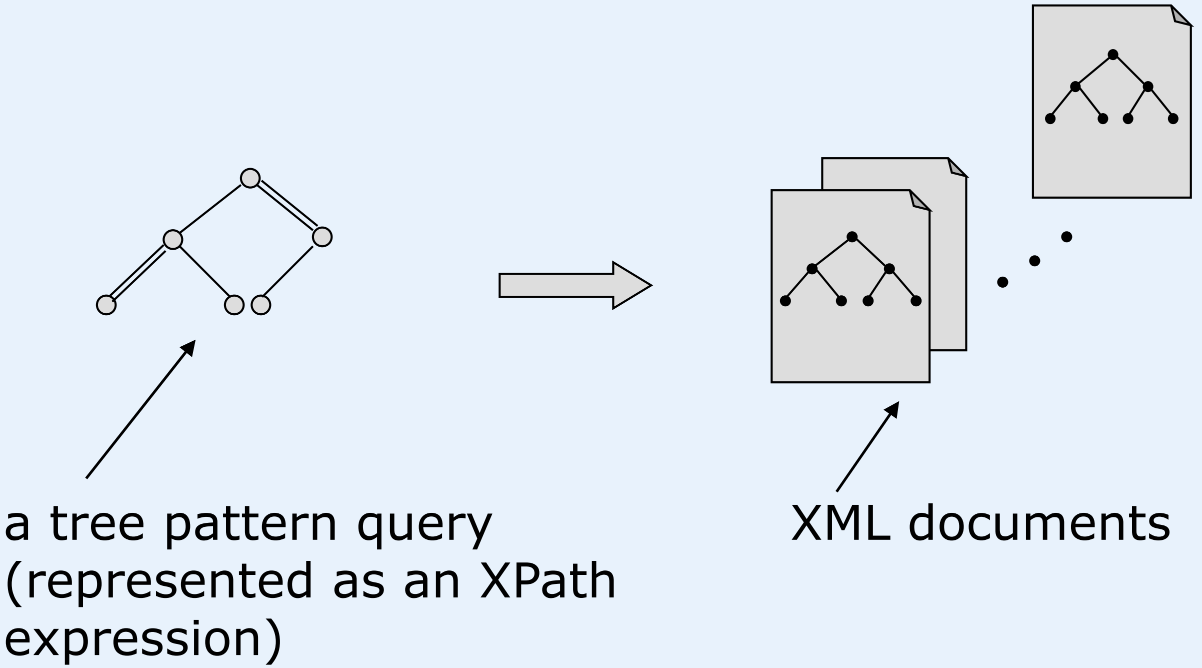
# Evaluation of Tree Pattern Queries

- Motivation
- Tree encoding and XML data streams
- Evaluation of unordered tree pattern queries
- Evaluation of ordered tree pattern queries
- XB-trees

# Motivation

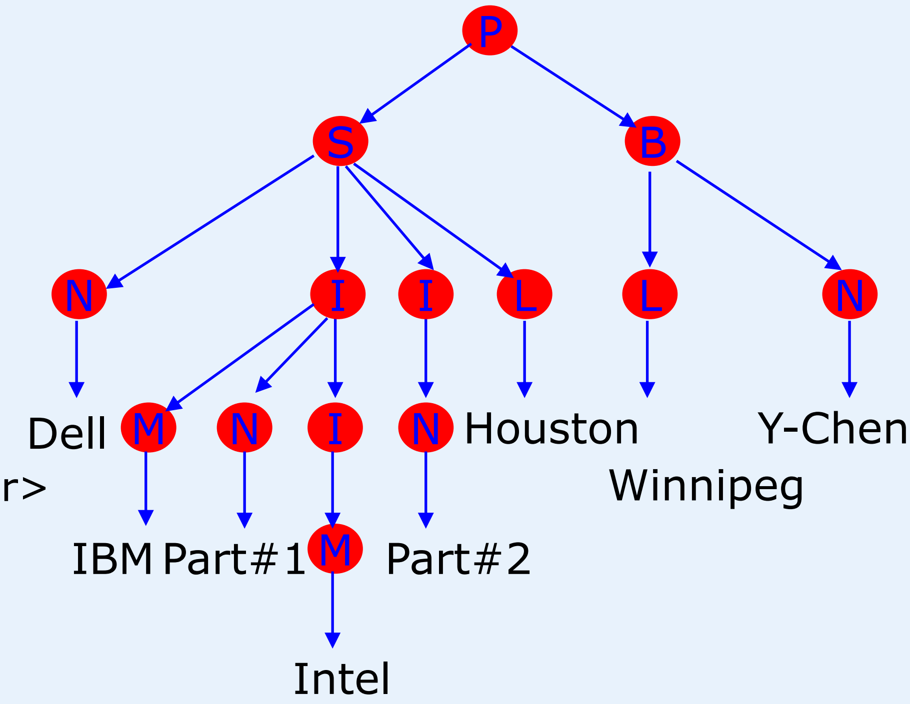
- **Efficient method to evaluate XPath expression queries – XML query processing**



# Motivation

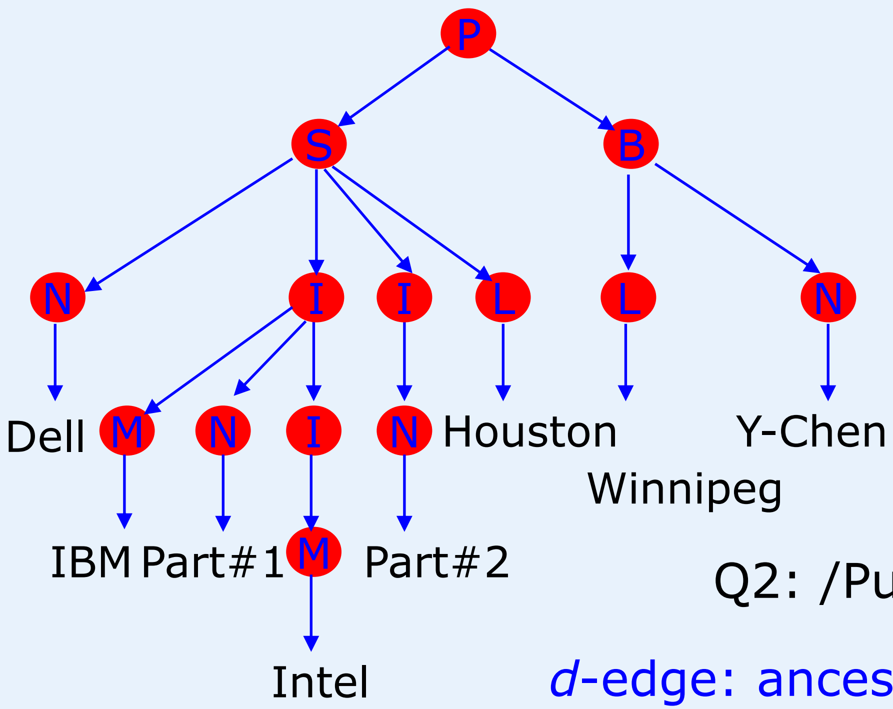
## Document:

```
<Purchase>
   <Seller>
      <Name>dell</Name>
      <Item>
         <Manufacturer>IBM</Manufacturer>
         <Name>part#1</Name>
         <Item>
            <Manufacturer>Intel</Manufacturer>
         </Item>
      </Item>
      <Item>
         <Name>Part#2</Name>
      </Item>
      <Location>Houston</Location>
   </Seller>
   <Buyer>
      <Location>Winnipeg</Location>
      <Name>Y-Chen</Name>
   </Buyer>
</Purchase>
```



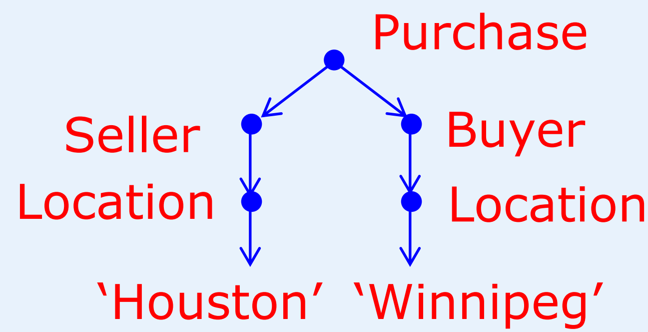
# Motivation

## Document:




Query – XPath expressions:

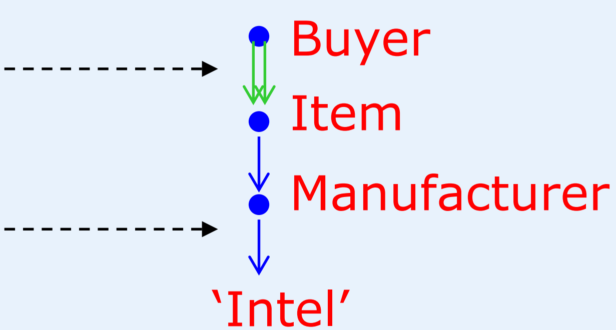
Q1: /Purchase[Seller/Location=‘Houston’]/
Buyer[Location = ‘Winnipeg’]




Q2: /Purchase//Item[Manufacturer = ‘Intel’]

*d*-edge: ancestor-descendant relationship

*c*-edge: parent-child relationship



# Tree Encoding

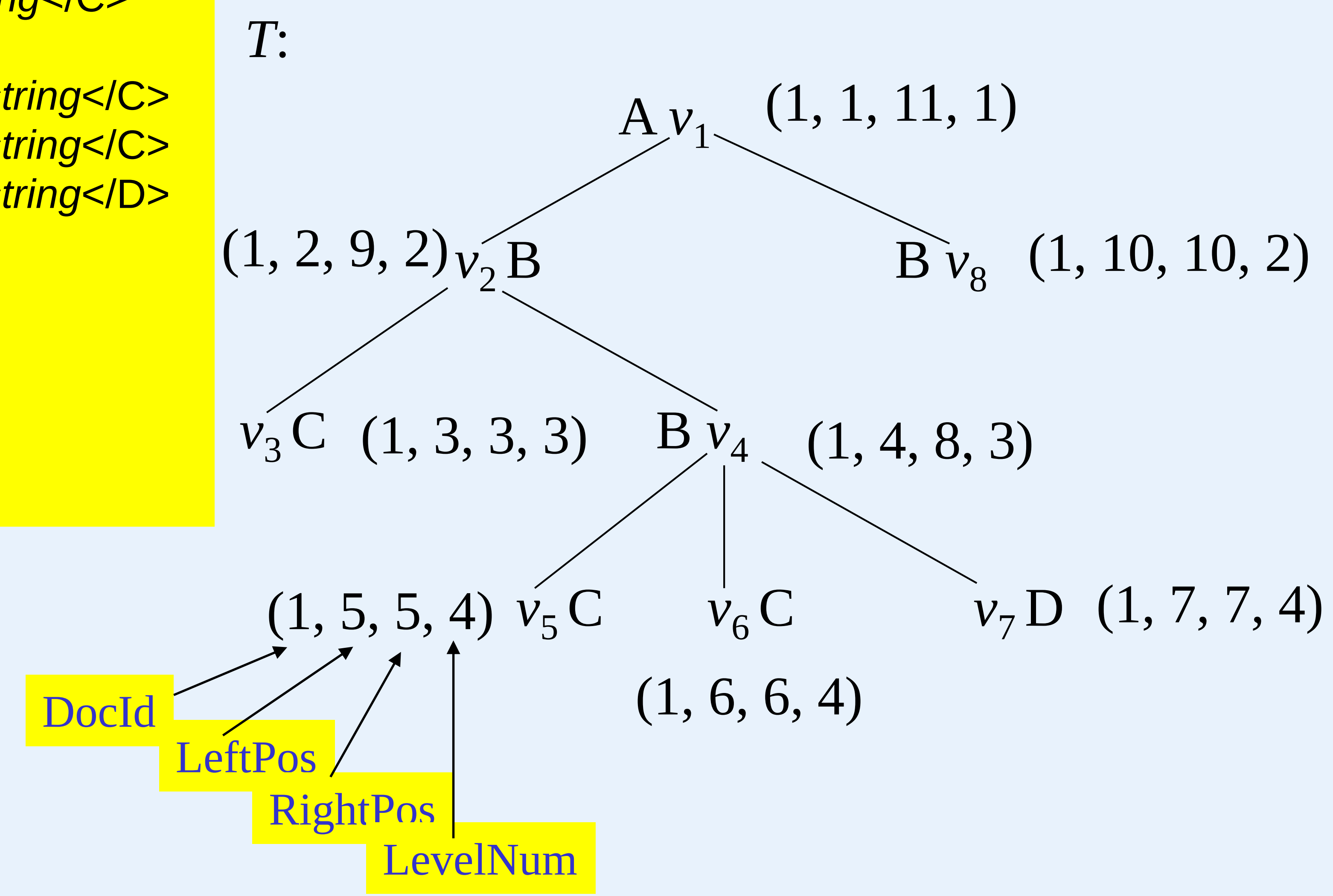
Let *T* be a document tree. We associate each node *v* in *T* with a quadruple (*DocId, LeftPos, RightPos, LevelNum*), denoted as $\alpha(v)$, where

- DocId is the document identifier;
- LeftPos and RightPos are generated by counting word numbers from the beginning of the document until the *start* and *end* of the element, respectively; and
- LevelNum is the nesting depth of the element in the document.

By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined.

# Evaluation of Tree Pattern Queries

```
<A>
   <B>
      <C>string</C>
      <B>
         <C>string</C>
         <C>string</C>
         <D>string</D>
      </B>
   </B>
   <B>
      string
   </B>
</A>
```



## Tree Encoding

*(i)* *ancestor-descendant*: a node $v_1$ associated with $(d_1, l_1, r_1, ln_1)$ is an ancestor of another node $v_2$ with $(d_2, l_2, r_2, ln_2)$ iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$.

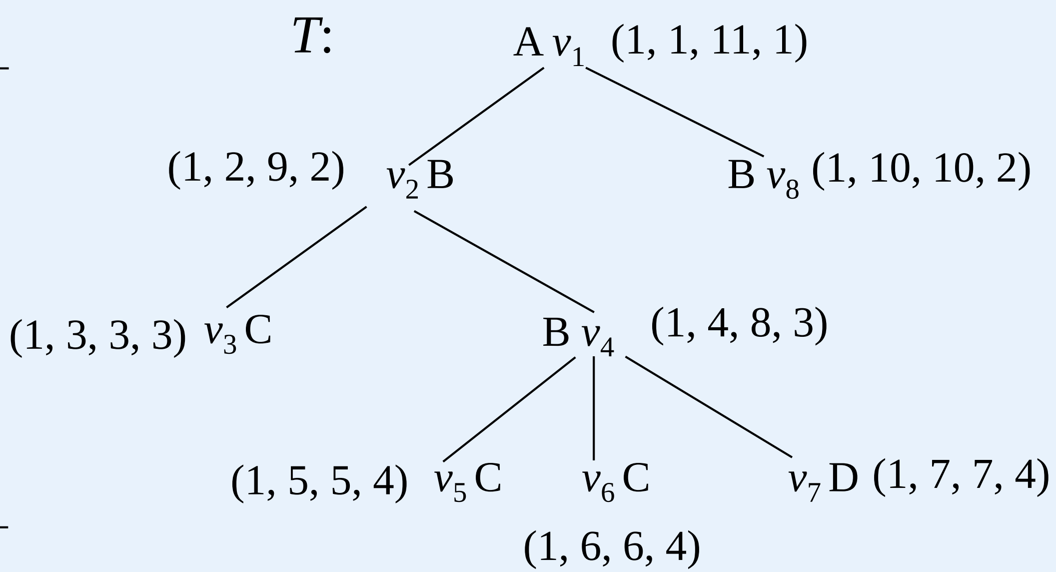
*(ii)* *parent-child*: a node $v_1$ associated with $(d_1, l_1, r_1, ln_1)$ is the parent of another node $v_2$ with $(d_2, l_2, r_2, ln_2)$ iff $d_1 = d_2$, $l_1 < l_2$, $r_1 > r_2$, and $ln_2 = ln_1 + 1$.

*(iii)* *from left to right*: a node $v_1$ associated with $(d_1, l_1, r_1, ln_1)$ is to the left of another node $v_2$ with $(d_2, l_2, r_2, ln_2)$ iff $d_1 = d_2$, $r_1 < l_2$.

## Data Streams

| A: | B: |
|---|---|
| (1, 1, 11, 1) | (1, 2, 9, 2) |
| | (1, 4, 8, 3) |
| | (1, 10, 10, 2) |

| C: | D: |
|---|---|
| (1, 3, 3, 3) | (1, 7, 7, 4) |
| (1, 5, 5, 4) | |
| (1, 6, 6, 4) | |

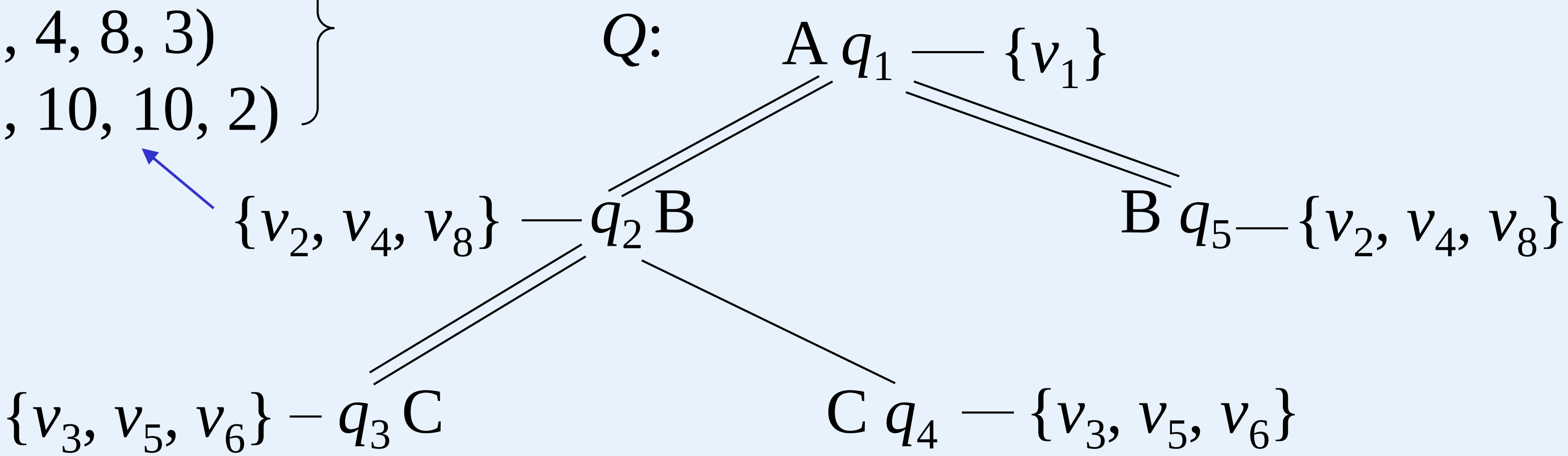



The data streams are sorted by (DocID, LeftPos).

# Tree Pattern queries

XPath: /A[.//B[.//C]/C]//B

(1, 2, 9, 2)
(1, 4, 8, 3)
(1, 10, 10, 2)




═══ descendant edge (//-edge, $u \Rightarrow v$)

─── child edge (/-edge, $u \rightarrow v$)

# Data Streams – *B*(*q*)'s (Sorted according to LeftPos)

Search tree in preorder (top-down)

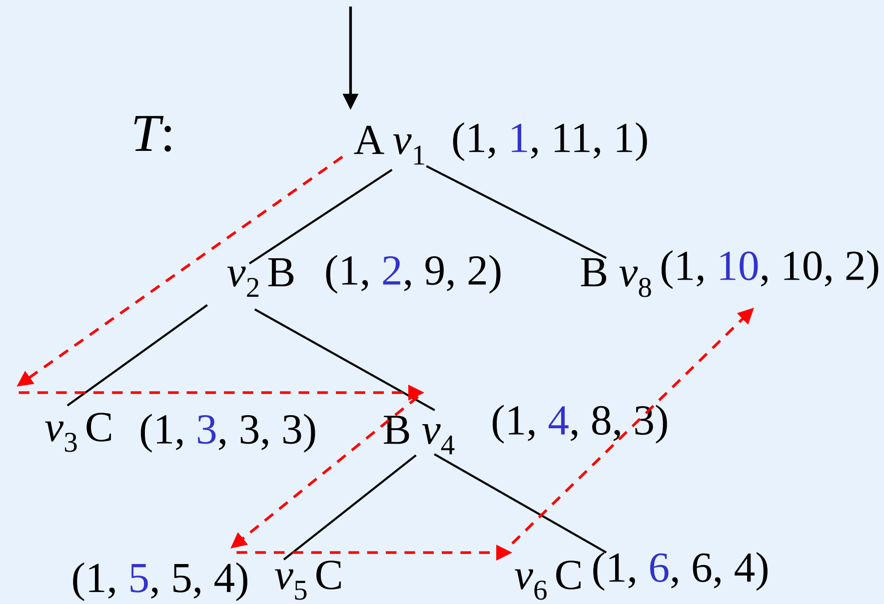
$B(q_1)$:

(1, 1, 11, 1) $v_1$

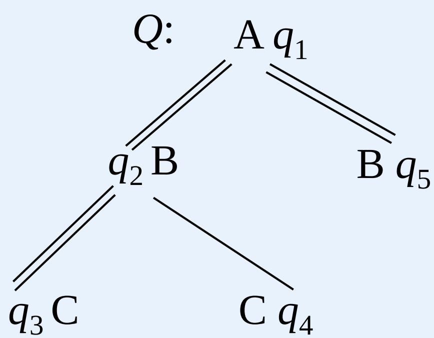
$B(\{q_2, q_5\})$:

(1, 2, 9, 2) $v_2$
(1, 4, 8, 3) $v_4$
(1, 10, 10, 2) $v_8$

*T*:




*Q*:




$B(\{q_3, q_4\})$:

(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

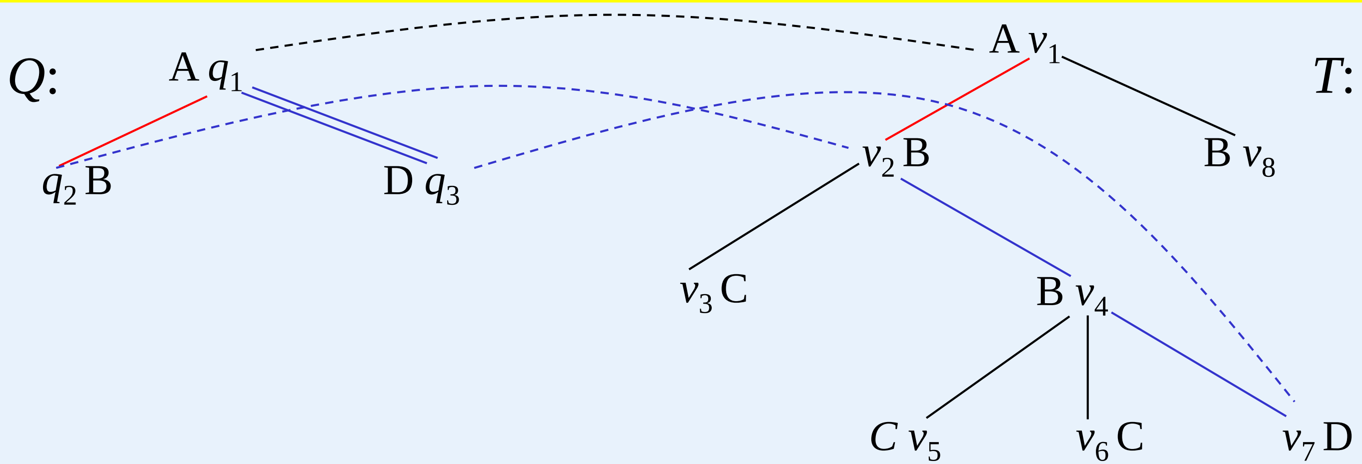
The data streams are sorted by (DocID, LeftPos).

# Unordered Tree Matching

**Definition** An embedding of a tree pattern $Q$ into an XML document $T$ is a mapping $f$: $Q \rightarrow T$, from the nodes of $Q$ to the nodes of $T$, which satisfies the following conditions:

*(i)* *Preserve node type*: For each $u \in Q$, $u$ and $f(u)$ are of the same tag, (or more generally, $u$'s label is the same as $f(u)$'s label.)

*(ii)* *Preserve ancestor/descendant-parent/child relationships*: If $u \rightarrow v$ in $Q$, then $f(v)$ is a child of $f(u)$ in $T$; if $u \Rightarrow v$ in $Q$, then $f(v)$ is a descendant of $f(u)$ in $T$.



**Algorithm for Unordered Tree Matching Based on Two Concepts:**

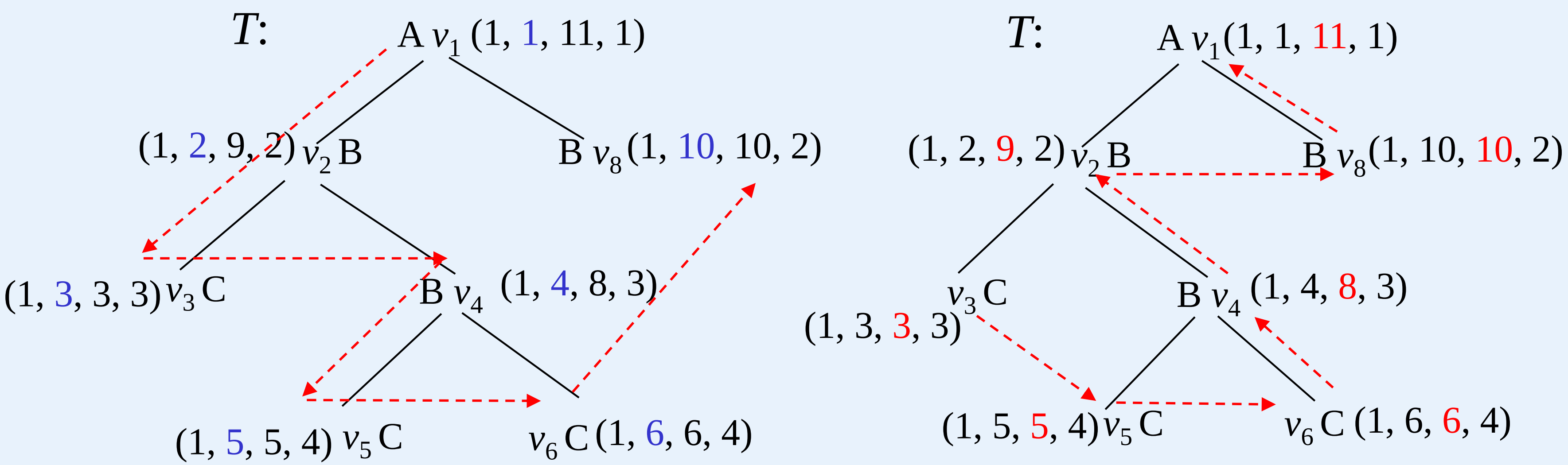
- **XML Data Stream Transformation**

- **Matching Subtrees**

The data stream transformation can be done for the documents, independent of queries.

## Data Stream Transformation

- Note that iterating through the stream nodes in sorted order of their LeftPos values corresponds to access of document nodes in preorder (top-down search).
- We can transform a data stream to another, in which the quadruples are sorted by RightPos values, corresponding to a search in postorder (bottom-up search). (It is because our algorithm needs to access the data stream in this way.)



# Evaluation of Tree Pattern Queries

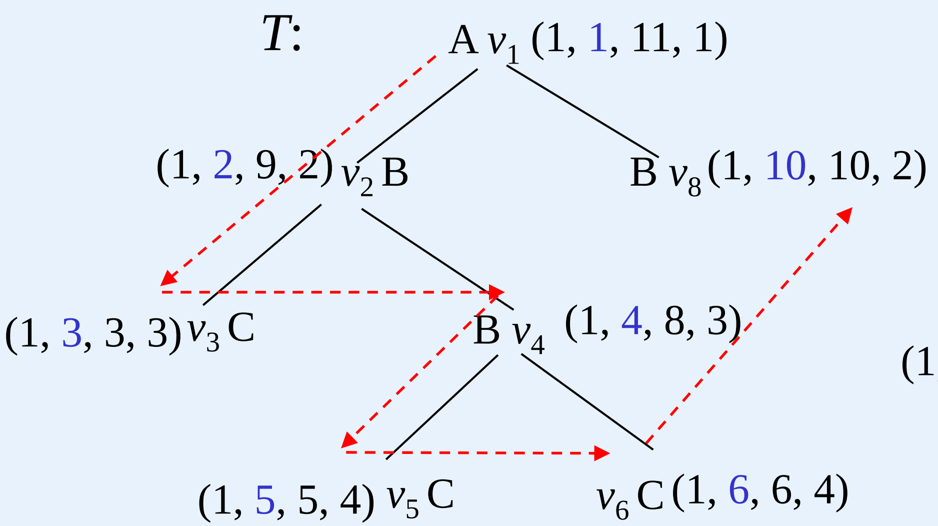
*T*

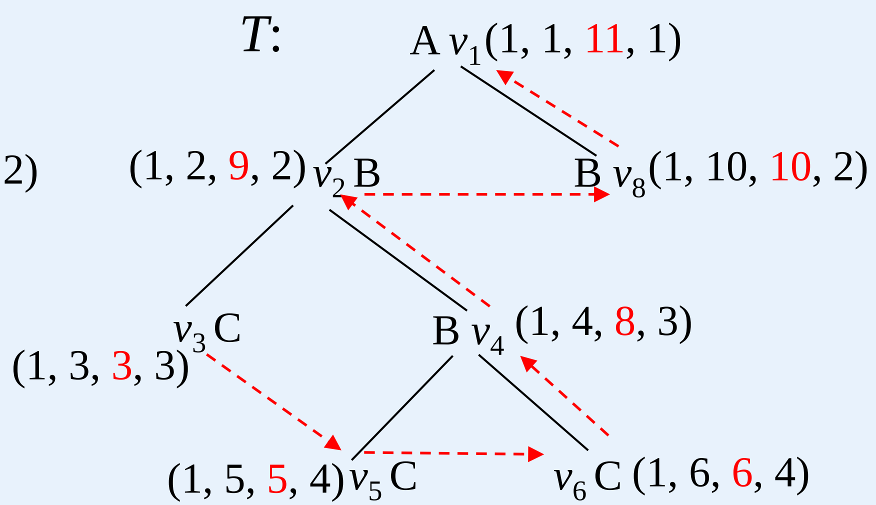
A (1, 1, 11, 1)
B (1, 2, 9, 2)
C (1, 3, 3, 3)
B (1, 4, 8, 3)
C (1, 5, 5, 4)
C (1, 6, 6, 4)
B (1, 10, 10, 2)

transformation

*T*

C (1, 3, 3, 3)
C (1, 5, 5, 4)
C (1, 6, 6, 4)
B (1, 4, 8, 3)
B (1, 2, 9, 2)
B (1, 10, 10, 2)
C (1, 1, 11, 1)



# **Data Streams – $L(q)$'s (Sorted according to RightPos)**

$L(q_1)$:

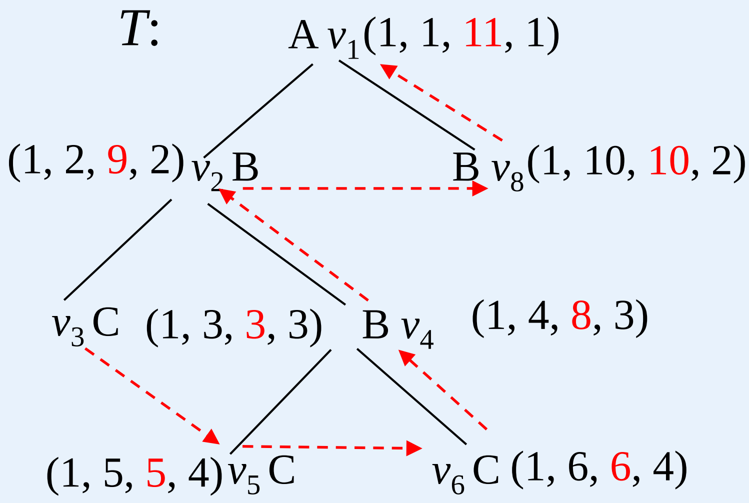
(1, 1, 11, 1) $v_2$

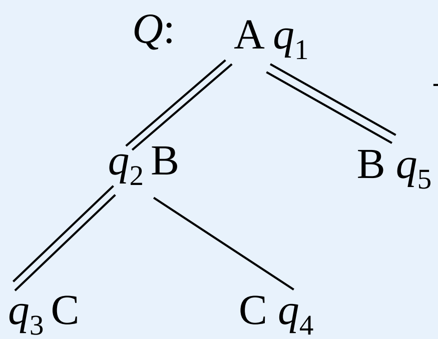
$L(\{q_2, q_5\})$:

(1, 4, 8, 3) $v_4$

(1, 2, 9, 2) $v_2$

(1, 10, 10, 2) $v_8$




$L(q_3, q_4)$:

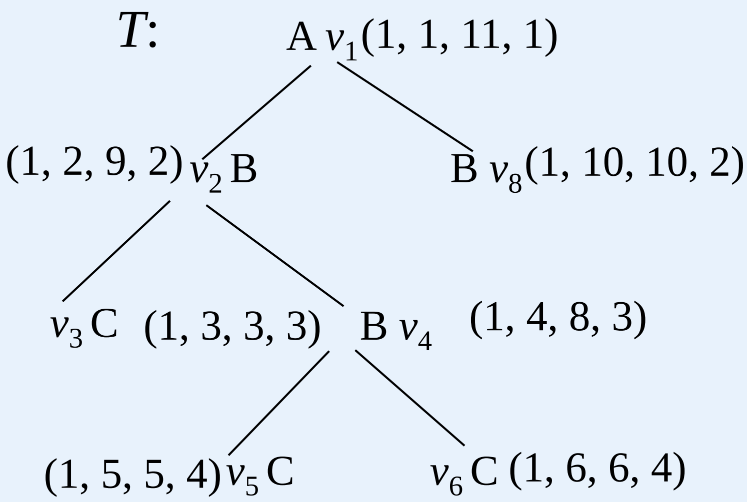
(1, 3, 3, 3) $v_3$

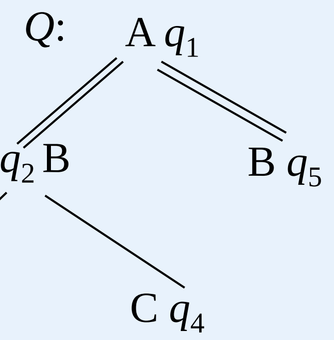
(1, 5, 5, 4) $v_5$

(1, 6, 6, 4) $v_6$

The data streams are sorted by (DocID, RightPos).

# Evaluation of Tree Pattern Queries




| $B(q_1)$: |
|---|
| (1, 1, 11, 1) $v_1$ |

| $B(\{q_2, q_5\})$: |
|---|
| (1, 2, 9, 2) $v_2$ |
| (1, 4, 8, 3) $v_4$ |
| (1, 10, 10, 2) $v_8$ |

| $B(\{q_3, q_4\})$: |
|---|
| (1, 3, 3, 3) $v_3$ |
| (1, 5, 5, 4) $v_5$ |
| (1, 6, 6, 4) $v_6$ |

$\Rightarrow$

| $L(q_1)$: |
|---|
| (1, 1, 11, 1) $v_2$ |

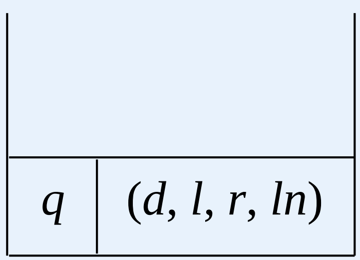
| $L(\{q_2, q_5\})$: |
|---|
| (1, 4, 8, 3) $v_4$ |
| (1, 2, 9, 2) $v_2$ |
| (1, 10, 10, 2) $v_8$ |

| $L(q_3, q_4)$: |
|---|
| (1, 3, 3, 3) $v_3$ |
| (1, 5, 5, 4) $v_5$ |
| (1, 6, 6, 4) $v_6$ |

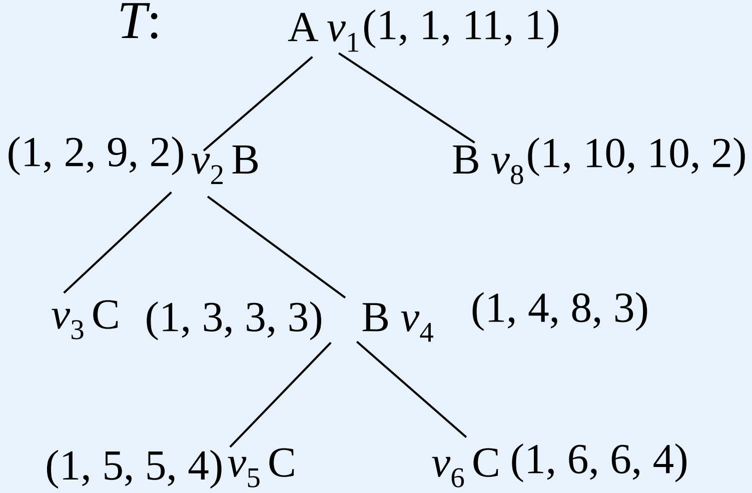
# Algorithm for Data Stream Transformation

- We maintain a global stack *ST* to make a transformation of data streams using the following algorithm.
- In *ST*, each entry is a pair $(q, v)$ with $q \in Q$, $v \in T$ ($v$ is represented by its quadruple) and $label(v) = label(q)$.

*ST*:




*T*:



```
Algorithm stream-transformation(B(q_i)'s)
input: all data streams B(q_i)'s, each sorted by LeftPos.
output: new data streams L(q_i)'s, each sorted by RightPos.
begin
    repeat until each B(q_i) becomes empty
    {       identify q_i such that the first element v of B(q_i) is of
            the minimal LeftPos value; remove v from B(q_i);
            while ST is not empty and ST.top is not v's ancestor do
            {   x ← ST.pop(); Let x = (q_j, u);
                put u at the end of L(q_j);
            }
            ST.push(q_i, v);
    }
    Pop out all the remaining elements in ST and insert them into the
    corresponding L(q_i)'s;
end
```

$B(q_1)$ - A:

(1, 1, 11, 1) $v_1$

$B(\{q_2, q_5\})$ - B:

(1, 2, 9, 2) $v_2$
(1, 4, 8, 3) $v_4$
(1, 10, 10, 2) $v_8$

$B(\{q_3, q_4\})$ - C:

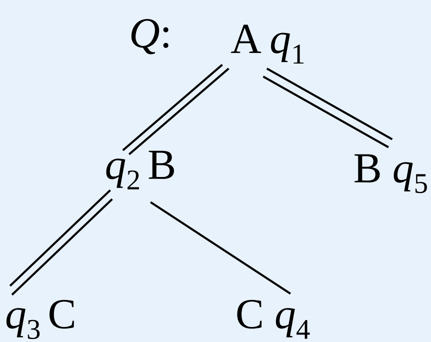
(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

$B(\,)$ - D:

(1, 7, 7, 4) $v_6$

- In the above algorithm, *ST* is used to keep all the nodes on a path until we meet a node *v* that is not a descendant of *ST.top*.
- Then, we pop up all those nodes that are not *v*'s ancestor; put them at the end of the corresponding $L(q_i)$'s (see lines 3 - 4), and push *v* into *ST* (see line 7), where $L(q_i)$ is another data stream created for $q_i$, sorted by (DocID, RightPos) values.
- All the data streams $L(q_i)$'s make up the output of the algorithm.
- However, we remark that the popped nodes are in postorder. So we can directly handle the nodes in this order without explicitly generating $L(q_i)$'s.

*Q*:

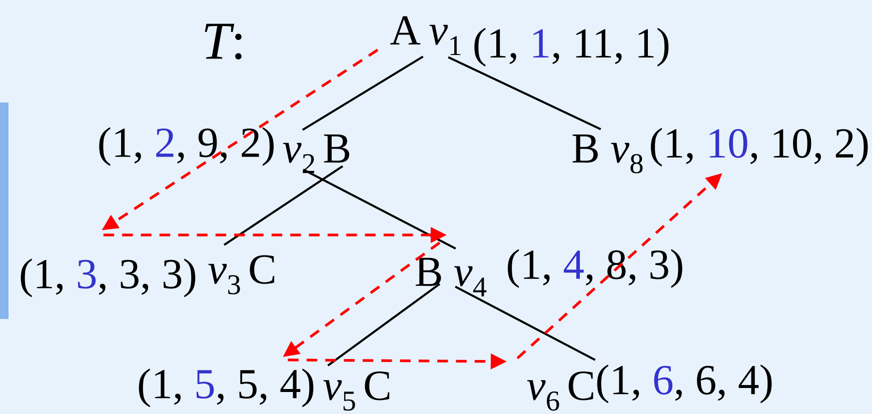



| $B(q_1)$ - A: |
|---|
| (1, 1, 11, 1) $v_1$ |

| $B(\{q_3, q_4\})$ - C: |
|---|
| (1, 3, 3, 3) $v_3$ |
| (1, 5, 5, 4) $v_5$ |
| (1, 6, 6, 4) $v_6$ |

| $B(\{q_2, q_5\})$ - B: |
|---|
| (1, 2, 9, 2) $v_2$ |
| (1, 4, 8, 3) $v_4$ |
| (1, 10, 10, 2) $v_8$ |

*T*:



*ST*:

| | |
|---|---|
| | |
| | |
| $q_1$ | $v_1$ |

$B(q_1)$ - A:

(1, 1, 11, 1) $v_1$

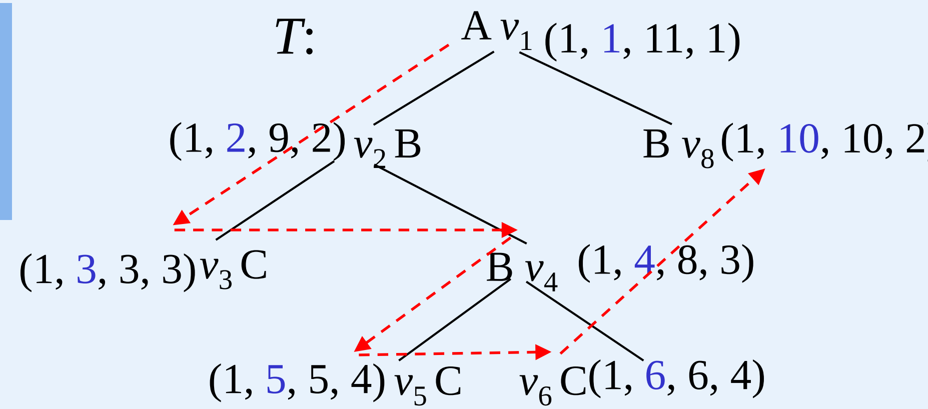
$B(\{q_3, q_4\})$ - C:

(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

$B(\{q_2, q_5\})$ - B:

(1, 2, 9, 2) $v_2$
(1, 4, 8, 3) $v_4$
(1, 10, 10, 2) $v_8$

*T*:




When checking $v_4$, $v_3$ will be popped out and inserted into $L(q_3)$ since $v_3$ is not a descendant of $v_4$. After that $v_4$ will be pushed into the stack.

*ST*:

| | |
|---|---|
| $q_3$ | $v_3$ |
| $q_2$ | $v_2$ |
| $q_1$ | $v_1$ |

⇨

| | |
|---|---|
| $q_2$ | $v_4$ |
| $q_2$ | $v_2$ |
| $q_1$ | $v_1$ |

$B(q_1)$:

$B(\{q_2, q_5\})$:

(1, 10, 10, 2) $v_8$

$L(\{q_3, q_4\})$:

(1, 3, 3, 3) $v_3$

$B(\{q_3, q_4\})$:

(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

When checking $v_5$, it will be pushed into the stack.

$ST$:

| | |
|---|---|
| $q_3$ | $v_5$ |
| $q_2$ | $v_4$ |
| $q_2$ | $v_2$ |
| $q_1$ | $v_1$ |

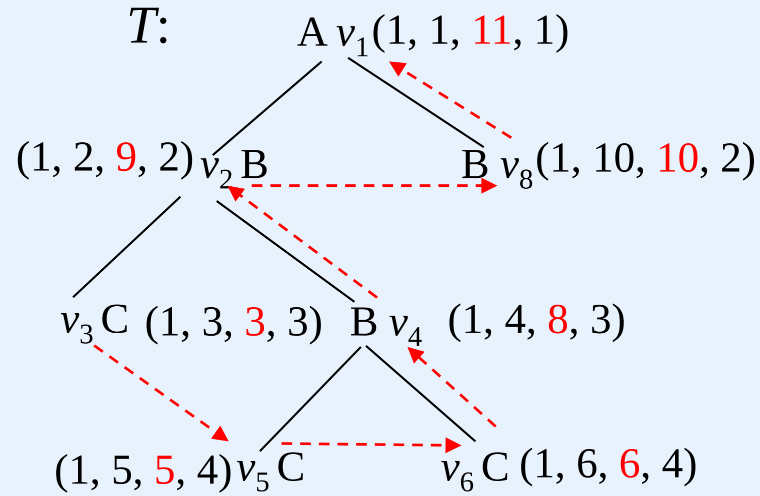
$B(q_1)$:

$B(\{q_2, q_5\})$:
(1, 10, 10, 2) $v_8$

$B(\{q_3, q_4\})$:
(1, 6, 6, 4) $v_6$

$L(\{q_3, q_4\})$:
(1, 3, 3, 3) $v_3$

$T$:




When checking $v_6$, $v_5$ will be popped out and inserted into $L(q_3)$ since $v_6$ is not a descendant of $v_5$. After that $v_6$ will be pushed into the stack.

$ST$:

| | |
|---|---|
| $q_3$ | $v_6$ |
| $q_2$ | $v_4$ |
| $q_2$ | $v_2$ |
| $q_1$ | $v_1$ |

$B(q_1)$:

$B(\{q_2, q_5\})$:
(1, 10, 10, 2) $v_8$

$B(\{q_3, q_4\})$:

$L(\{q_3, q_4\})$:
(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$

# Evaluation of Tree Pattern Queries

When checking $v_8$, $v_6$ will be popped out and inserted into $L(q_3)$ since $v_8$ is not a descendant of $v_6$. After that $v_6$ will be pushed into the stack.

*ST*:

| | |
|---|---|
| $q_2$ | $v_4$ |
| $q_2$ | $v_2$ |
| $q_1$ | $v_1$ |

$B(q_1)$:

$B(\{q_2, q_5\})$:

$B(\{q_3, q_4\})$:

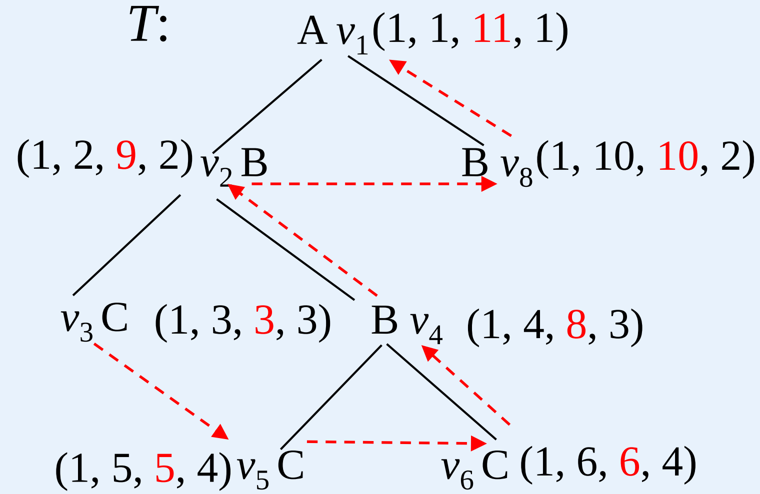
$L(\{q_3, q_4\})$:

(1, 3, 3, 3) $v_3$

(1, 5, 5, 4) $v_5$

(1, 6, 6, 4) $v_6$

*T*:




After that $v_4$ will be popped out and inserted into $L(q_2)$ since $v_8$ is not a descendant of $v_4$.

*ST*:

| | |
|---|---|
| $q_2$ | $v_2$ |
| $q_1$ | $v_1$ |

$B(q_1)$:

$B(\{q_2, q_5\})$:

$B(\{q_3, q_4\})$:

$L(\{q_3, q_4\})$:

(1, 3, 3, 3) $v_3$

(1, 5, 5, 4) $v_5$

(1, 6, 6, 4) $v_6$

$L(\{q_2, q_5\})$:

(1, 4, 8, 3) $v_4$

After that $v_2$ will be popped out and inserted into $L(q_2)$ since $v_8$ is not a descendant of $v_2$.

$ST$:

| | |
|---|---|
| $q_1$ | $v_1$ |

$B(q_1)$:

$B(\{q_2, q_5\})$:

$B(\{q_3, q_4\})$:

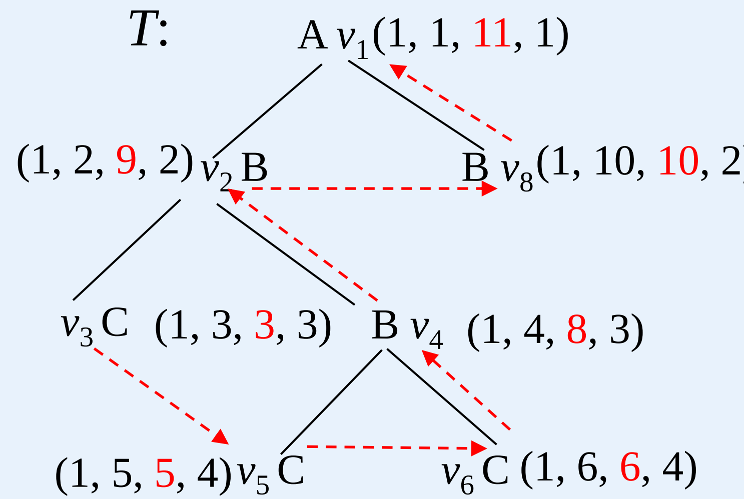
$L(\{q_3, q_4\})$:

(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

$L(\{q_2, q_5\})$:

(1, 4, 8, 3) $v_4$
(1, 2, 9, 2) $v_2$

$T$:




Since $v_8$ is a descendant of $v_1$, it will be pushed into the stack.

$ST$:

| | |
|---|---|
| $q_2$ | $v_8$ |
| $q_1$ | $v_1$ |

$B(q_1)$:

$B(\{q_2, q_5\})$:

$B(\{q_3, q_4\})$:

$L(\{q_3, q_4\})$:

(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

$L(\{q_2, q_5\})$:

(1, 4, 8, 3) $v_4$
(1, 2, 9, 2) $v_2$

After that $v_8$ will be popped out and inserted into $L(q_2)$.

$ST$:

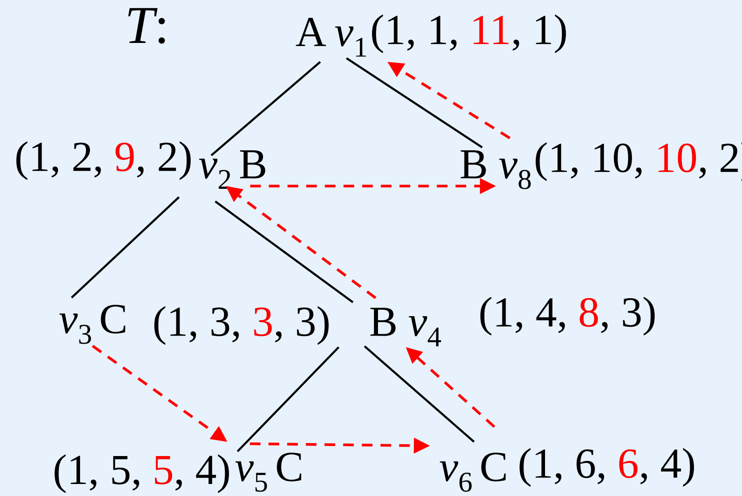
| | |
|---|---|
| $q_1$ | $v_1$ |

$B(q_1)$:    $B(\{q_2, q_5\})$:

$B(\{q_3, q_4\})$:

| $L(\{q_3, q_4\})$: | $L(\{q_2, q_5\})$: |
|---|---|
| (1, 3, 3, 3) $v_3$ | (1, 4, 8, 3) $v_4$ |
| (1, 5, 5, 4) $v_5$ | (1, 2, 9, 2) $v_2$ |
| (1, 6, 6, 4) $v_6$ | (1, 10, 10, 2) $v_8$ |

$T$:




After that $v_1$ will be popped out and inserted into $L(q_1)$.

$B(q_1)$:    $B(\{q_2, q_5\})$:

$ST$:

| | |
|---|---|
| | |

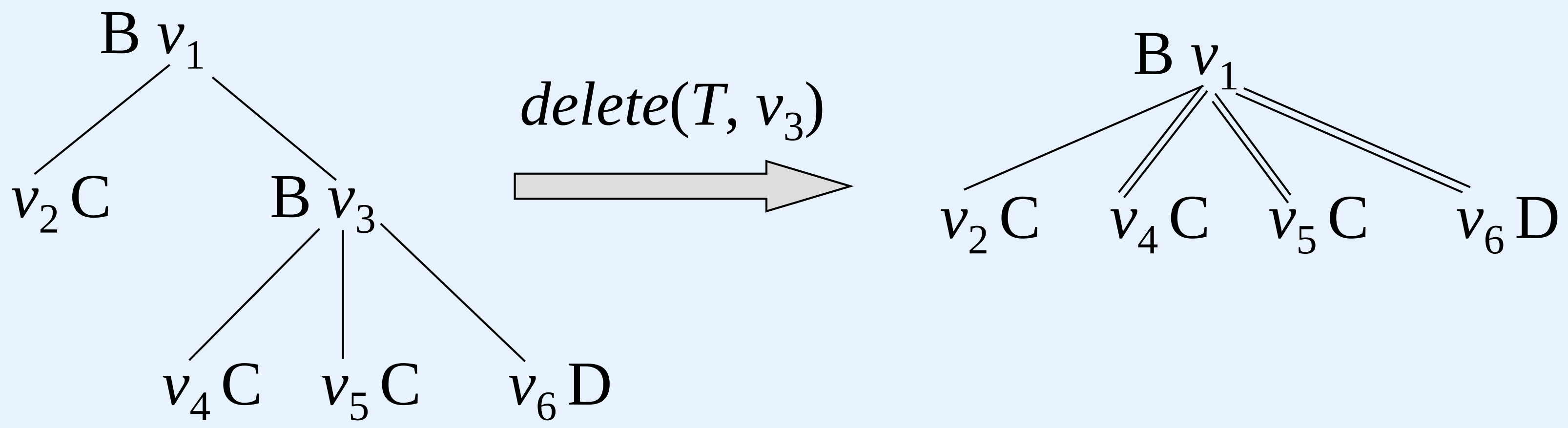
$B(\{q_3, q_4\})$:

| $L(\{q_3, q_4\})$: | $L(\{q_2, q_5\})$: |
|---|---|
| (1, 3, 3, 3) $v_3$ | (1, 4, 8, 3) $v_4$ |
| (1, 5, 5, 4) $v_5$ | (1, 2, 9, 2) $v_2$ |
| (1, 6, 6, 4) $v_6$ | (1, 10, 10, 2) $v_8$ |

| $L(q_1)$: |
|---|
| (1, 1, 11, 1) $v_1$ |

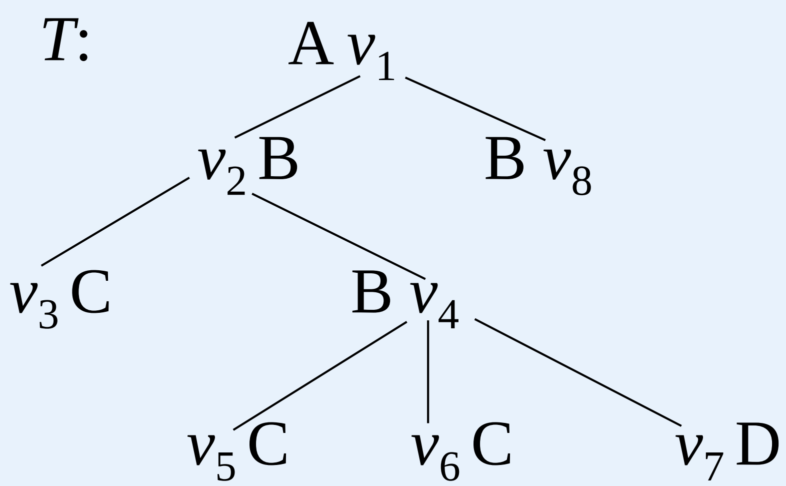
## Matching Subtrees

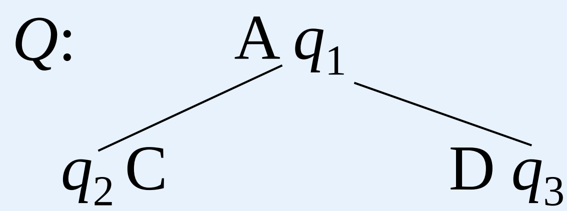
Let *T* be a tree and *v* be a node in *T* with parent node *u*. Denote by *delete*(*T*, *v*) the tree obtained from *T* by removing node *v*. The children of *v* become ‘descendant’ children of *u*.



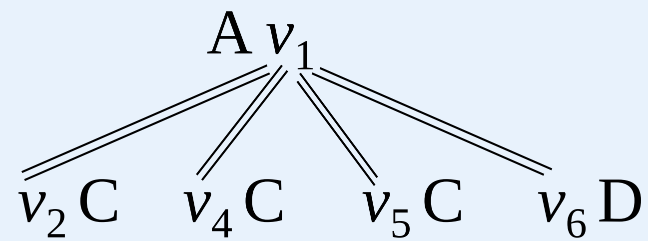
# Evaluation of Tree Pattern Queries

**Definition** (*matching subtrees*) A matching subtree *T'* of *T* with respect to a tree pattern *Q* is a tree obtained by a series of deleting operations to remove any node in *T*, which does not match any node in *Q*.



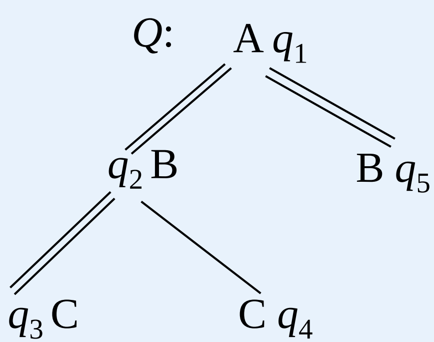




a matching subtree:



## Construction of Matching Subtree from Data Streams

- The algorithm given below handles the case when the streams contain nodes from a single XML document. (When the streams contain nodes from multiple documents, the algorithm is easily extended to test equality of DocId before manipulating the nodes in the streams.)
- It is simply an iterative process to access the nodes in $L(Q)$ one by one. Here, $L(Q) = L(\boldsymbol{q}_1) \cup L(\boldsymbol{q}_2) \; \dots \cup L(\boldsymbol{q}_k)$.




| $L(\boldsymbol{q}_1)$: A | $L(\boldsymbol{q}_2)$: B | $L(\boldsymbol{q}_3)$: C |
|---|---|---|
| (1, 1, 11, 1) $v_1$ | (1, 4, 8, 3) $v_4$ | (1, 3, 3, 3) $v_3$ |
| | (1, 2, 9, 2) $v_2$ | (1, 5, 5, 4) $v_5$ |
| | (1, 10, 10, 2) $v_8$ | (1, 6, 6, 4) $v_6$ |

$\boldsymbol{q}_1 = \{q_1\}$

$\boldsymbol{q}_2 = \{q_2, q_5\}$

$\boldsymbol{q}_3 = \{q_3, q_4\}$

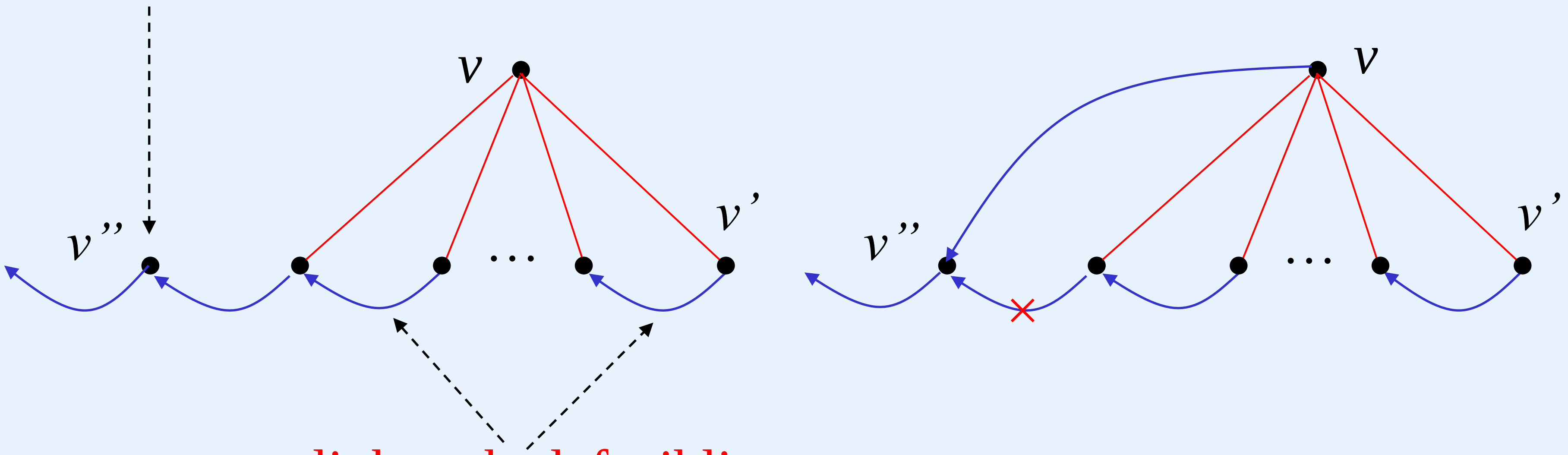
## Construction of Matching Subtree from Data Streams

It is simply an iterative process to access the nodes in $L(Q)$ (= $L(\boldsymbol{q}_1) \cup L(\boldsymbol{q}_2) \ldots \cup L(\boldsymbol{q}_k)$) one by one:

1. Identify a data stream $L(\boldsymbol{q})$ with the first element being of the minimal RightPos value. Choose the first element $v$ of $L(\boldsymbol{q})$. Remove $v$ from $L(\boldsymbol{q})$.
2. Generate a node for $v$.
3. If $v$ is not the first node, we do the following:
   Let $v'$ be the node chosen just before $v$.
   - If $v'$ is not a child (descendant) of $v$, create a link from $v$ to $v'$, called a *left-sibling* link and denoted as *left-sibling*($v$) = $v'$.


   - If $v'$ is a child (descendant) of $v$, we will first create a link from $v'$ to $v$, called a *parent* link and denoted as *parent*($v'$) = $v$. Then, we will go along the left-sibling chain starting from $v'$ until we meet a node $v''$ which is not a child (descendant) of $v$. For each encountered node $u$ except $v''$, set *parent*($u$) $\leftarrow v$. Finally, set *left-sibling*($v$) $\leftarrow v''$.

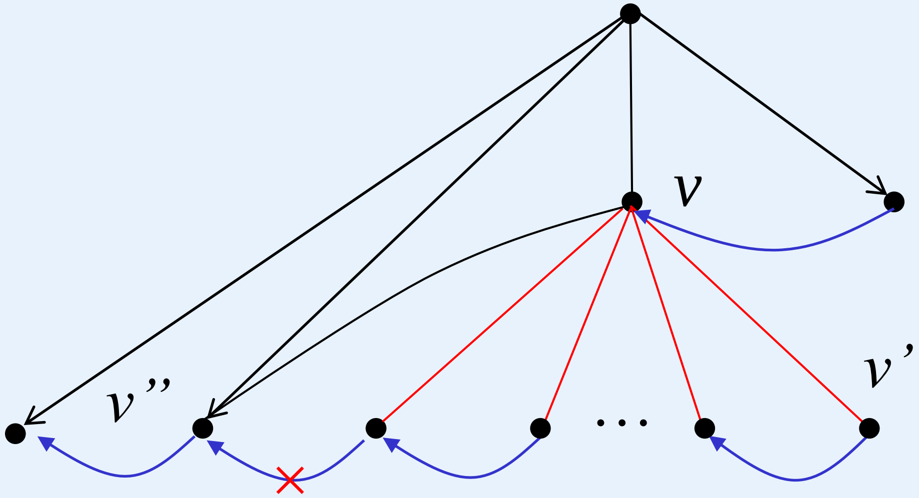
*v''* is not a child of *v*.




link to the left sibling

In the figure, we show the navigation along a left-sibling chain starting from *v'* when we find that *v'* is a child (descendant) of *v*. This process stops whenever we meet *v''*, a node that is not a child (descendant) of *v*. The figure shows that the left-sibling link of *v* is set to *v''*, which is previously pointed to by the left-sibling Link of *v*'s left-most child.

# Evaluation of Tree Pattern Queries



**Algorithm** *matching-tree-construction*($L(Q)$) (* $L(Q) = L(\boldsymbol{q_1}) \cup L(\boldsymbol{q_2})$ … $\cup L(\boldsymbol{q_k})$ *)
input: all data streams $L(Q)$.
output: a matching subtree $T'$.
**begin**
**1.** **repeat until** each $L(\boldsymbol{q})$ in $L(Q)$ becomes empty
2. { identify $\boldsymbol{q}$ such that the first element $v$ of $L(\boldsymbol{q})$ is of the minimal RightPos value; remove $v$ from $L(\boldsymbol{q})$;
3. generate node $v$;
**4.** **if** $v$ is not the first node created **then**
5. { **let** $v'$ be the node generated just before $v$;
**6.** **if** $v'$ is not a child (descendant) of $v$ **then**
**7.** *Left-sibling*($v$) $\leftarrow v'$; (*generate a left-sibling link.*)
8. {$v'' \leftarrow v'$, $w \leftarrow v'$, (*$v''$ and $w$ are two temporary variables.*)
**9.** **while** $v''$ is a child (descendant) of $v$ **do**
10. { *parent*($v''$) $\leftarrow v$; (*generate a parent link. Also, indicate whether $v''$ is a /-child or a //-child.*)
11. $w \leftarrow v''$; $v'' \leftarrow$ *left-sibling*($v''$);
12. }
14. *left-sibling*($v$) $\leftarrow v''$; } }
15. }
**end**

$L(q_1)$ - A:
(1, 1, 11, 1) $v_1$

$L(\{q_3, q_4\})$ - C:
(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

$L(\{q_2, q_5\})$ - B:
(1, 4, 8, 3) $v_4$
(1, 2, 9, 2) $v_2$
(1, 10, 10, 2) $v_8$

- In the above algorithm, for each chosen *v* from a *L*(***q***), a node is created.
- At the same time, a left-sibling link of *v* is established, pointing to the node *v'* that is generated before *v*, if *v'* is not a child (descendant) of *v* (see line 7).
- Otherwise, we go into a **while**-loop to travel along the left-sibling chain starting from *v'* until we meet a node *v''* which is not a child (descendant) of *v*.
- During the process, a parent link is generated for each node encountered except *v''*. (See lines 9 - 13.) Finally, the left-sibling link of *v* is set to be *v''* (see line 14).

**Example** Consider the following data stream $L(\boldsymbol{q})$'s:

# Data Streams – $L(\boldsymbol{q})$'s

$L(q_1)$:

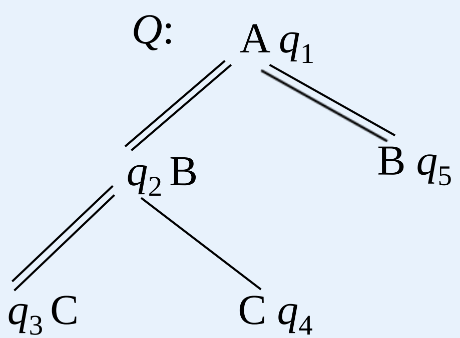
(1, 1, 11, 1) $v_2$

$L(\{q_2, q_5)$:

(1, 4, 8, 3) $v_4$

(1, 2, 9, 2) $v_2$

(1, 10, 10, 2) $v_8$

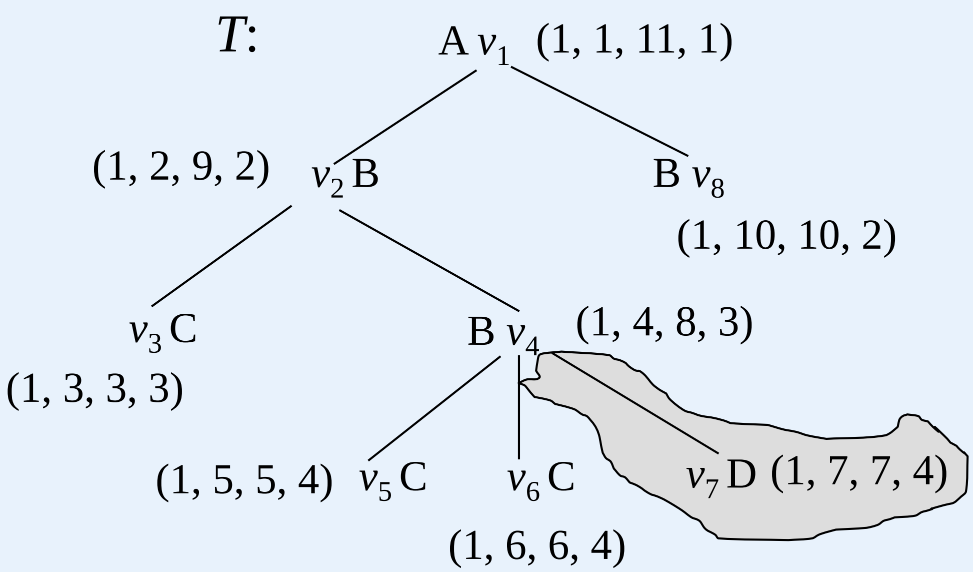



$L(q_3,q_4)$:

(1, 3, 3, 3) $v_3$

(1, 5, 5, 4) $v_5$

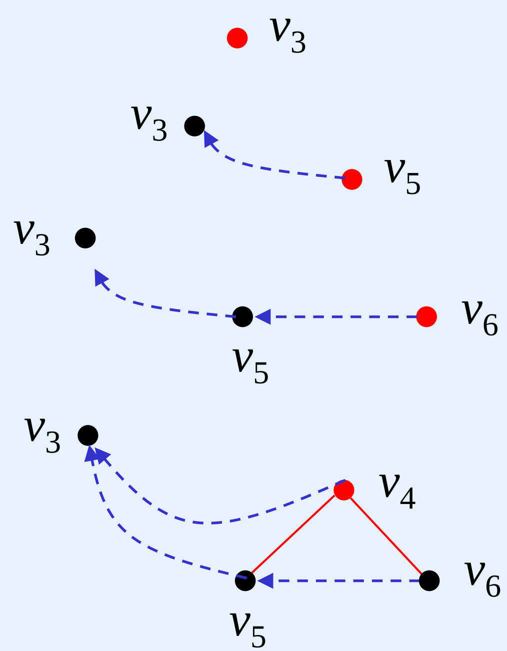
(1, 6, 6, 4) $v_6$




The data streams are sorted by (DocID, RightPos).

**Example** (continued) $L(\boldsymbol{q}) = \{v_1\}$, $L(\boldsymbol{q}') = \{v_4, v_2, v_8\}$, $L(\boldsymbol{q}'') = \{v_3, v_5, v_6\}$, where $\boldsymbol{q} = \{q_1\}$, $\boldsymbol{q}' = \{q_2, q_5\}$, $\boldsymbol{q}'' = \{q_3, q_4\}$. Applying the above algorithm to the data streams, we generate a series of data structures as shown below.

| | $v$ with the least RightPos: | Generated data structure: |
|---|---|---|
| step 1: | $v_3$ | |
| step 2: | $v_5$ | |
| step 3: | $v_6$ | |
| step 4: | $v_4$ | |




$L(q_1)$:

(1, 1, 11, 1) $v_2$

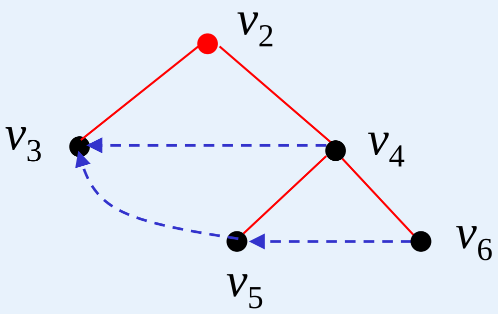
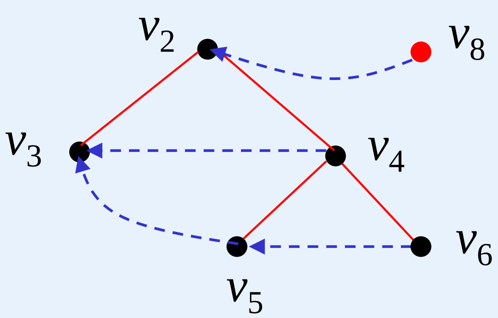
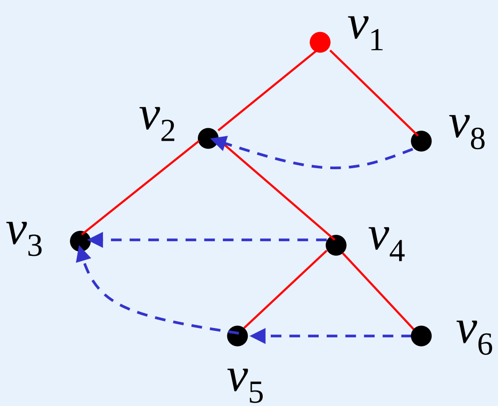
$L(\{q_2, q_5\})$:

(1, 4, 8, 3) $v_4$
(1, 2, 9, 2) $v_2$
(1, 10, 10, 2) $v_8$

$L(q_3,q_4)$:

(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

# Evaluation of Tree Pattern Queries

| $v$ with the least RightPos: | Generated data structure: |
|---|---|
| step 5: $v_2$ |  |
| step 6: $v_8$ |  |
| step 7: $v_1$ |  |

$L(q_1)$:

(1, 1, 11, 1) $v_{\mathbf{2}}$

$L(\{q_2, q_5\})$:

(1, 4, 8, 3) $v_{\mathbf{4}}$

(1, 2, 9, 2) $v_{\mathbf{2}}$

(1, 10, 10, 2) $v_{\mathbf{8}}$

$L(q_3,q_4)$:

(1, 3, 3, 3) $v_{\mathbf{3}}$

(1, 5, 5, 4) $v_{\mathbf{5}}$

(1, 6, 6, 4) $v_{\mathbf{6}}$

# Evaluation of Tree Pattern Queries

The *time complexity* of this process is easy to analyze.

- First, we notice that each quadruple in all the data streams is accessed only once.
- Secondly, for each node in $T'$, all its child nodes will be visited along a left-sibling chain for a second time.

So we get the total time

$$O(|D|\cdot|Q|) + \sum_i d_i = O(|D|\cdot|Q|) + O(|T'|) = O(|D|\cdot|Q|),$$

where $D$ is the largest data stream and $d_i$ represents the outdegree of node $v_i$ in $T'$.

During the process, for each encountered quadruple, a node $v$ will be generated. Associated with this node have we at most two links (a left-sibling link and a parent link). So the used extra space is bounded by $O(|T'|)$.

**Proposition 1** Let $T$ be a document tree. Let $Q$ be a tree pattern. Let $L(Q) = \{L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)\}$ be all the data streams with respect to $Q$ and $T$, where each $\boldsymbol{q}_i$ ($1 \leq i \leq l$) is a subset of sorted query nodes of $Q$, which share the same data stream. Algorithm *matching-tree-construction*($L(Q)$) generates the matching subtree $T'$ of $T$ with respect to $Q$ correctly.

*Proof*. Denote $L = |L(\boldsymbol{q}_1)| + ... + |L(\boldsymbol{q}_l)|$. We prove the proposition by induction on $L$.

*Basis*. When $L = 1$, the proposition trivially holds.

*Induction hypothesis*. Assume that when $L = k$, the proposition holds.

*Induction step*. We consider the case when $L = k + 1$. Assume that all the quadruples in $L(Q)$ are $\{u_1, ..., u_k, u_{k+1}\}$ with RightPos($u_1$) < RightPos($u_2$) < ... < RightPos($u_k$) < RightPos($u_{k+1}$).

The algorithm will first generate a tree structure $T_k$ for $\{u_1, ..., u_k\}$. In terms of the induction hypothesis, $T_k$ is correctly created. It can be a tree or a forest. If it is a forest, all the roots of the subtrees in $T_k$ are connected through left-sibling links. When we meet $v_{k+1}$, we consider two cases:

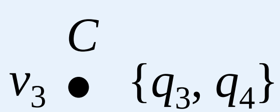
i) $v_{k+1}$ is an ancestor of $v_k$,

ii) $v_{k+1}$ is to the right of $v_k$.

In case (i), the algorithm will generate an edge $(v_{k+1}, v_k)$, and then travel along a left-sibling chain starting from $v_k$ until we meet a node $v$ which is not a descendant of $v_{k+1}$. For each node $v'$ encountered, except $v$, an edge $(v_{k+1}, v')$ will be generated. Therefore, $T_{k+1}$ is correctly constructed. In case (ii), the algorithm will generate a left-sibling link from $v_{k+1}$ to $v_k$. It is obviously correct since in this case $v_{k+1}$ cannot be an ancestor of any other node. This completes the proof.

# Tree pattern matching

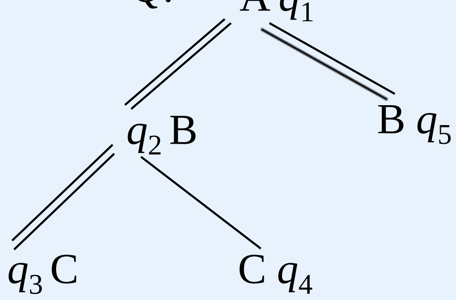
We observe that during the reconstruction of a matching subtree *T'*, we can also associate each node *v* in *T'* with a ***query node stream*** *QS*(*v*). That is, each time we choose a *v* with the least RightPos value from a data stream *L*(***q***), we will insert all the query nodes in ***q*** into *QS*(*v*).



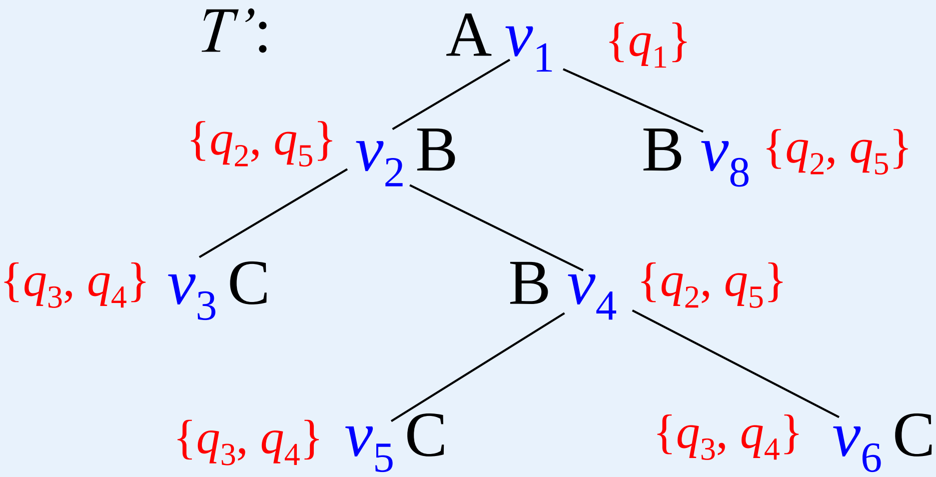

$L(q_3,q_4)$:

(1, 3, 3, 3) $v_3$
(1, 5, 5, 4) $v_5$
(1, 6, 6, 4) $v_6$

Q:




T':

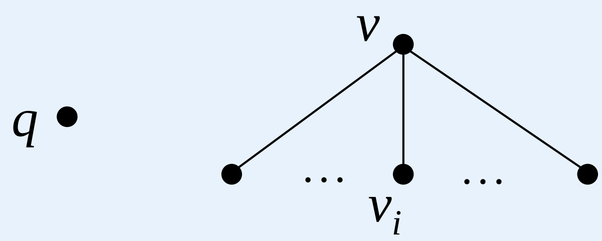


# Evaluation of Tree Pattern Queries

If we check, before a $q$ is inserted into the corresponding $QS(v)$, whether $Q[q]$ (the subtree rooted at $q$) can be imbedded into $T'[v]$, we get in fact an algorithm for tree pattern matching. The challenge is how to conduct such a checking efficiently.

- For this purpose, we associate each $q$ in $Q$ with a variable, denoted $\chi(q)$.
- During the process, $\chi(q)$ will be dynamically assigned a series of values $a_0, a_1, ..., a_m$ for some $m$ in sequence, where $a_0 = \phi$ and $a_i$'s ($i = 1, ..., m$) are different nodes of $T'$.

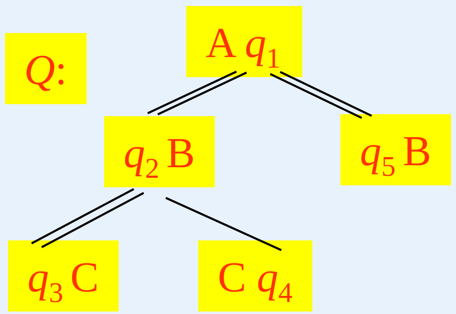
$\chi(q) = v$ indicates that $Q[q]$ matches $T'[v_i]$ for some child $v_i$ of $v$.

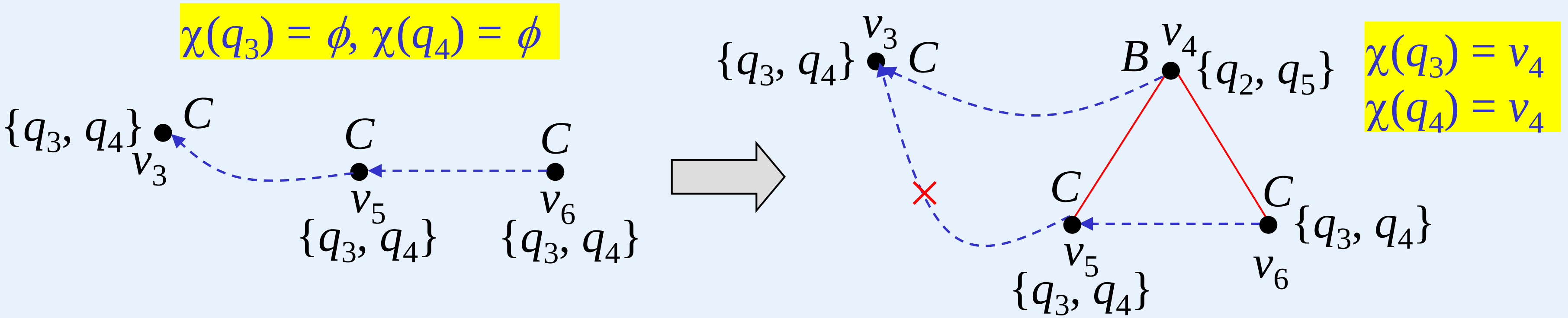



If $Q[q]$ matches $T'[v_i]$, $\chi(q)$ is set to be $v$. Some time later, when $q$ is checked again, $\chi(q)$ will be changed.

For this purpose, we associate each $q$ in $Q$ with a variable, denoted $\chi(q)$. During the process, $\chi(q)$ will be dynamically assigned a series of values $a_0, a_1, \ldots, a_m$ for some $m$ in sequence, where $a_0 = \phi$ and $a_i$'s ($i = 1, \ldots, m$) are different nodes of $T'$.

- Initially, $\chi(q)$ is set to $a_0 = \phi$.
- $\chi(q)$ will be changed from $a_{i-1}$ to $a_i = v$ ($i = 1, \ldots, m$) when the following conditions are satisfied.
  i) $v$ is the node currently encountered.
  ii) $q$ appears in $QS(u)$ for some child node $u$ of $v$.
  iii) $q$ is a //-child, or
  $q$ is a /-child, and $u$ is a /-child of $v$ with $label(u) = label(q)$.

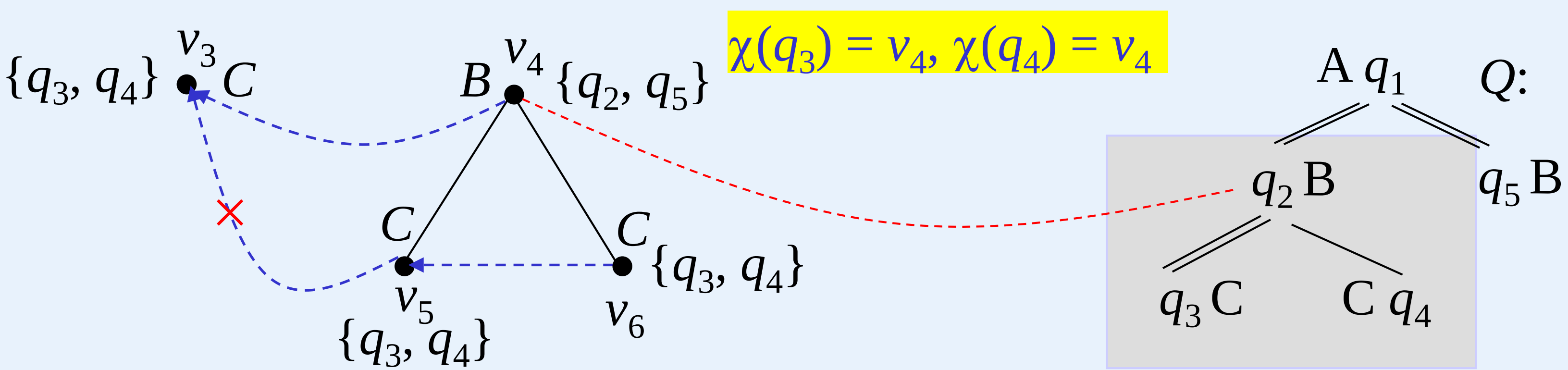


# Evaluation of Tree Pattern Queries

Then, each time before we insert $q$ into $QS(v)$, we will do the following checking:

1. Let $q_1$, ..., $q_k$ be the child nodes of $q$.
2. If for each $q_i$ ($i$ = 1, ..., $k$), $\chi(q_i)$ is equal to $v$ and $label(v) = label(q)$, insert $q$ into $QS(v)$.

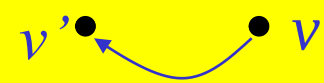



Since we search both $T$ and $Q$ bottom-up, the above checking guarantees that for any $q \in QS(v)$, $T'[v]$ contains $Q[q]$.

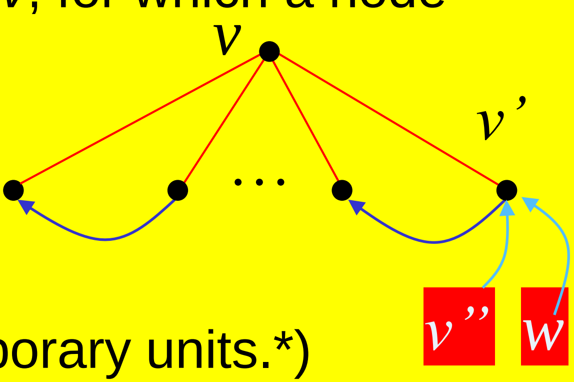
The following algorithm ***unordered-tree-matching*(*L*(*Q*))** is similar to Algorithm *matching-tree-construction*( ), by which

- a quadruple is removed in turn from the data streams $L(\boldsymbol{q})$'s and a node $v$ for it is generated and inserted into the matching subtree.
- It will be checked for each $q \in \boldsymbol{q}$ whether $q$ can be inserted into $QS(v)$.

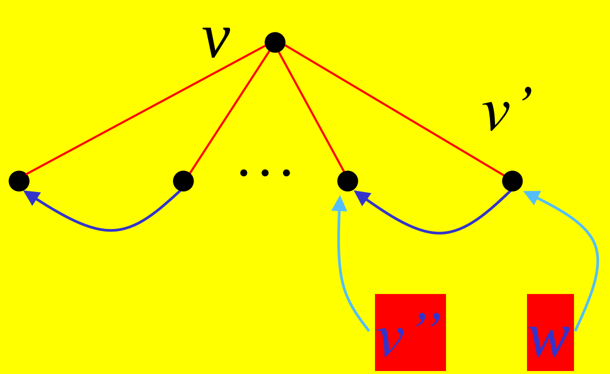
**Algorithm** *unordered-tree-matching*($L$($Q$))
input: all data streams $L$($Q$).
output: a matching subtree $T'$ of $T$, $D_{root}$ and $D_{output}$.
**begin**
1. **repeat until** each $L$($\boldsymbol{q}$) in $L$($Q$) becomes empty {
2. identify $\boldsymbol{q}$ such that the first node $v$ of $L$($\boldsymbol{q}$) is of the minimal RightPos value; remove $v$ from $L$($\boldsymbol{q}$); generate node $v$;
3. **if** $v$ is the first node created **then**
4. {$QS$($v$) $\leftarrow$ *subsumption-check*($v$, $\boldsymbol{q}$); }




5. **else**
6. { let $v'$ be the quadruple chosen just before $v$, for which a node is constructed;
7. **if** $v'$ is not a child (descendant) of $v$ **then**
8. {*left-sibling*($v$) $\leftarrow$ $v'$ ; }
9. **else**
10. {$v''$ $\leftarrow$ $v'$; $w$ $\leftarrow$ $v'$; (*$v''$ and $w$ are two temporary units.*)



```
      while v" is a child (descendant) of v do
        {parent(v") ← v; (*generate a parent link. Also, indicate
          whether v" is a /-child or a //-child.*)
          for each q in QS(v") do {    (*For each q in QS(v"), compute χ(q).*)
          if ((q is a //-child) or (q is a /-child and v" is a /-child and
              label(q) = label(v")))
          then χ(q) ← v;}
          w ← v"; v" ← left-sibling(v");
          remove left-sibling(w);
        }
        left-sibling(v) ← v";
      }
    q ← subsumption-check(v, q);
    let v_1, ..., v_j be the child nodes of v;
    q' ← merge(QS(v_1), ..., QS(v_j));
    remove QS(v_1), ..., QS(v_j);
    QS(v) ← merge(q, q');
  } }
end
```




By *merge*($QS(v_1)$, $QS(v_2)$), we will put $QS(v_1)$ and $QS(v_2)$ together, but remove all those nodes which are descendants of some other nodes.

*subsumption-check*($v$, $\boldsymbol{q}$) – for each $q$ in $\boldsymbol{q}$, check whether $Q[q]$ can be embedded in $T[v]$.

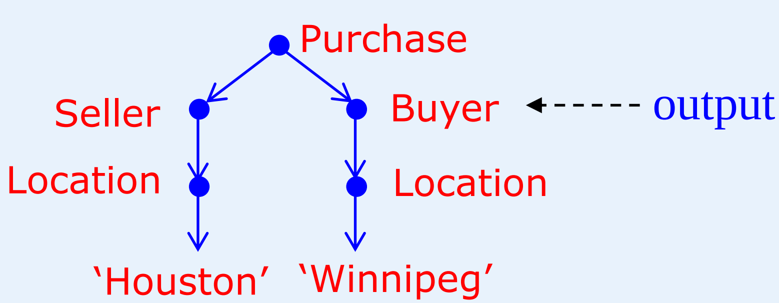
Two data structures are used:

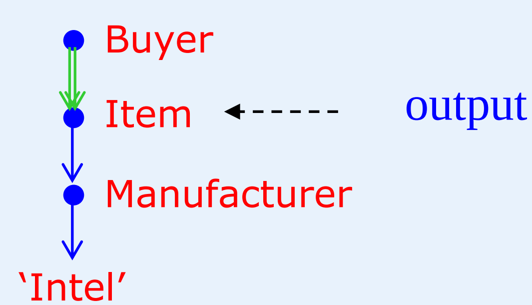
$D_{root}$ - a subset of document nodes $v$ such that $Q$ can be embedded in $T[v]$.

$D_{output}$ - a subset of document nodes $v$ that is in a subtree containing $Q$, and matches $q_{output}$, where $q_{output}$ is the output node of $Q$.

Q1: /Purchase[Seller[Loc=‘Boston’]]/
Buyer[Loc = ‘New York’]

Q2: /Purchase//Item[Manufacturer = ‘Intel’]



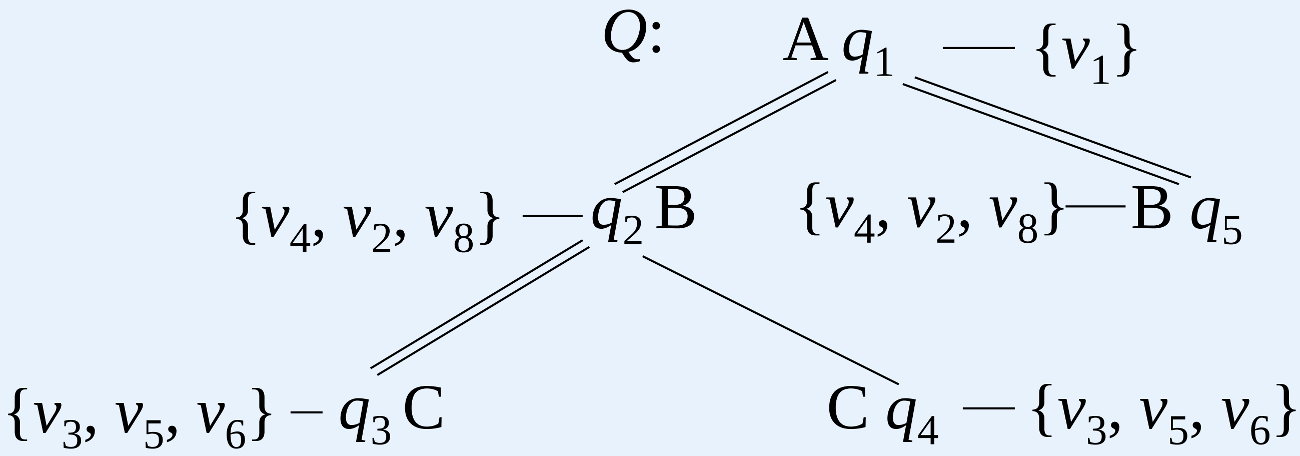
*subsumption-check*($v$, $\boldsymbol{q}$) – for each $q$ in $\boldsymbol{q}$, check whether $Q[q]$ can be embedded in $T[v]$.

**Function** *subsumption-check*($v$, $\boldsymbol{q}$) (*$v$ satisfies the node name test
1. $QS \leftarrow \phi$; at each $q$ in $\boldsymbol{q}$.*)
2. **for** each $q$ in $\boldsymbol{q}$ do {
3. let $q_1$, ..., $q_j$ be the child nodes of $q$.
4. **if** for each /-child $q_i$ $\chi(q_i) = v$ and for each //-child $q$ $\chi(q_i)$ is subsumed by $v$ **then**
**5.** {$QS \leftarrow QS \cup \{q\}$;
6. **if** $q$ is the root of $Q$ **then**
**7.** $D_{root} \leftarrow D_{root} \cup \{v\}$;
8. **if** $q$ is the output node **then** $D_{output} \leftarrow D_{output} \cup \{v\}$; } }
9. return $QS$;

**end**

If $q$ is a leaf node and $label(q) = label(v)$, do $QS \leftarrow QS \cup \{q\}$.

**Example.**



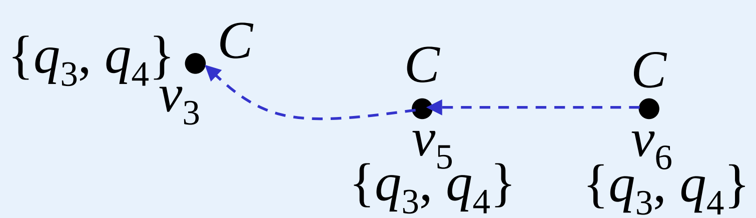

| $B(q_1)$ - A: |
|---|
| (1, 1, 11, 1) $v_1$ |

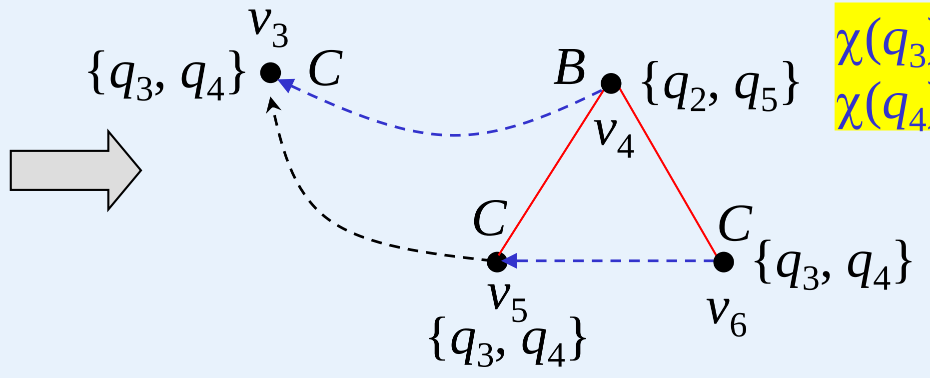
| $B(\{q_2, q_5\})$ - B: |
|---|
| (1, 2, 9, 2) $v_2$ |
| (1, 4, 8, 3) $v_4$ |
| (1, 10, 10, 2) $v_8$ |

| $B(\{q_3, q_4\})$ - C: |
|---|
| (1, 3, 3, 3) $v_3$ |
| (1, 5, 5, 4) $v_5$ |
| (1, 6, 6, 4) $v_6$ |

The data streams are sorted by (DocID, RightPos).

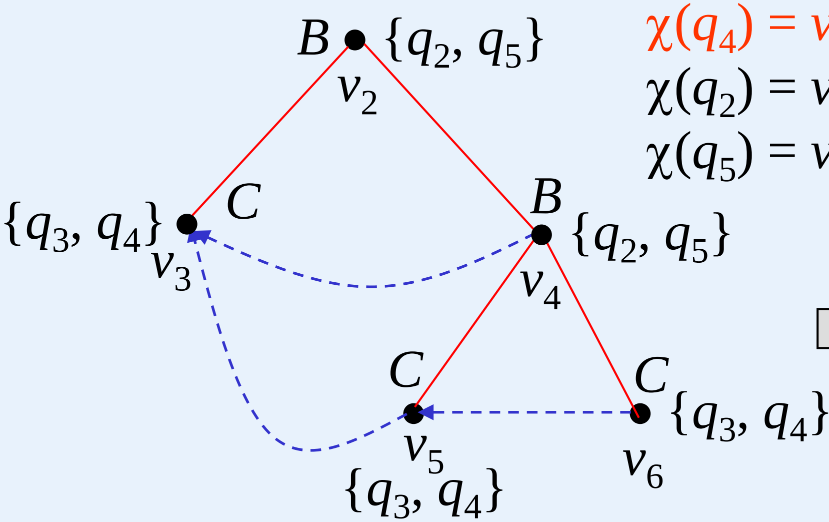
$\chi(q_3) = \phi, \chi(q_4) = \phi$



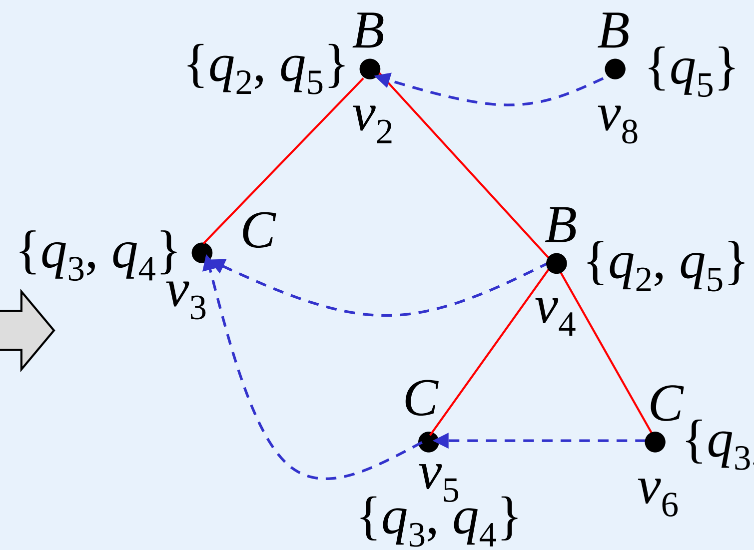




$\chi(q_3) = v_4$
$\chi(q_4) = v_4$

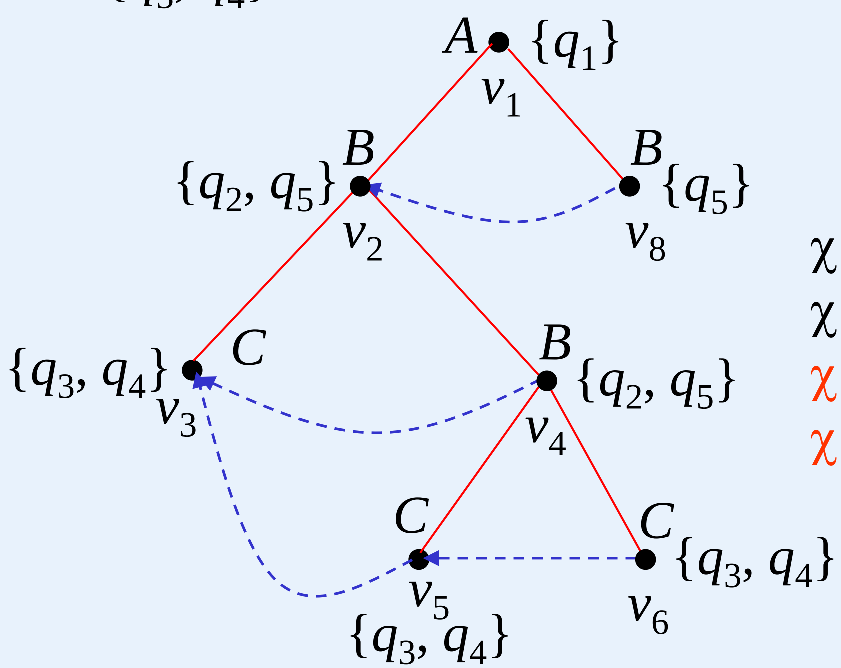
# Evaluation of Tree Pattern Queries



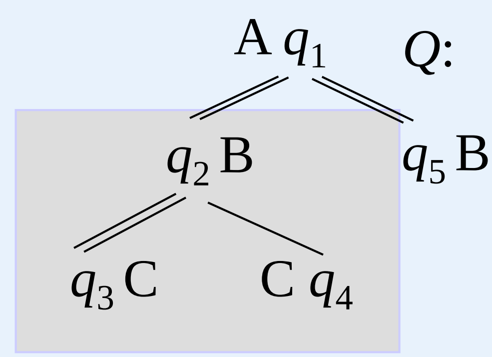

$\chi(q_3) = v_2$
$\chi(q_4) = v_2$
$\chi(q_2) = v_2$
$\chi(q_5) = v_2$




$\chi(q_3) = v_2$
$\chi(q_4) = v_2$
$\chi(q_2) = v_2$
$\chi(q_5) = v_2$



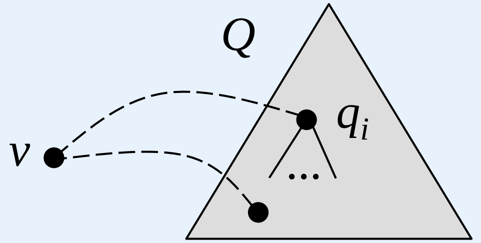

$\chi(q_3) = v_2$
$\chi(q_4) = v_2$
$\chi(q_2) = v_1$
$\chi(q_5) = v_1$



The time complexity of the algorithm can be divided into three parts:

1. The first part is the time spent on accessing $L(q)$'s. Since each element in a $L(q)$ is visited only once, this part of cost is bounded by $O(|D|\cdot|Q|)$, where $D$ is the largest data stream associated with a query node.
2. The second part is the time used for constructing $QS(v)$'s. For each node $v$ in the matching subtree, we need $O(\sum_i c_i)$ time to do the task, where $c_i$ is the outdegree of $q_i$, which matches $v$. So this part of cost is bounded by

$$O(\sum_v \sum_i c_i) \leq O(|D| \cdot \sum_i^{|Q|} c_i\ ) = O(|D|\cdot|Q|).$$




3. The third part is the time for establishing $\chi(q)$ values, which is the same as the second part since for each $q$ in a $QS(v)$ its $\chi(q)$ value is assigned only once.

The space overhead of the algorithm is easy to analyze.

- Besides the data streams, each node in the matching tree needs a parent link and a left-sibling link to facilitate the subtree reconstruction, and an *QS* to calculate $\chi(q)$ values.
- However, the *QS*($v$) data structure is removed once its parent node is created. In addition, each node in the tree pattern is associated with a $\chi$ value. So the extra space requirement is bounded by

$$O(leaf_{T'}\cdot|Q| + |T'|) + O(|Q|) = O(leaf_{T'}\cdot|Q| + |T'|),$$

where $leaf_{T'}$ represents the number of the leaf nodes of $T'$.

$leaf_{T'}\cdot|Q|$ - the upper bound on the size of all *QS*($v$)'s
$T'$ – the matching subtree

## Ordered Tree Matching

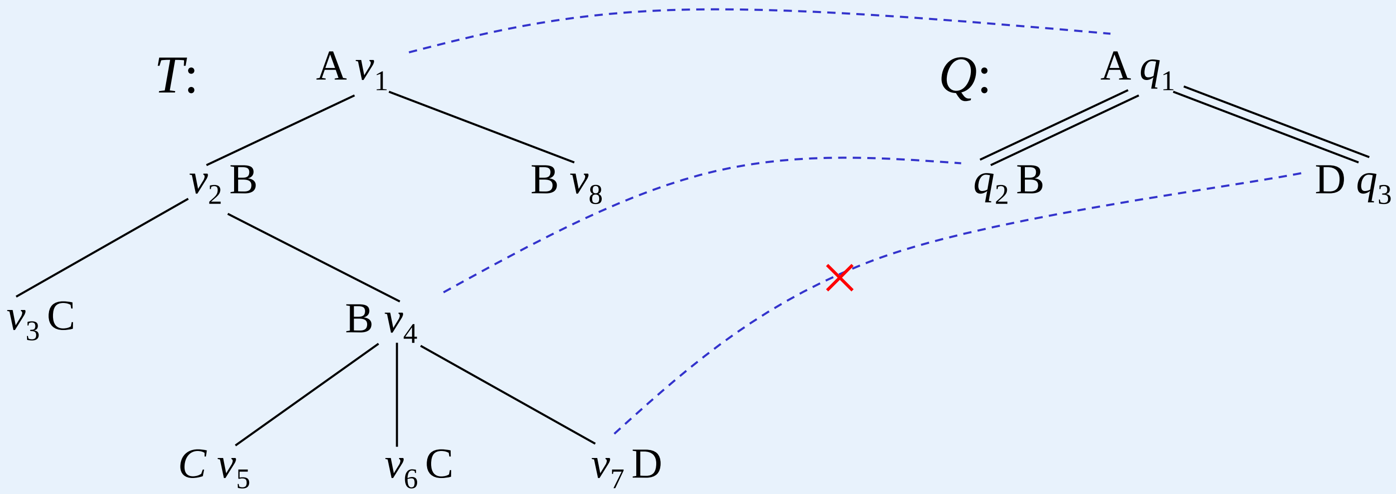
**Definition** An embedding of a tree pattern $Q$ into an XML document $T$ is a mapping $f$: $Q \rightarrow T$, from the nodes of $Q$ to the nodes of $T$, which satisfies the following conditions:

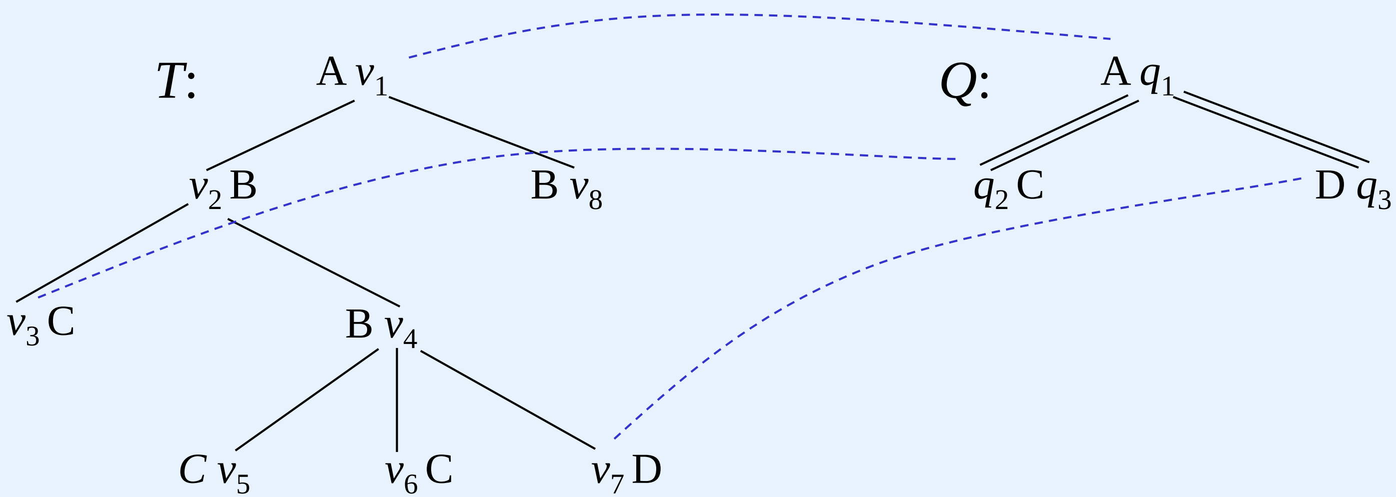
*(i)* *Preserve node type*: For each $u \in Q$, $u$ and $f(u)$ are of the same type. (or more generally, $u$'s predicate is satisfied by $f(u)$.)

*(ii)* *Preserve child/descendant-child relationships*: If $u \rightarrow v$ in $Q$, then $f(v)$ is a child of $f(u)$ in $T$; if $u \Rightarrow v$ in $Q$, then $f(v)$ is a descendant of $f(u)$ in $T$.

*(iii)* *Preserve left-to-right order*: For any two siblings $v_1$, $v_2$ in $Q$, if $v_1$ is to the left of $v_2$, then $f(v_1)$ is to the left of $f(v_2)$ in $T$.

# Evaluation of Tree Pattern Queries



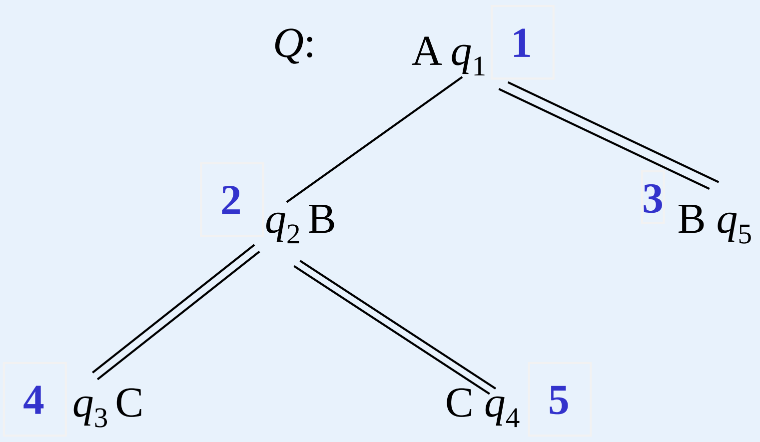



# Algorithm for Ordered Tree Matching Based on two concepts:

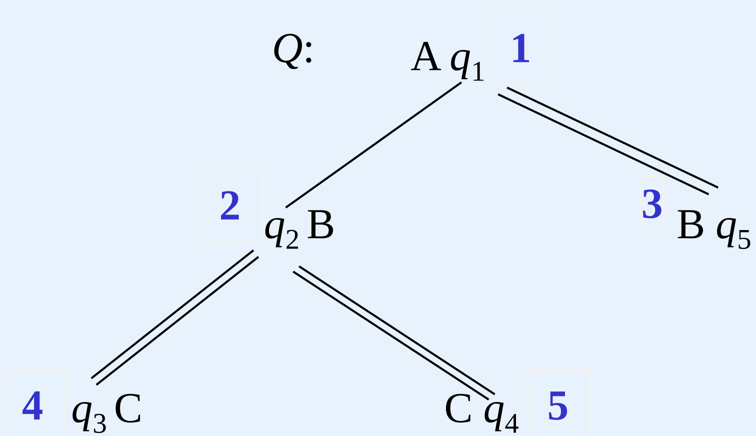
- **Breadth-first numbering**
- **Linked list of quadruples**

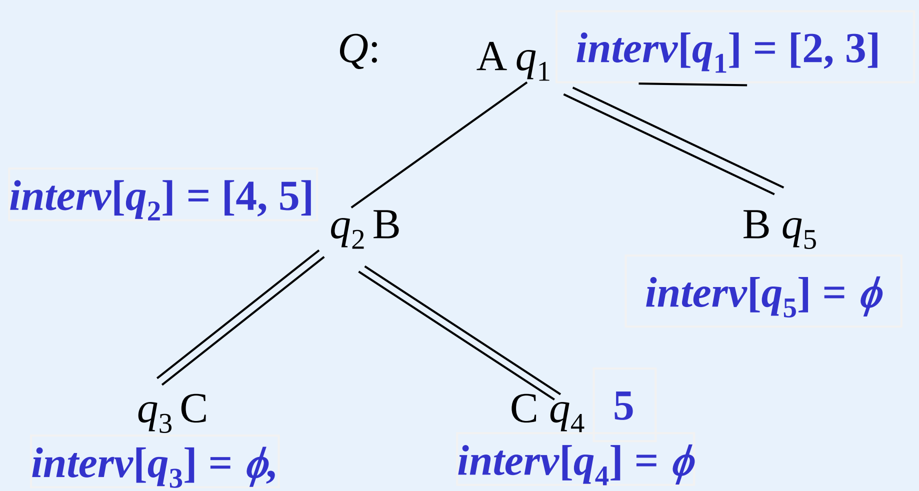
## Breadth-first numbering

- In order to capture the order of siblings, we create a new number for each node *q* in *Q* by searching *Q* in the breadth-first fashion. Such a number is then called a *breadth-first number* and denoted as *bf*(*q*). As illustrated in the following figure (see the numbers in boldface), they represent the left-to-right order of siblings in a simple way.



- Then, we use *interval*($q$) to represent an interval covering all the breadth-first numbers of $q$'s children.
- For example, for $Q$ shown in the following figure, we have *interval*($q_1$) = [2, 3] and *interval*($q_2$) = [4, 5]. (If no confusion will be caused, we will also use $q$ and *bf*($q$) interchangeably in the following discussion.)



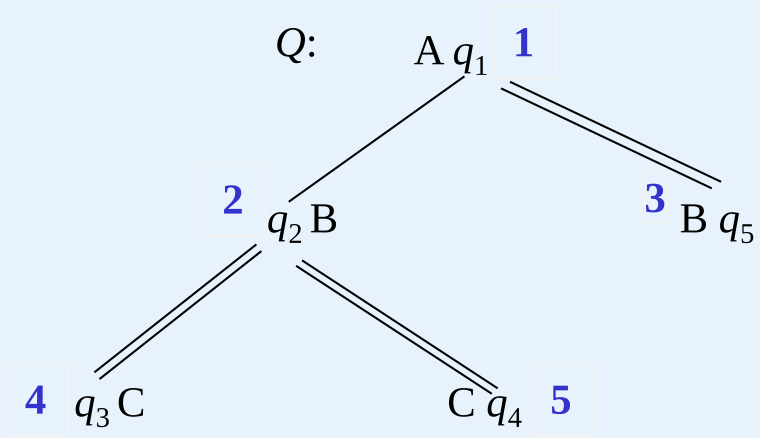



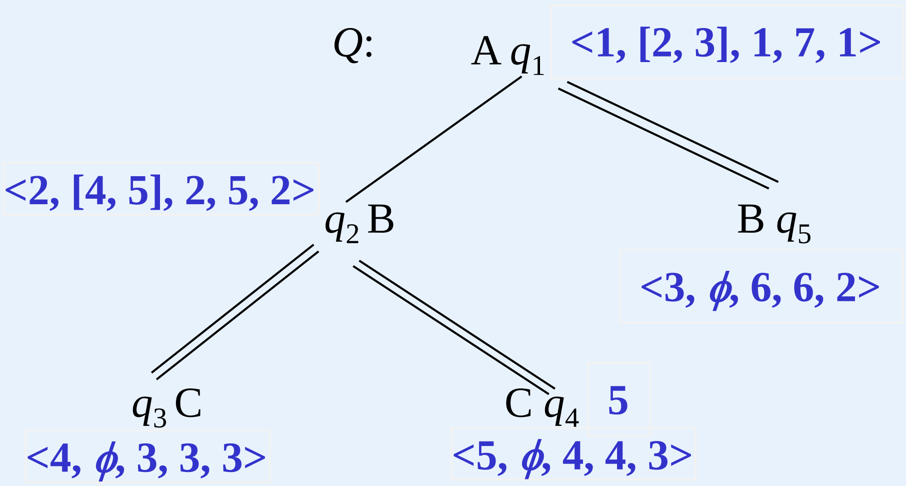
Next, we associate each $q$ with a tuple:

$g(q) = \langle bf(q), interval(q), \text{LeftPos}(q), \text{RightPos}(q), \text{LevelNum}(q)\rangle$,

as shown in the following figure.

These tuples can be generated in $O(|Q|)$ time and used to facilitate the computation.

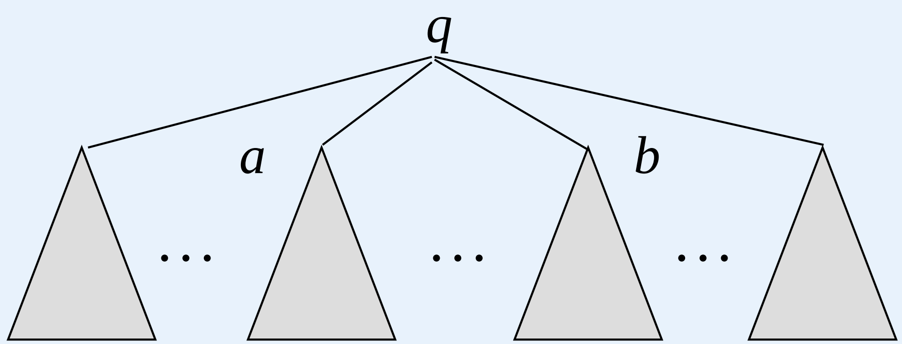


**Linked list of quadruples**

- When checking the tree embedding of $Q$ in $T'$, we will associate each generated node $v$ in $T'$ with a linked list $A_v$ to record what subtrees in $Q$ can be embedded in $T'[v]$.
- For this purpose, the intervals associated with query nodes will be used.
- Each entry in $A_v$ is a quadruple $e = (q, interval, L, R)$, where $q$ is a node in $Q$, $interval = [a, b] \subseteq interval(q)$ (for some $a \leq b$), $L$ = LeftPos($a$) and $R$ = RightPos($b$). Here, we use $a$ and $b$ to refer to the nodes with the breadth-first numbers $a$ and $b$, respectively.
- An entry $e = (q, [a, b], L, R)$ in $A_v$ indicates that the subtrees rooted respectively at $a, a + 1, \ldots, b$ can be embedded in $T'[v]$.

A quadruple associated with a node *v* in *T* represents a set of subtrees (in *Q*[*q*]) rooted respectively at *a*, *a* + 1, ..., *b* (i.e., a set of subtrees rooted at a set of consecutive breadth-first numbers) which can be embedded in *T*[*v*].




quadruple: $e = (q, \mathit{interval}, L, R)$

# Evaluation of Tree Pattern Queries

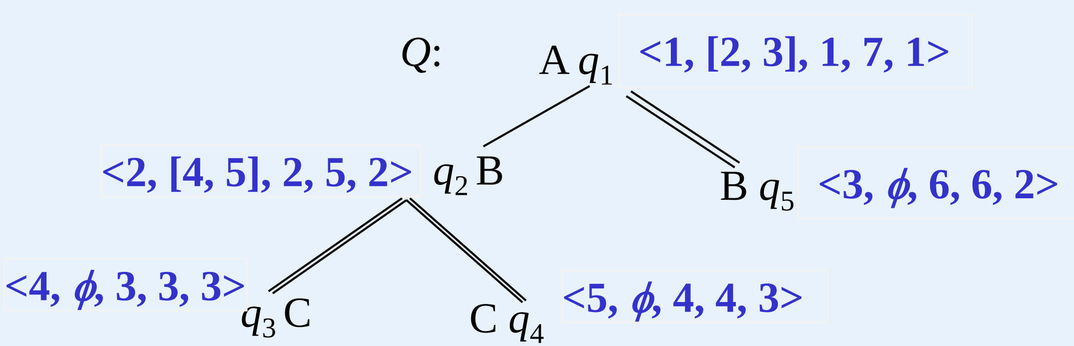
Before we discuss how such entries in $A_v$'s are generated, we first specify two conditions, which must be satisfied by them. We say, a query node $q$ is subsumed by a pair $(L, R)$ if $L \leq$ LeftPos($q$) and $R \geq$ RightPos($q$).

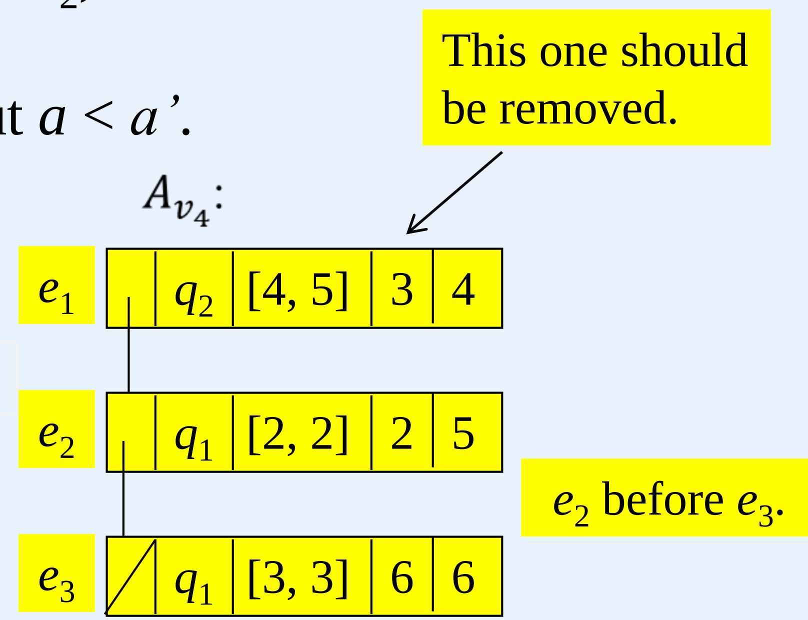
i) For any two entries $e_1$ and $e_2$ in $A_v$, $e_1.q$ is not subsumed by $(e_2.L, e_2.R)$, nor is $e_2.q$ subsumed by $(e_1.L, e_1.R)$. In addition, we require that if $e_1.q = e_2.q$, $e_1.interval \not\subset e_2.interval$ and $e_2.interval \not\subset e_1.interval$.

ii) For any two entries $e_1$ and $e_2$ in $A_v$ with $e_1.interval = [a, b]$ and $e_2.interval = [a', b']$, if $e_1$ appears before $e_2$, then

RightPost($e_1.q$) $<$ RightPost($e_2.q$) or

RightPost($e_1.q$) $=$ RightPost($e_2.q$) but $a < a'$.



- Condition (i) is used to avoid redundancy due to the following lemma.

**Lemma 1** Let $q$ be a node in $Q$. Let $[a, b]$ be an interval. If $q$ is subsumed by (LeftPos($a$), RightPos($b$)), then there exists an integer $0 \leq i \leq b - a$ such that $bf(q)$ is equal to $a + i$ or $q$ is an descendant of $a + i$.

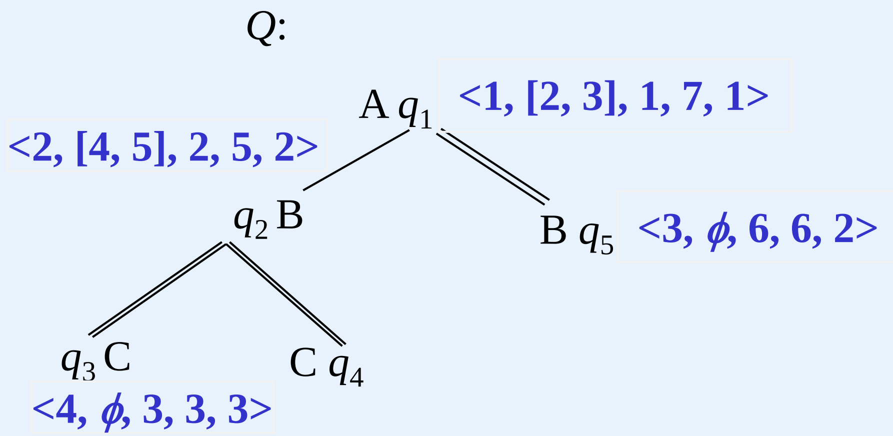
*Proof*. The proof is trivial.

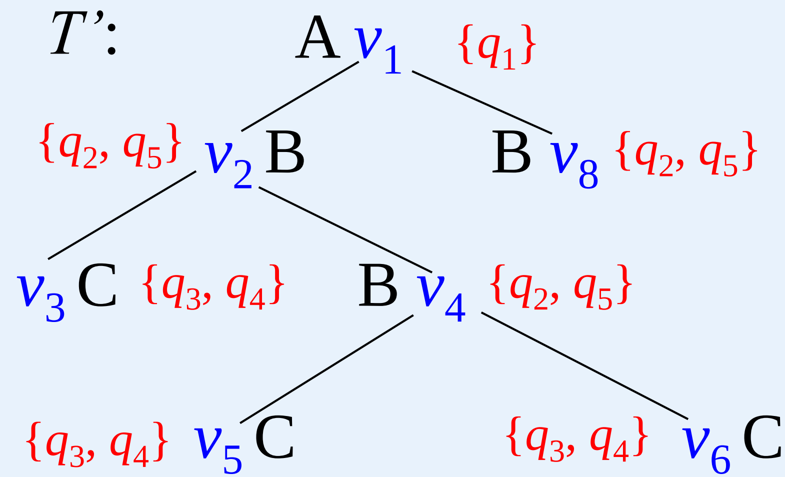
So $A_v$ keeps only quadruples which represent pairwise non-covered subtrees by imposing condition (i).

- Condition (ii) is met if the nodes in $Q$ are checked along their increasing RightPos values. It is because in such an order the parents of the checked nodes must be non-decreasingly sorted by theiRightPos values.

Since we explore $Q$ bottom-up, condition (ii) is always satisfied.

# Evaluation of Tree Pattern Queries




$A_{v_5}$:

| | | | | | |
|---|---|---|---|---|---|
| $e_1$ | | $q_2$ | [4, 4] | 3 | 3 |
| $e_2$ | | $q_2$ | [5, 5] | 4 | 4 |

$A_{v_6}$ is the same as $A_{v_5}$.

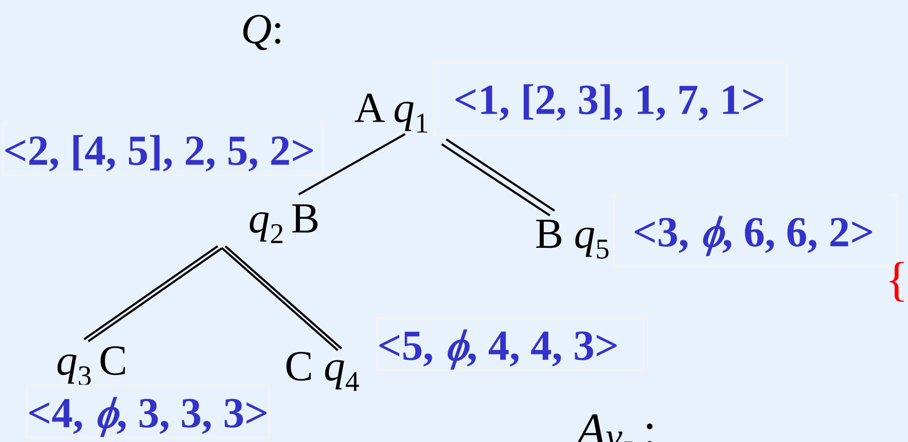
$A_{v_4}$:

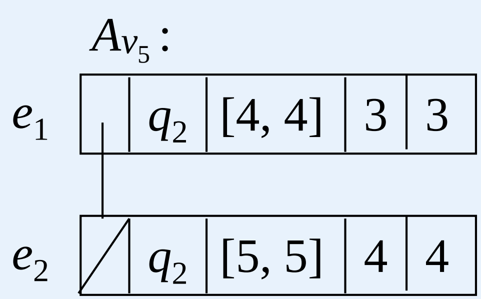
| | | | | | |
|---|---|---|---|---|---|
| $e_1'$ | | $q_2$ | [4, 5] | 3 | 4 |
| $e_2'$ | | $q_1$ | [2, 2] | 2 | 5 |
| $e_3'$ | | $q_1$ | [3, 3] | 6 | 6 |

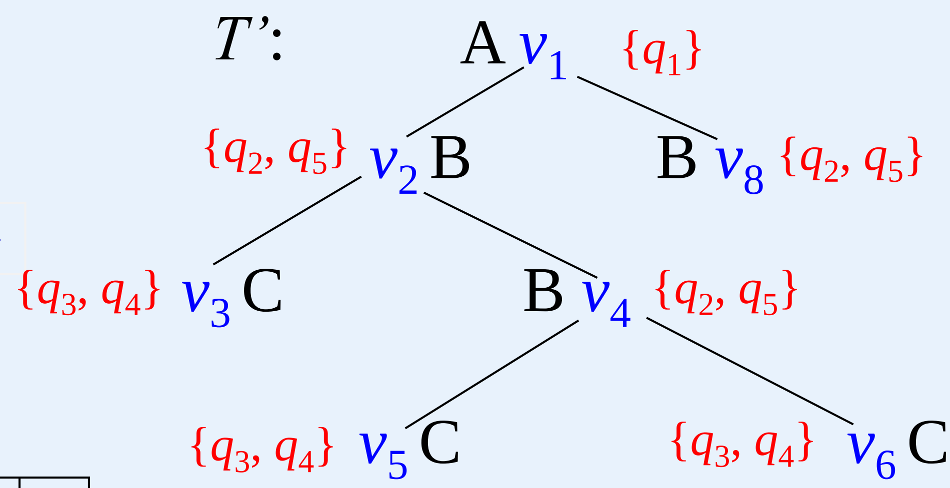
$A_{v_4}$:

| | | | | |
|---|---|---|---|---|
| | $q_1$ | [2, 2] | 2 | 5 |
| | $q_1$ | [3, 3] | 6 | 6 |

- The first linked list is created for $v_5$ in *T'* when it is generated and checked against $q_3$ and $q_4$ in *Q*. Since both $q_3$ and $q_4$ are leaf nodes, $T'[v_5]$ is able to embed either $Q[q_3]$ or $Q[q_4]$ and so we have two entries $e_1$ and $e_2$ in $A_{v_5}$. Note that $bf(q_3) = 4$ and $bf(q_3) = 5$. In addition, each of them is a child of $q_2$. Thus, we have $e_1.q = e_2.q = q_2$.

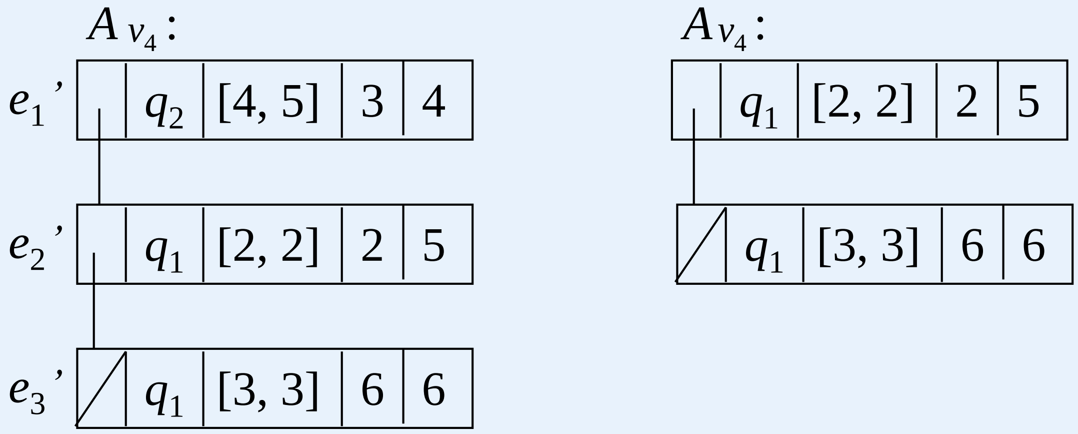



$A_{v_6}$ is the same as $A_{v_5}$.

- The linked list for $v_4$ contains three entries $e_1'$, $e_2'$ and $e_3'$. Special attention should be paid to $e_1'$. Its *interval* is [4, 5], showing that $T'[v_4]$ is able to embed both $Q[q_3]$ and $Q[q_4]$. In this case, $e_1'.L$ is set to 3 and $e_1'.R$ to 4.
- Since $e_1'.q = q_2$ is subsumed by $(e_2'.L, e_2'.R) = (2, 5)$, the entry will be removed, as shown by the third linked list.



**Main Algorithm**

With the linked lists associated with the nodes in *T'*, the embedding of a subtree *Q*[*q*] in *T'*[*v*] can be checked very efficiently by running the following procedure.

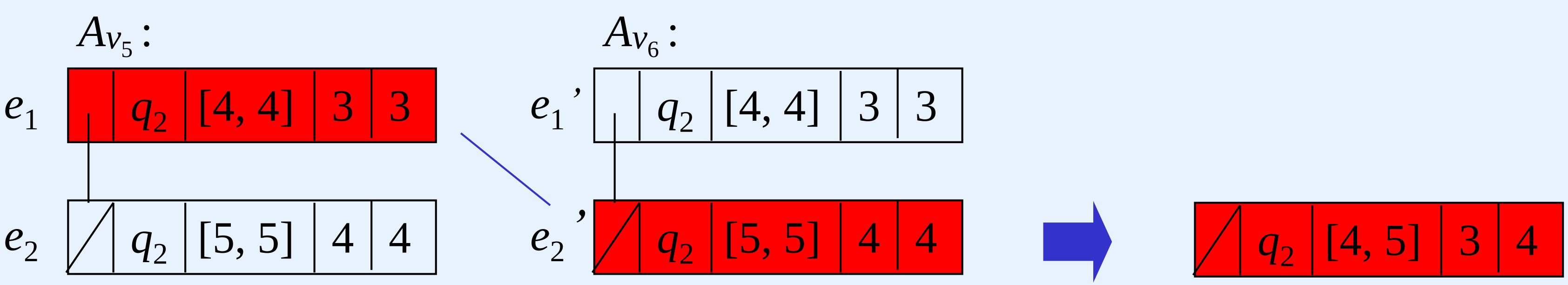
1. Explore *T'* bottom-up.
2. For each *v* with children $v_1$, ..., $v_k$ in *T'*, explore *Q* bottom-up, doing (i), (ii) and (iii) below:
   i) let *q* be the current query node;
   ii) check whether *T'*[*v*] contains *Q*[*q*] by using $A_{v_i}$'s ($i$ = 1, ..., $k$);
   iii) add new entries into $A_v$ according to the results obtained in (ii).

In the above process, we search both *T'* and *Q* bottom-up; and for each encountered pair (*v*, *q*) we check whether *Q*[*q*] can be embedded in *T'*[*v*] by using the linked lists associated with *v*'s children. The results of the checking is then recorded in the linked list associated with *v*.

While the above general process is straightforward, it is very challenging to manipulate $A_v$'s efficiently. In the following, we elaborate this process.

First, we define a simple operation over two intervals $[a, b]$ and $[a', b']$, which share the same parent:

$$[a, b]\ \Delta\ [a', b'] = \begin{cases} [a, b'], & \text{If } a \leq a' \leq b + 1,\ b \leq b'; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$



The general operation to merge two linked list is described below.

1. Let $A_1$ and $A_2$ be two linked list associated with the first two child nodes of a node $v$ in $T'$, which is being checked against $q$ with $label(v) = label(q)$.
2. Scan both $A_1$ and $A_2$ from the beginning to the end. Let $e_1$ (from $A_1$) and $e_2$ (from $A_2$) be the entries encountered. We will perform the following checkings.
   - If RightPos($e_2.q$) > RightPos($e_1.q$), $e_1 \leftarrow next(e_1)$.
   - If RightPos($e_2.q$) < RightPos($e_1.q$), then $e_2' \leftarrow e_2$; insert $e_2'$ into $A_1$ just before $e_1$; $e_2 \leftarrow next(e_2)$.
   - If RightPos($e_2.q$) = RightPos($e_1.q$), then we will compare the intervals in $e_1$ and $e_2$. Let $e_1.interval = [a, b]$. Let $e_2.interval = [a', b']$.

     If $a' > b + 1$, then $e_1 \leftarrow next(e_1)$.

     If $a \leq a' \leq b + 1$ and $b \leq b'$, then replace $e_1.interval$ with $[a, b] \Delta [a', b']$ in $A_1$; $e_1$.RightPost $\leftarrow$ RightPos($b'$); $e_1 \leftarrow next(e_1)$; $e_2 \leftarrow next(e_2)$.

     If $[a', b'] \subseteq [a, b]$, then $e_2 \leftarrow next(e_2)$.

     If $a' < a$, then $e_2' \leftarrow e_2$; insert $e_2'$ into $A_1$ just before $e_1$; $e_2 \leftarrow next(e_2)$.
3. If $A_1$ is exhausted, all the remaining entries in $A_2$ will be appended to the end of $A_1$.

- The result of the above process is stored in $A_1$, denoted as

  $merge(A_1, A_2)$.

- We further define

  $merge(A_1, ..., A_k) = merge(merge(A_1, ..., A_{k-1}), A_k)$,

  where $A_1, ..., A_k$ are the linked lists associated with $v$'s child nodes: $v_1, ..., v_k$, respectively.

If in $merge(A_1, ..., A_k)$ there exists an $e$ such that $e.interval = interval(q)$, $T'[v]$ embeds $Q[q]$.

# Evaluation of Tree Pattern Queries

- For the merging operation described above, we require that the entries in a linked list are sorted. That is, all the entries *e* are in the order of increasing RightPos(*e*.*q*) values; and for those entries with the same RightPos(*e*.*q*) value their intervals are 'from-left-to-right' ordered.
- Such an order is obtained by searching *Q* bottom-up (or say, in the order of increasing RightPos values) when checking a node *v* in *T'* against the nodes in *Q*. Thus, no extra effort is needed to get a sorted linked list.
- Moreover, if the input linked lists are sorted, the output linked lists must also be sorted.

# Evaluation of Tree Pattern Queries

**Algorithm** *tree-embedding*($L(Q)$)
Input: all data streams $L(Q)$.
Output: $S_v$'s, which show the tree embedding.
**begin**
1. **repeat until** each $L(\boldsymbol{q})$ in $L(Q)$ become empty
2. {identify $\boldsymbol{q}$ such that the first element $v$ of $L(\boldsymbol{q})$ is of the minimal RightPos value; remove $v$ from $L(\boldsymbol{q})$;
3. generate node $v$; $A_v \leftarrow \phi$;
4. let $v_1$, ..., $v_k$ be the children of $v$.
5. $B \leftarrow merge(A_{v_1}, ..., A_{v_k})$;
6. **for** each $q \in \boldsymbol{q}$ **do** { (*nodes in $\boldsymbol{q}$ are sorted.*)
7. **if** $q$ is a leaf **then** {$S_v \leftarrow S_v \cup \{q\}$; }
8. **else** (*$q$ is an internal node.*)

| $L(q_1)$ - A: |
|---|
| (1, 1, 11, 1) $v_1$ |

| $L(\{q_3, q_4\})$ - C: |
|---|
| (1, 3, 3, 3) $v_3$ |
| (1, 5, 5, 4) $v_5$ |
| (1, 6, 6, 4) $v_6$ |

| $L(\{q_2, q_5\})$ - B: |
|---|
| (1, 4, 8, 3) $v_4$ |
| (1, 2, 9, 2) $v_2$ |
| (1, 10, 10, 2) $v_8$ |

9. {**if** there exists $e$ in $B$ such that $e.interval = interval(q)$
10. **then** $S_v \leftarrow S_v \cup \{q\}$; }
11.}
12. **for** each $q \in S_v$ **do** {
13. append ($q$'s parent, [$bf(q)$, $bf(q)$)], $q$.LeftPos, $q$.RightPos) to the
    end of $A_v$;}
14. $A_v \leftarrow merge(A_v, B)$; Scan $A_v$ to remove subsumed entries;
15. remove all $A_{v_i}$'s; }
16.}
**end**

In the above algorithm, *left*-*sibling* links should be generated to reconstruct a tree structure as in the algorithm *matching*-*tree*-*construction*( ). However, such technical details are omitted for simplicity.

- In Algorithm *tree-embedding*( ), the nodes in *T'* is created one by one as done in Algorithm *matching-tree-construction*( ).
- But for each node *v* generated for an element from a $L(\boldsymbol{q})$, we will first merge all the linked lists of their children and store the output in a temporary variable *B* (see line 5).
- Then, for each $q \in \boldsymbol{q}$, we will check whether there exists an entry *e* such that *e.interval* = *interval*(*q*) (see lines 8 - 9). If it is the case, we will construct an entry for *q* and append it to the end of the linked list *A*v (see lines 12 - 13).
- The final linked list for *v* is established by executing line 14.
- Afterwards, all the $A_{v_i}$'s (for *v*'s children) will be removed since they will not be used any more (see line 15).

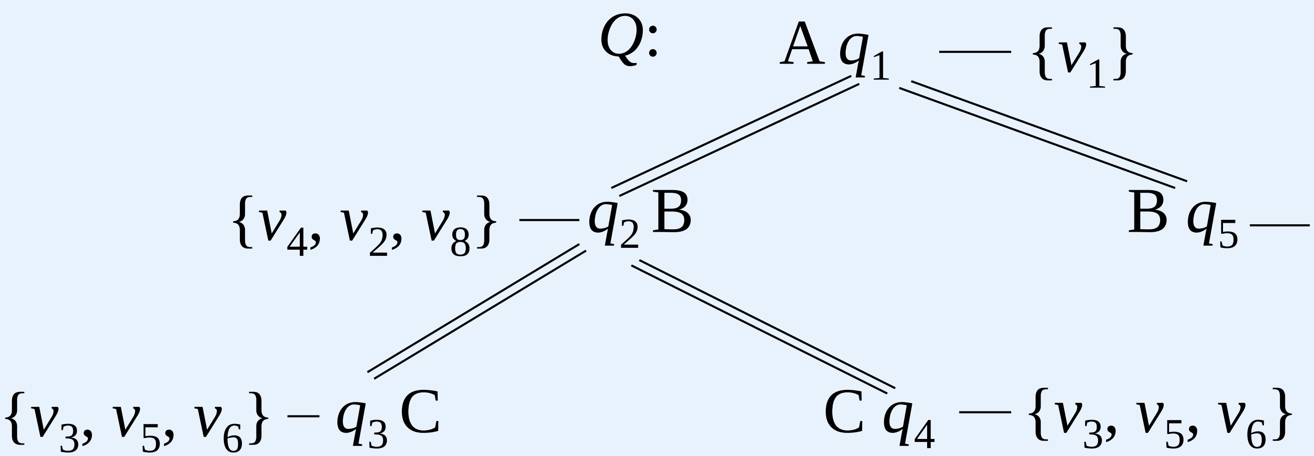
Finally, we point out that the above merging operation can be used only for the case that $Q$ contains no /-edges. In the presence of both //-edges and /-edges, the linked lists should be slightly modified as follows.

i) Let $q_j$ be a /-child of $q$ with $bf(q_j) = a$. Let $A_i$ be a linked list associated with $v_i$ (a child of $v$) which contains an entry $e$ with $e.interval = [c, d]$ such that $c \leq a$ and $a \leq d$.

ii) If $label(q_j) = label(v_i)$ and $v$i is a /-child of $v$, $e$ needn't be changed. Otherwise, $e$ will be replaced with two entries:

- ($e.q$, [$c$, $a$ - 1], LeftPos($c$), LeftPos($a$ - 1)), and
- ($e.q$, [$a$ + 1, $d$], LeftPos($a$ + 1), LeftPos($d$)).

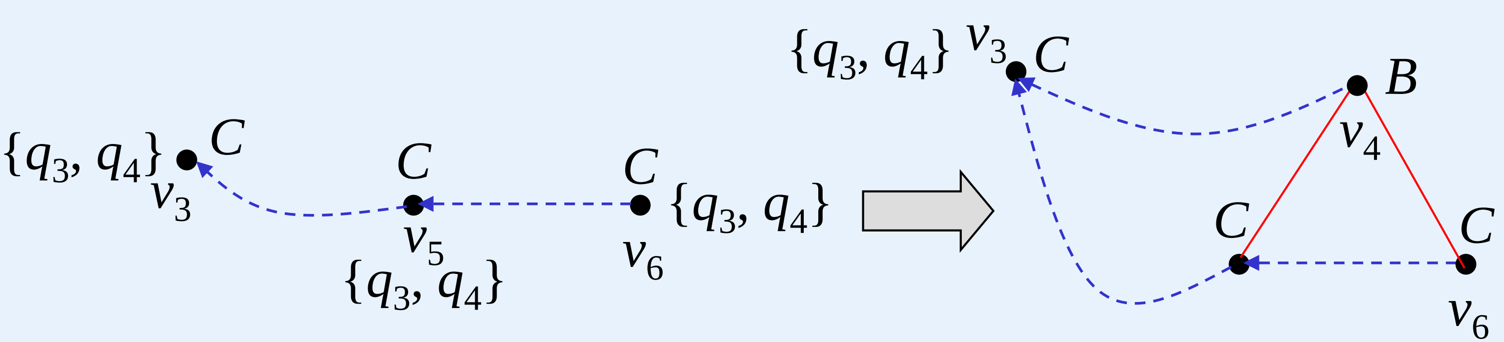
**Example.**




The data streams are sorted by (DocID, RightPos).

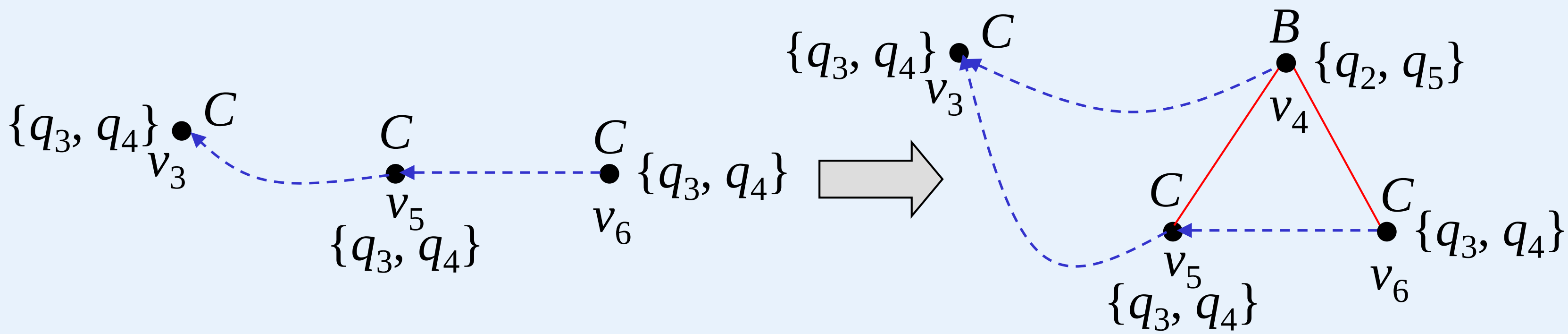
| $L(q_1)$ - A: |
|---|
| (1, 1, 11, 1) $v_1$ |

| $L(\{q_3,q_4\})$ - C: |
|---|
| (1, 3, 3, 3) $v_3$ |
| (1, 5, 5, 4) $v_5$ |
| (1, 6, 6, 4) $v_6$ |




| $L(\{q_2, q_5\})$ - B: |
|---|
| (1, 4, 8, 3) $v_4$ |
| (1, 2, 9, 2) $v_2$ |
| (1, 10, 10, 2) $v_8$ |

# Evaluation of Tree Pattern Queries




$A_{v_3}$ :

| | | | | |
|---|---|---|---|---|
| | $q_2$ | [4, 4] | 3 | 3 |
| | $q_2$ | [5, 5] | 4 | 4 |

$A_{v_5}$ :

| | | | | |
|---|---|---|---|---|
| | $q_2$ | [4, 4] | 3 | 3 |
| | $q_2$ | [5, 5] | 4 | 4 |

$A_{v_6}$ :

| | | | | |
|---|---|---|---|---|
| | $q_2$ | [4, 4] | 3 | 3 |
| | $q_2$ | [5, 5] | 4 | 4 |

| | | | | |
|---|---|---|---|---|
| | $q_2$ | [4, 5] | 3 | 3 |
| | $q_2$ | [5, 5] | 4 | 4 |

$A_{v_4}$ :

| | | | | |
|---|---|---|---|---|
| | $q_1$ | [2, 2] | 2 | 5 |
| | $q_1$ | [3, 3] | 6 | 6 |

| | | | | |
|---|---|---|---|---|
| | $q_2$ | [4, 4] | 3 | 3 |
| | $q_2$ | [5, 5] | 4 | 4 |
| | $q_1$ | [2, 2] | 2 | 5 |
| | $q_1$ | [3, 3] | 6 | 6 |

**Proposition** Algorithm *tree-embedding*( ) computes the entries in $A_v$'s correctly.

*Proof*. We prove the proposition by induction on the heights of nodes in $T'$. We use $h(v)$ to represent the height of node $v$.

*Basic step*. It is clear that any node $v$ with $h(v) = 0$ is a leaf node. Then, each entry in $A$v corresponds to a leaf node $q$ in $Q$ with *label*($v$) = *label*($q$). Since all those leaf nodes in $Q$ are checked in the order of increasing RightPos values, the entries in $A_v$ must be sorted.

*Induction step*. Assume that for any node $v$ with $h(v) \leq l$, the proposition holds. We will check any node $v$ with $h(v) = l + 1$. Let $v_1$, ..., $v_k$ be the children of $v$. Then, for each $v_i$ ($i$ = 1, ..., $k$), we have $h(v_i) \leq l$. In terms of the induction hypothesis, each is correctly constructed and sorted. Then, the output of *merge*($A_{v_1}$, ..., $A_{v_k}$) is sorted.

If there exists an $e$ such that $e.interval = interval(q)$ for some $q$ with $label(v) = label(q)$, an entry for $q$ will be constructed and appended to the end of $A_v$. Again, since the nodes in $Q$ are checked in the order of increasing RightPos values, $A_v$ must be sorted. So $merge(A_v, merge(A_{v_1}, ..., A_{v_k}))$ is correctly constructed and sorted.

**Time Complexity**

Now we analyze the time complexity of the algorithm. First, we see that for each node $v$ in $T'$, $d_v$ merging operations will be conducted, where $d_v$ is the outdegree of $v$. The cost of a merging operation is bounded by $O(leaf_Q)$ since the length of each linked list $A_v$ associated with a node $v$ in $T'$ is bounded by $O(leaf_Q)$ according to the following analysis. Consider two nodes $q_1$ and $q_2$ on a path in $Q$, if both $Q[q_1]$ and $Q[q_2]$ can be embedded in $T'[v]$, $A_v$ keeps only one entry for them.

If $q_1$ is an ancestor of $q_2$, then $A_v$ contains only the entry for $q_1$ since embedding of $Q[q_1]$ in $T'[v]$ implies the embedding of $Q[q_2]$ in $T'[v]$. Otherwise, $A_v$ keeps only the entry for $q_2$. Obviously, $Q$ can be divided into exactly $leaf_Q$ root-to-leaf paths. Furthermore, the merge of two linked lists $A_1$ and $A_2$ takes only O($max\{|A_1|, |A_2|\}$) time since both $A_1$ and $A_2$ are sorted lists according to the proof of above Proposition. (It works in a way similar to the *sort merge join*.) Therefore, the cost for generating all the linked lists is bounded by

$$\sum_{v \in T} d_v \cdot leaf_Q = \mathrm{O}(|T'| leaf_Q)$$

In addition, for each node $v$ taken from a $L(\boldsymbol{q})$, each $q$ in $\boldsymbol{q}$ will be checked (see line 6 in Algorithm *tree-embedding*( ).) This part of checking can be slightly improved as follows. Let $L(\boldsymbol{q}) = \{q_1, ..., q_k\}$. Each $q_j$ ($j = 1, ..., k$) is associated with an interval $[a_j, b_j]$. Since $q_j$'s are sorted by RightPos values, we can check $B$ (= $merge(A_{v_1}, ..., A_{v_k})$) against $\boldsymbol{q}$ in one scanning to find, for each $q_j$, whether there is an interval in $B$, which is equal to $[a_j, b_j]$. This process needs only $O(|B| + |\boldsymbol{q}|)$ time. So the total cost of this task is bounded by $O(|T'|\cdot leaf_Q) + O(|D|\cdot|Q|)$.

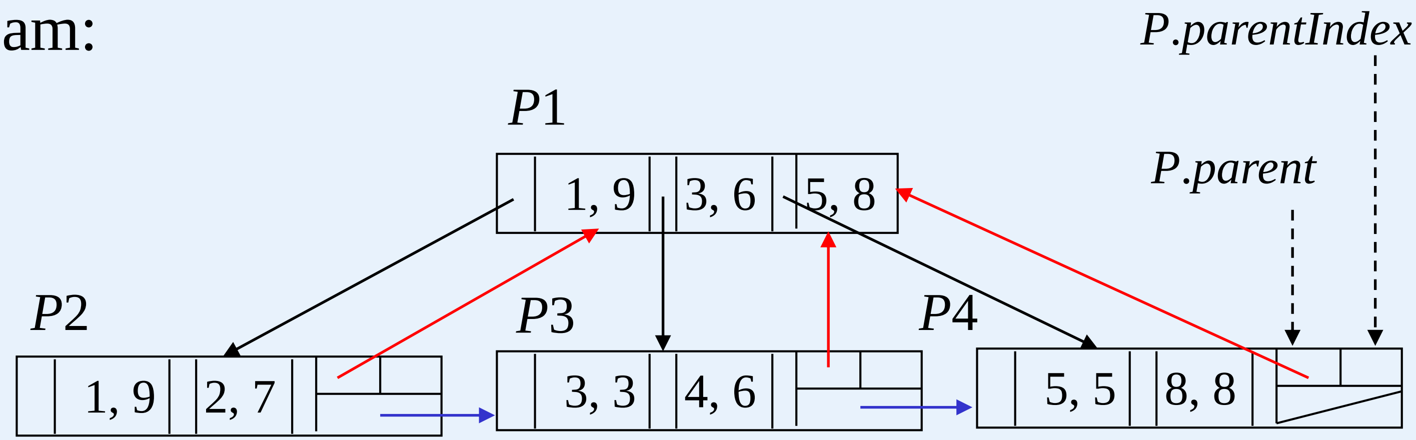
**Proposition** The time complexity of Algorithm *tree-embedding*( ) is bounded by $O(|T'|\cdot leaf_Q) + O(|D|\cdot|Q|)$.

# XB-Trees

- An XB-tree is a variant of $B^+$-tree over a quadruple sequences. In such an index structure, each entry in a page is a pair *e* = (LeftPos, RightPos) (referred to as a bounding segment) such that any entry appearing in the subtree pointed to by the pointer associated with *e* is subsumed by *e*.
- All the entries in a page are sorted by their LeftPos value.
- In each page *P* of an XB-tree, the bounding segments may partially overlap
- Each page has two extra data fields: *P.parent* and *P.parentIndex*. *P.parent* is a pointer to the parent of *P*, and *P.parentIndex* is a number *i* to indicate that the *i*th pointer in *P.parent* points to *P*.

a data stream:

(1, 1, 9, 1)
(1, 2, 7, 2)
(1, 3, 3, 3)
(1, 4, 6, 3)
(1, 5, 5, 4)
(1, 8, 8, 2)




*P*3.*parentIndex* = 2 since the second pointer in *P*1 (the parent of *P*3) points to *P*3.

- In a $Q$ we may have more than one query nodes $q_1, ..., q_k$ with the same label.
- So they will share the same data stream and the same XB-tree. For each $q_j$ ($j = 1, ..., k$), we maintain a pair ($P$, $i$), denoted $\beta_{q_j}$, to indicate that the $i$th entry in the page $P$ is currently accessed for $q_j$. Thus, each $\beta_{q_j}$ ($j = 1, ..., k$) corresponds to a different searching of the same XB-tree as if we have a separate copy of that XB-tree over $B(q_j)$.

**Two operations for navigating XB-trees:**

1. *advance*($\beta_q$) (going up from a page to its parent): If $\beta_q = (P, i)$ does not point to the last entry of $P$, $i \leftarrow i + 1$. Otherwise,

   $\beta_q \leftarrow (P.parent, P.parentIndex + 1)$.
2. *drilldown*($\beta_q$) (going down from a page to one of its children): If $\beta_q = (P, i)$ and $P$ is not a leaf page, $\beta_q \leftarrow (P', 1)$, where $P'$ is the $i$th child page of $P$.

# Evaluation of Tree Pattern Queries

- Initially, for each $q$, $\beta_q$ points to (*rootPage*, 0), the first entry in the root page.
- We finish a traversal of the XB-tree for $q$ when $\beta_q$ = (*rootPage*, *last*), where *last* points to the last entry in the root page, and we advance it (in this case, we set $\beta_q$ to $\phi$, showing that the XB-tree over $B(q)$ is exhausted.)
- The entries in $B(q)$'s will be taken from the corresponding XB-tree; and many entries can be possibly skipped. Again, the entries taken from XB-trees will be reordered as shown in Algorithm *stream-transformation*( ).

Remember that in the previously discussed algorithms, the document tree nodes are taken from $B(q)$'s one by one. Now we will take the tree nodes from the corresponding XB-trees. To do this, we will search $Q$ top-down. Each time we determine a $q$ ($\in Q$), for which an entry from $B(q)$ (i.e., the corresponding XB-tree) is taken, the following three conditions are satisfied:

i) For $q$, there exists an entry $v_q$ in $B(q)$ such that it has a descendant $v_{q_i}$ in each of the streams $B(q_i)$ (where $q_i$ is a child of $q$.)

ii) Each $v_{q_i}$ recursively satisfies (i).

iii) LeftPos($v_q$) is minimum.

**In function *getNext*(*q*), the following operations are used:**

*isLeaf*($q$) - returns *true* if $q$ is a leaf node of $Q$; otherwise, *false*.

*currL*($q$) - returns the leftPos of the entry pointed to by $\beta_q$.

*currR*($\beta_q$) - returns the rightPos of the entry pointed to by $\beta_q$.

*isPlainValue*($\beta_q$) - returns *true* if $\beta_q$ is pointing to a leaf node in the corresponding XB-tree.

*end*($Q$) - if for each leaf node $q$ of $Q$ $\beta_q = \phi$ (*i.e.*, $B(q)$ is exhausted), then returns *true*; otherwise, *false*.

***getNext(q)*** returns *q'*, *but its goal is to figure out* $\beta_{q'}$ *by using the XB-tree.*

**Function** *getNext*($q$) (*Initially, $q$ is the root of $Q$.*)
**begin**
1. **if** (*isLeaf*($q$)) **then** return $q$;
2. **for** each child $q_i$ of $q$ **do**
3. $\{r_i \leftarrow getNext(q_i)$;
4. **if** ($r_i \neq q_i \vee \neg isPlainValue(\beta r_i)$) **then** return $r_i$; }
5. $q_{min} \leftarrow q''$ such that $currL(\beta_{q''}) = min_i\{currL(\beta_{r_j})\}$;
6. $q_{max} \leftarrow q'''$ such that $currL(\beta_{q'''}) = max_i\{currL(\beta_{r_i})\}$;
7. **while** ($currR(\beta_q) < currL(\beta_{q_{max}})$) **do** *advance*($\beta_q$);
8. **if** ($currL(\beta_q) < currL(\beta_{q_{min}})$) **then** return $q$;
9. **else** return $q_{min}$; }

**end**

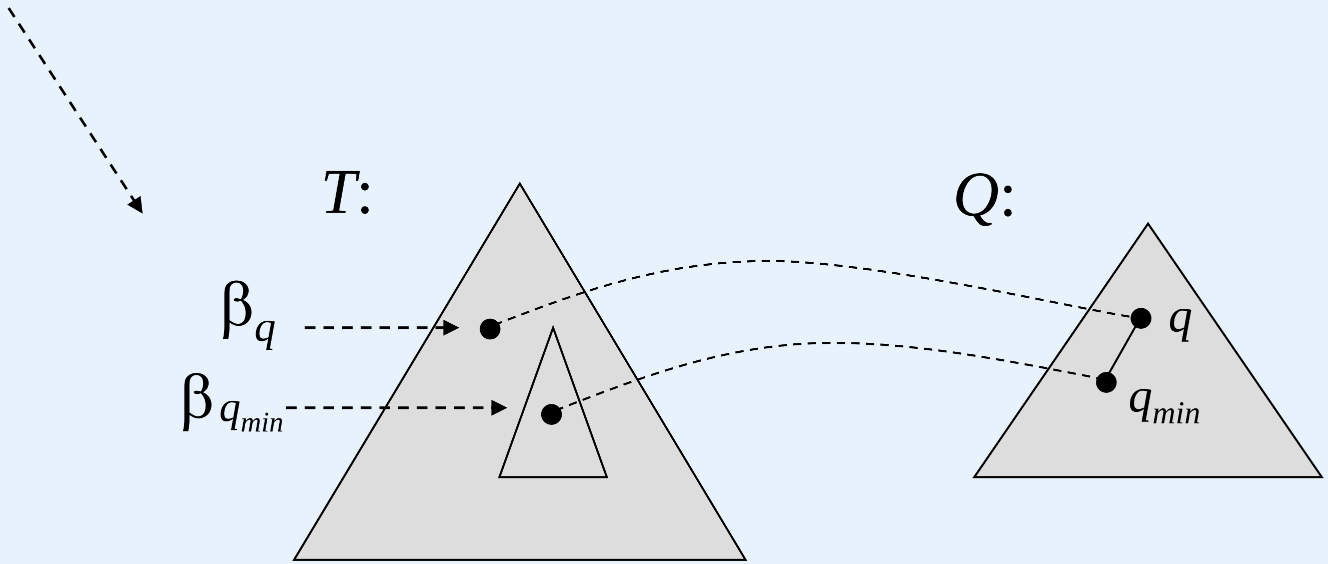
When $r_i \neq q_i$ , we will return $r_i$ since $q$ cannot satisfy condition (i) (see line 9).

The goal of the above function is to figure out a query node to determine what entry from data streams will be checked in a next step, which has to satisfy the above conditions (i) - (iii).

- Lines 7 – 9 are used to find a query node satisfying condition (i) (see the figure for illustration of line 7.)
- The recursive call performed in line 3 shows that condition (ii) is met.
- Since each XB-tree is navigated top-down and the entries in each node is scanned from left to right, condition (iii) must be always satisfied.

# Evaluation of Tree Pattern Queries

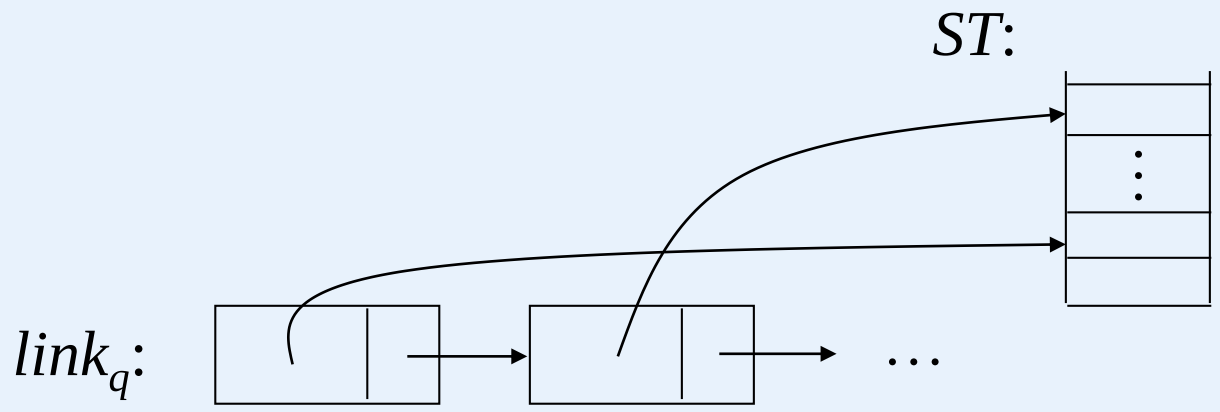
If $currR(\beta_q) < currL(\beta_{q_{min}})$, we have to *advance* $\beta_q$.



**Algorithm *tree-embeddingXB*(*Q*)**

- Once a $q \in Q$ is returned, we will further check $\beta_q$. If it is an entry in a leaf node in the corresponding XB-tree, insert it into stack *ST* (see Algorithm *stream-transformation*( ).) Otherwise, we will do *advance*($\beta_q$) or *drilldown*($\beta_q$), according to the relationship between $\beta_q$ and the nodes stored in *ST*.
- We associate each $q \in Q$ with an extra linked list, denoted $link_q$, such that each entry in it contains a pointer to a node *v* stored in *ST* with *label*(*v*) = *label*(*q*). We append entries to the end of a $link_q$ one by one as the document nodes are inserted into *ST*, as illustrated in the following figure. The last entry in $link_q$ is denoted a $link_{q,last}$

# Evaluation of Tree Pattern Queries



**Algorithm** *tree-embeddingXB*($Q$)

**begin**

**while** ($\neg$*end*($Q$)) **do**

{$q \leftarrow$ *getNext*(*root-of-Q*);

**if** (*isPlainValue*($\beta_q$) **then**

{let $v$ be the node pointed to by $\beta_q$;

**while** *ST* is not empty and *ST.top* is not $v$'s ancestor **do**

{$x \leftarrow$ *ST.pop*(); Let $x = (q', u)$; (*a node for $u$ will be created.*)

call *embeddingCheck*($q'$, $u$); }

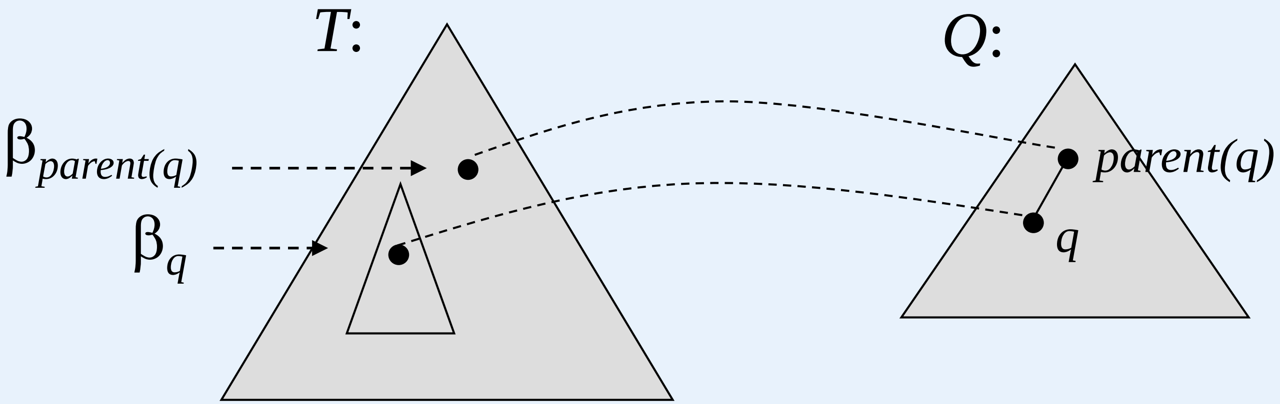
*ST.push*($q$, $v$); *advance*($\beta_q$);

}

**10. else if** $(\neg isRoot(q) \wedge link_q \neq \phi$
$\wedge\ currR(\beta_q) < \mathrm{LeftPos}(link_{q,last})$
11. **then** $advance(\beta_q)$ (*not part of a solution*)
12. **else** $drilldown(\beta_q)$; (*may find a solution.*)
}
**end**

# Evaluation of Tree Pattern Queries

In the above algorithm, we distinguish between two cases. If $\beta_q$ is an entry in a leaf node in the corresponding XB-tree, we will insert it into *ST*. Otherwise, lines 10 - 12 will be carried out. If $currR(\beta_q) <$ LeftPos($link_{parent(q),last}$), we have a situation as illustrated in the following figure. In this case, we will advance $\beta_q$ (see line 11.) If it is not the case, we will drill down the corresponding XB-tree (see line 12) since a solution may be found.



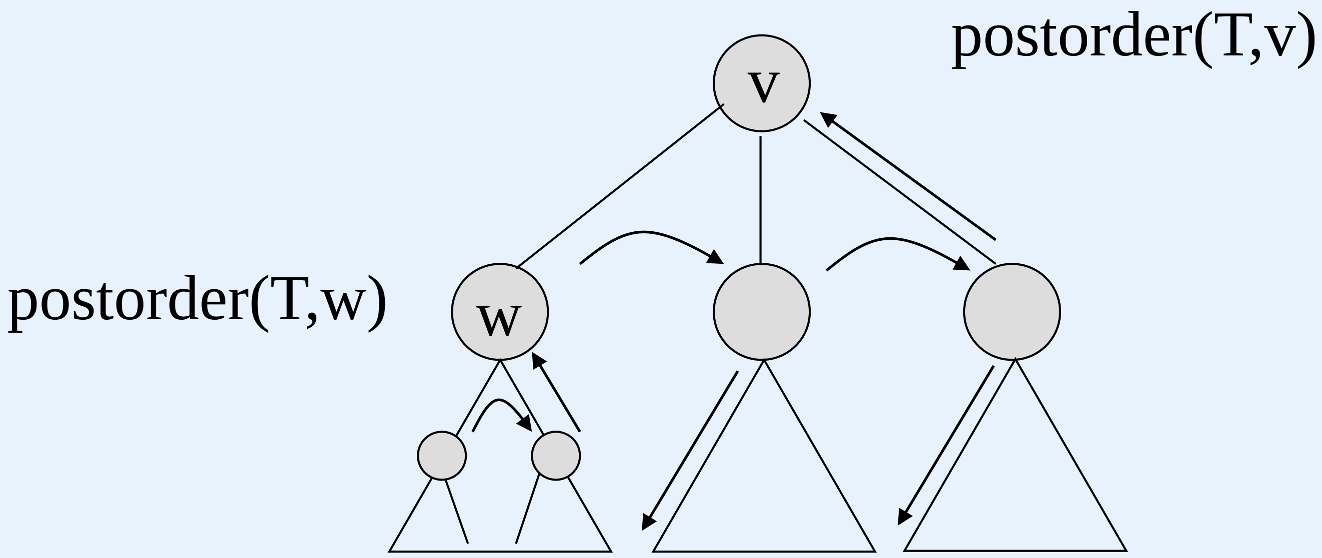
## Appendix – bottom-up tree searching

Algorithm postorder(T,v):

for each child w of v

  call postorder(T,w)

perform the “visit” action for node v



# Postorder traversal using Stack

```
Algorithm stack-postorder(T, v)
    establish stack S;
    S.push(v)
    while (S in not empty) do {
        u := S.top();
        if (u is leaf or marked) then {visit u; S.pop();}
        else    mark the top element of S;
                let u1, u2, …, un be the children of u;
                for (j = n; j >= 1; j--) S.push(uj);
                }
    }
```