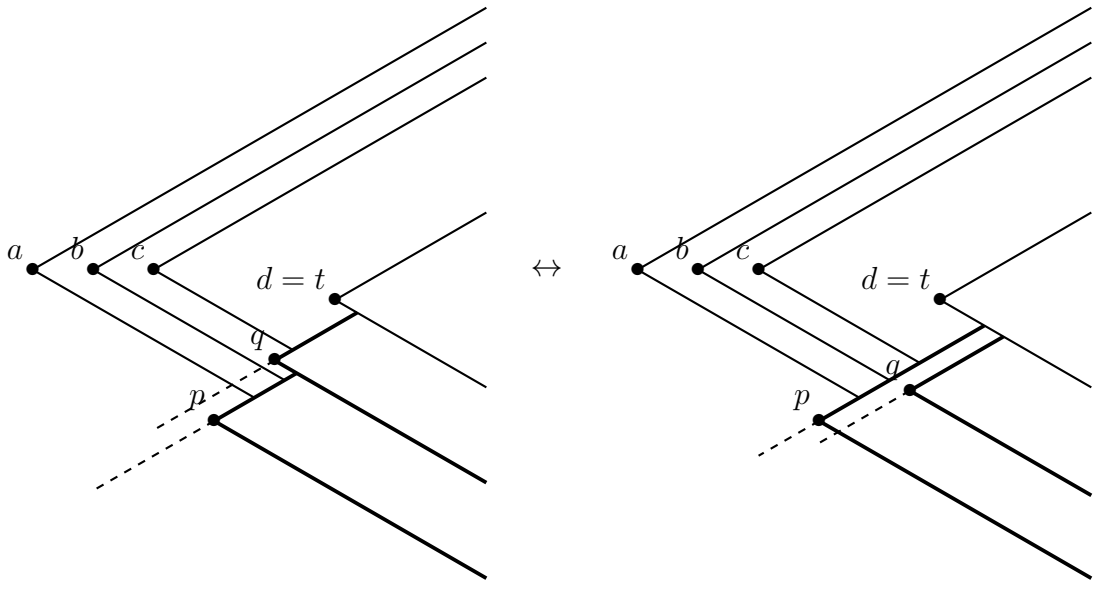


Figura 37: Da direita para esquerda, p, que estava em $Hits_{up}(q)$ foi transferido para $Hits_{up}(t)$ e todos os elementos em $Hits_{low}(q)$ foram transferidos para $Hits_{low}(p)$. Da esquerda para direita, p, que estava em $Hits_{up}(t)$, foi transferido para $Hits_{up}(q)$ e os pontos à direita de v em $Hits_{low}(p)$ foram transferidos para $Hits_{low}(q)$.

Algoritmo 4.33: Função HORIZONTALEVENT.

```
1:  function HORIZONTALEVENT(p, q, dir)
2:      if q = OWNER(p.hitsUp(dir)) :
3:          t ← PREDECESSOR(p, Cands(q, dir), U)
4:          if t = NULL :
5:              t ← OWNER(q.hitsUp(dir))
6:          newCands ← SPLIT(NULL, t, Cands(q, dir))
7:          JOIN(Cands(p, dir), newCands)
8:          DELETE(p, Hits_up(q, dir))
9:          if t ≠ NULL :
10:             INSERT(p, Hits_up(t, dir))
11:         w ← OWNER(q.hitsLow(dir))
12:         if w ≠ NULL :
13:             DELETE(q, Hits_low(w, dir))
14:         INSERT(q, Hits_low(p, dir))
15:     else
16:         if p = OWNER(q.hitsLow(p, dir)) :
17:             t ← PREDECESSOR(q, Cands(p, dir), D)
18:             if t = NULL :
19:                 t ← OWNER(p.hitsLow(dir))
20:             newCands ← SPLIT(t, NULL, Cands(p, dir))
21:             JOIN(Cands(q, dir), newCands)
22:             DELETE(q, Hits_low(p, dir))
23:             if t ≠ NULL :
24:                 INSERT(q, Hits_low(t, dir))
25:             w ← OWNER(p.hitsUp(dir))
26:             if w ≠ NULL :
27:                 DELETE(p, Hits_up(w, dir))
28:             INSERT(p, Hits_up(q, dir))
29:     v ← OWNER(p.cands(dir))
30:     v' ← OWNER(q.cands(dir))
31:     if v = v' :
32:         UPDATELCAND(v, dir)
```

Algoritmo 4.34: Função upEvent.

```
1:  function upEvent(p, q, dir)
2:      if q = owner(p.hitsLow(dir)) :
3:          v ← owner(q.cands(dir))
4:          if v ≠ NULL :
5:              delete(q, Cands(v, dir))
6:          insert(q, Cands(p, dir))
7:          t ← successor(p, Hits_up(q, dir), H)
8:          if t = NULL :
9:              t ← owner(q.hitsLow(dir))
10:         newHits ← split(NULL, t, Hits_up(q, dir))
11:         join(Hits_up(p, dir), newHits)
12:         delete(p, Hits_low(q, dir))
13:         if t ≠ NULL :
14:             insert(p, Hits_low(t, dir))
15:     else
16:         if p = owner(q.cands(dir)) :
17:             t ← owner(p.hitsLow(dir))
18:             if t ≠ NULL :
19:                 delete(p, Hits_low(t, dir))
20:             insert(p, Hits_low(q, dir))
21:             v ← predecessor(q, Hits_up(p, dir), U)
22:             if v = NULL :
23:                 v ← owner(p.cands(dir))
24:             newHits ← split(v, NULL, Hits_up(p, dir))
25:             join(newHits, Hits_up(q, dir))
26:             delete(q, Cands(p, dir))
27:             if v ≠ NULL :
28:                 insert(q, Cands(v, dir))
```

Algoritmo 4.35: Função downEvent.

```
1:  function DOWNEVENT(p, q, dir)
2:      if q = OWNER(p.hitsUp(dir)) :
3:          v ← OWNER(q.cands(dir))
4:          if v ≠ NULL :
5:              DELETE(q, Cands(v, dir))
6:          INSERT(q, Cands(p, dir))
7:          t ← SUCCESSOR(p, Hits_low(q, dir), H)
8:          if t = NULL :
9:              t ← OWNER(q.hitsUp(dir))
10:         newHits ← SPLIT(NULL, t, Hits_low(q, dir))
11:         JOIN(Hits_low(p, dir), newHits)
12:         DELETE(p, Hits_up(q, dir))
13:         if t ≠ NULL :
14:             INSERT(p, Hits_up(t, dir))
15:     else
16:         if p = OWNER(q.cands(dir)) :
17:             t ← OWNER(p.hitsUp(dir))
18:             if t ≠ NULL :
19:                 DELETE(p, Hits_up(t, dir))
20:             INSERT(p, Hits_up(q, dir))
21:             v ← PREDECESSOR(q, Hits_low(p, dir), D)
22:             if v = NULL :
23:                 v ← OWNER(p.cands(dir))
24:             newHits ← SPLIT(v, NULL, Hits_low(p, dir))
25:             JOIN(newHits, Hits_low(q, dir))
26:             DELETE(q, Cands(p, dir))
27:             if v ≠ NULL :
28:                 INSERT(q, Cands(v, dir))
```

4.2.1 Tratamento de casos degenerados

Como citado anteriormente, a nossa fila de prioridade Q guarda os certificados cuja chave é t, o instante de tempo em que o certificado expira. Porém, é possível que dois ou mais certificados expirem no mesmo instante de tempo t, configurando um empate na nossa fila. Nos problemas anteriores, tais empates costumavam ser tratados por algum critério de desempate que era suficiente para garantir que as estruturas permanecessem corretas. No entanto, para o par mais próximo cinético encontrar tal critério não é uma tarefa tão simples.

Para olhar mais profundamente para o problema vamos antes considerar que os eventos associados ao torneio não fazem parte da fila.

Algumas situações em que dois certificados vencem no mesmo instante acabam não causando problemas, por exemplo, dois eventos que envolvem pares de pontos distintos (p, q) e (r, s), ocorrendo no mesmo instante de tempo t. Contudo, quando dois pontos se colidem, isto é, atingem a mesma posição em $\mathbb{R}^2$ num instante t', ocorrem trocas nas três direções ($0°$, $+60°$ e $-60°$). Essas trocas precisam ser realizadas de maneira consistente, para que as estruturas possam ser atualizadas da devida maneira.

A colisão não necessariamente envolve apenas dois pontos, podem haver mais pontos se colidindo no mesmo instante e, nesse caso, precisamos nos preocupar também como as trocas afetam os pares de pontos, mas como elas afetam todos o pontos envolvidos no geral.

Para determinar em que ordem os eventos devem ocorrer podemos simular um evento não degenerado. No caso, vamos simular que os pontos desviam em uma trajetória circular, ao longo de uma circunferência de raio proporcional à sua velocidade, com mesma velocidade angular, e depois seguem em frente normalmente em suas trajetórias lineares. Adicionaremos então um parâmetro $\theta \in [0, \pi]$ que indica o ângulo a ser percorrido até que o evento aconteça simulando essa trajetória circular, veja a figura 38.

Como esse novo critério de desempate somos capazes de simular essa trajetória não degenerada e os eventos ocorrerão em uma ordem apropriada, veja um exemplo na figura 39.

Existe ainda um outro detalhe de implementação que vale ser mencionado: quando temos vários pontos se colidindo no mesmo instante t' diversos eventos serão acionados em uma determinada ordem e, durante as alterações realizadas nos eventos, são realizadas checagens de predecessor e sucessor em uma determinada ordem. Porém, se os dois pontos se encontram na exata mesma posição, não é possível determinar quem é o sucessor e quem é o predecessor considerando apenas a posição dos pontos.

Um critério que pode ser utilizado para resolver esse problema é o seguinte: se os dois pontos possuem a mesma coordenada é certo que existiu, existe ou existirá um certificado

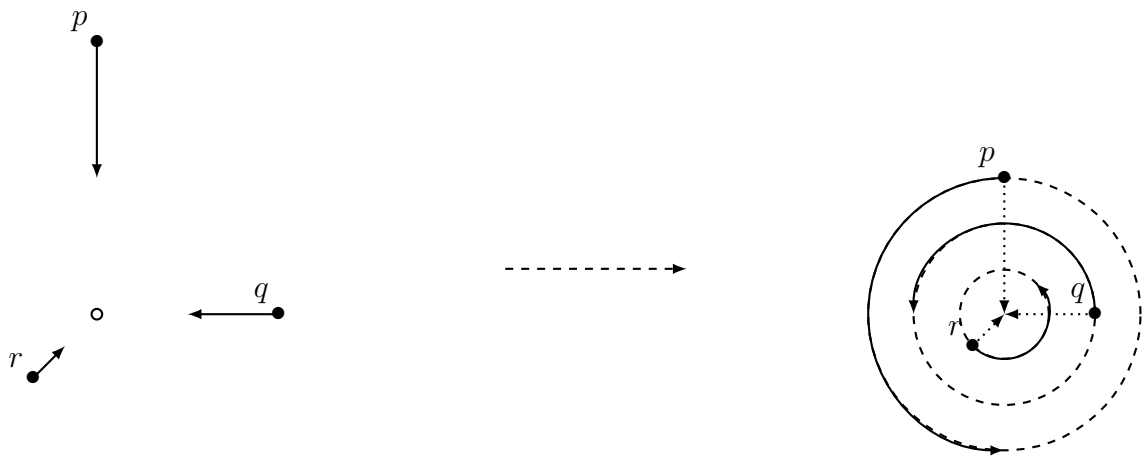


Figura 38: Exemplo de colisão entre três pontos.

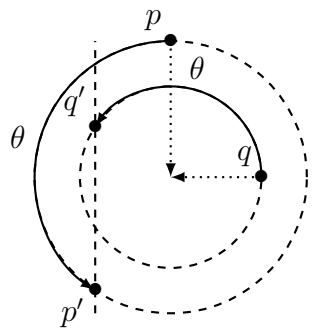


Figura 39: Exemplo de parâmetro de evento horizontal ocorrendo entre p e q, com a trajetória simulada.

na fila relacionado à troca da ordem entre eles em uma determinada direção. Então, comparamos esse certificado com o certificado atual da fila para verificar se ele já foi processado ou ainda virá a ser processado. Com isso, podemos determinar se queremos saber da situação em um instante $t' - \epsilon$ ou $t' + \epsilon$, que pode ser computada facilmente considerando apenas o vetor velocidade dos pontos.