Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . . ).
Datastructure?

Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . . ).

⇒ Using the proper data-structure for a task matters!
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, ...).
⇒ Using the proper data-structure for a task matters!

Abstract specification (Interfaces)

- Collection
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . . ). ⇒ Using the proper data-structure for a task matters!

Abstract specification (Interfaces)

- Collection
- Stack
Datastructure?

Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . . ). ⇒ Using the proper data-structure for a task matters!

Abstract specification
(Interfaces)
   ▶ Collection
   ▶ Stack
   ▶ Set
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . . ).

⇒ Using the proper data-structure for a task matters!

Abstract specification
(Interfaces)

- Collection
- Stack
- Set
- Map
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, ...). ⇒ Using the proper data-structure for a task matters!

Abstract specification (Interfaces)
- Collection
- Stack
- Set
- Map

Concrete types (Classes)
- (Pair)
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . .).

⇒ Using the proper data-structure for a task matters!

Abstract specification (Interfaces)

- Collection
- Stack
- Set
- Map

Concrete types (Classes)

- (Pair)
- List
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . . ). ⇒ Using the proper data-structure for a task matters!

Abstract specification (Interfaces)

- Collection
- Stack
- Set
- Map

Concrete types (Classes)

- (Pair)
- List
- Ordered Tree
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, ...). ⇒ Using the proper data-structure for a task matters!

Abstract specification (Interfaces)
- Collection
- Stack
- Set
- Map

Concrete types (Classes)
- (Pair)
- List
- Ordered Tree
- Hashtable
Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags, . . .).

⇒ Using the proper data-structure for a task matters!

**Abstract specification (Interfaces)**
- Collection
- Stack
- Set
- Map

**Concrete types (Classes)**
- (Pair)
- List
- Ordered Tree
- Hashtable
Collection

Simplest abstract data-structure, allows to group several objects (called *elements*) in the same structure. Operations:

- `isEmpty()`, test for emptyness
Simplest abstract data-structure, allows to group several objects (called *elements*) in the same structure. Operations:

- `isEmpty()`, test for emptiness
- `add(E)`, adds an object to the collection
Collection

Simplest abstract data-structure, allows to group several objects (called *elements*) in the same structure. Operations:

- `isEmpty()`, test for emptiness
- `add(E)`, adds an object to the collection
- `remove(E)`, removes an object from the collection

All the datastructures presented here support this!
Collection

Simplest abstract data-structure, allows to group several objects (called \textit{elements}) in the same structure. Operations:

- \texttt{isEmpty()}, test for emptyness
- \texttt{add(E)}, adds an object to the collection
- \texttt{remove(E)}, removes an object from the collection
- \texttt{contains(E)}, tests if an objects is in the collection
Collection

Simplest abstract data-structure, allows to group several objects (called *elements*) in the same structure. Operations:

- `isEmpty()`, test for emptyness
- `add(E)`, adds an object to the collection
- `remove(E)`, removes an object from the collection
- `contains(E)`, tests if an objects is in the collection
- `iterator()`, returns an iterator over the elements
Collection

Simplest abstract data-structure, allows to group several objects (called *elements*) in the same structure. Operations:

▶ *isEmpty()*, test for emptiness
▶ *add(E)*, adds an object to the collection
▶ *remove(E)*, removes an object from the collection
▶ *contains(E)*, tests if an objects is in the collection
▶ *iterator()*, returns an iterator over the elements

All the datastructures presented here support this!
Stack

LIFO data-structure, elements are ordered in reverse order of insertion. Operations:

- `push(E)`, puts an element on the top of the stack
Stack

LIFO data-structure, elements are ordered in reverse order of insertion.
Operations:

- **push(E)**, puts an element on the top of the stack
- **pop()**, removes the topmost element and returns it
Stack

LIFO data-structure, elements are ordered in reverse order of insertion. Operations:

- **push(E)**, puts an element on the top of the stack
- **pop()**, removes the topmost element and returns it
- **peek()**, returns the topmost element without returning it
Stack

LIFO data-structure, elements are ordered in reverse order of insertion. Operations:

- `push(E)`, puts an element on the top of the stack
- `pop()`, removes the topmost element and returns it
- `peek()`, returns the topmost element without returning it
Collection of *unique* elements.
Operations are the same as for the collection:
Set

Collection of *unique* elements.
Operations are the same as for the collection:

- `add(E)` adds an element to the set, returns *true* if the set was modified, else *false*
Set

Collection of *unique* elements.

Operations are the same as for the collection:

- `add(E)` adds an element to the set, returns `true` if the set was modified, else `false`
- `remove(E)`, removes an element from the set, returns `true` if the set was modified, else `false`
Collection of *unique* elements.

Operations are the same as for the collection:

- `add(E)` adds an element to the set, returns `true` if the set was modified, else `false`
- `remove(E)`, removes an element from the set, returns `true` if the set was modified, else `false`
Map (or Dictionary)

Collection of pairs of elements (*key*, *data*). Associates any data with a key, e.g.:

```{  
  "www.google.com" → [209 85 171 100]  
  "www.unsw.edu.au" → [149 171 96 58]  
  ...  
}  ```
Map (or Dictionary)

Collection of pairs of elements \((key, data)\).
Associates any data with a key, e.g.:

\[
\begin{aligned}
\{ \\
"www.google.com" & \rightarrow \begin{pmatrix} 209 & 85 & 171 & 100 \end{pmatrix} \\
"www.unsw.edu.au" & \rightarrow \begin{pmatrix} 149 & 171 & 96 & 58 \end{pmatrix} \\
\ldots \\
\} \\
\end{aligned}
\]

- \(\text{put}(K, E)\) adds an element to the map with the specified key, returns the previous mapping for \(K\) or null
Map (or Dictionary)

Collection of pairs of elements \((key, data)\). Associates any data with a key, e.g.:

\[
\begin{align*}
\{ & \\
\quad "www.google.com" \rightarrow & \begin{array}{c}
209 \\
85 \\
171 \\
100 \\
\end{array} \\
\quad "www.unsw.edu.au" \rightarrow & \begin{array}{c}
149 \\
171 \\
96 \\
58 \\
\end{array} \\
\ldots & \\
\}\n\end{align*}
\]

- put\((K, E)\) adds an element to the map with the specified key, returns the previous mapping for \(K\) or null
- get\((K)\), returns the element associated with \(K\) or null
Map (or Dictionary)

Collection of pairs of elements \((key, data)\). Associates any data with a key, e.g.:

\[
\{ \\
    "www.google.com" \rightarrow \begin{array}{c}
        209 \\
        85 \\
        171 \\
        100 \\
    \end{array} \\
    "www.unsw.edu.au" \rightarrow \begin{array}{c}
        149 \\
        171 \\
        96 \\
        58 \\
    \end{array} \\
    \ldots \\
\}
\]

- put\((K, E)\) adds an element to the map with the specified key, returns the previous mapping for \(K\) or null
- get\((K)\), returns the element associated with \(K\) or null

All the keys form a Set (keys are unique)
The add and remove methods take the key as argument
Pair (1/2)

Not provided in Java but extremely useful (exists in the C++ STL)
Encapsulates exactly 2 objects.
  - get/setFirst() returns/sets the first component
  - get/setSecond() returns/setsthe second component

Implement it in java:

class Pair {
    private Object first;
    private Object second;

    Pair (Object x, Object y) {
        first = x;
        second = y;
    }

    public Object getFirst () {
        return first;
    }

    public Object getSecond () {
        return second;
    }

    public void setFirst (Object e) {
        first = e;
    }

    public void setSecond (Object e) {
        second = e;
    }
}
Pair (1/2)

Not provided in Java but extremely useful (exists in the C++ STL)
Encapsulates exactly 2 objects.
▶ get/setFirst() returns/sets the first component
▶ get/setSecond() returns/sets the second component

Implement it in java:

class Pair {
    private Object first;
    private Object second;
    Pair (Object x, Object y){
        first = x;
        second = y;
    }
    public Object getFirst(){ return first; }
    public Object getSecond(){ return second; }
    public void setFirst(Object e){ first=e; }
    public void setSecond(Object e){ second=e; }
}
Implement it in java with *generics*:

```java
class Pair<X, Y> {
    private X first;
    private Y second;

    Pair(X x, Y y) {
        first = x;
        second = y;
    }

    public X getFirst() {
        return first;
    }

    public Y getSecond() {
        return second;
    }

    public void setFirst(X x) {
        first = x;
    }

    public void setSecond(Y y) {
        second = y;
    }
}
```
Implement it in java with *generics*:

```java
class Pair<X,Y> {
    private X first;
    private Y second;
    Pair (X x, Y y){
        first = x;
        second = y;
    }
    public X getFirst() { return first; }
    public Y getSecond() { return second; }
    public void setFirst(X x){ first=x; }
    public void setSecond(Y y){ second=y; }
}
```
Pair (2/2)

Implement it in java with generics:

class Pair<X,Y> {
    private X first;
    private Y second;
    Pair (X x, Y y){
        first = x;
        second = y;
    }
    public X getFirst(){ return first; }
    public Y getSecond(){ return second; }
    public void setFirst(X x){ first=x; }
    public void setSecond(Y y){ second=y; }
}

- less error-prone
- more efficient
Implement sequences of elements, as a chain of cells. The following operations can be done in constant time ("superhypermegafastlolroflmaoomgwtfbbq"): 
- addFirst(E), adds an element at the beginning of the list 
- getFirst(), returns the first element of the list 
- removeFirst(), removes the first element of the list 

Does it look like something you know?
Implement sequences of elements, as a chain of cells. The following operations can be done in constant time ("superhypermegafastlolrolmaoomgwtfbbq"):

- `addFirst(E)`, adds an element at the beginning of the list
- `getFirst()`, returns the first element of the list
- `removeFirst()`, removes the first element of the list

Does it look like something you know?
Implement it in Java (with generics)

```java
class LinkedList<E> {
    private E content;
    private LinkedList<E> next;
    LinkedList (E e, LinkedList<E> l) {
        content = e;
        next = l;
    }
}
```
`LinkedList (E e) { LinkedList(e, null) };
LinkedList () { LinkedList(null, null) };

public E getFirst() { returns content; }`
```java
public E getFirst() { returns content; }
public void addFirst(E e) {
    if (content == null) {
        content = e;
    } else {
        LinkedList<E> tail =
            new LinkedList<E>(content, next);
        next = tail;
        content = e;
    }
}
...
public void removeFirst() {
    if (content == null)
        return;
    else {
        content = next.content;
        next = next.next;
    }
}
public void removeFirst() {
    if (content == null)
        return;
    else {
        content = next.content;
        next = next.next;
    }
}

Can you implement this using only pairs?
public void removeFirst() {
    if (content == null)
        return;
    else {
        content = next.content;
        next = next.next;
    }
}
Can you implement this using only pairs?
Write a List class which allows null elements
One fits all data-structure?

- Can you implement a Stack with a List? Is it efficient?
One fits all data-structure?

- Can you implement a Stack with a List? Is it efficient?
- Can you implement a Set with a List? Is it efficient?
One fits all data-structure?

- Can you implement a Stack with a List? Is it efficient?
- Can you implement a Set with a List? Is it efficient?
- Can you implement a Map with a List? Is it efficient?
One fits all data-structure?

- Can you implement a Stack with a List? Is it efficient?
- Can you implement a Set with a List? Is it efficient?
- Can you implement a Map with a List? Is it efficient?
Balanced Binary Tree (1/3)

Relies on a total ordering of elements (must implement the Comparable interface):

int compareTo(0): o1.compareTo(o2) returns an integer i:

- $i < 0$ if $o1 < o2$
- $i = 0$ if $o1 = o2$
- $i > 0$ if $o1 > o2$

BFW: (Big Fat Warning) $o1 == o2 \Rightarrow$ compareTo(o2) but the converse IS NOT TRUE! in general. It is only true for immediate values int, char, bool, null, not for pointers (i.e. Integers, Chars, ...).
Two types of Tree object:

- EmptyTree
- Node(E elem, Tree left, Tree right)
Balanced Binary Tree (2/3)

Two types of Tree object:
- EmptyTree
- Node(E elem, Tree left, Tree right)

Properties:
- $\forall x \in left, x.compareTo(elem) < 0$
- $\forall x \in right, x.compareTo(elem) > 0$
- $|height(left) - height(right)| = 1$
Two types of Tree object:
- EmptyTree
- Node(E elem, Tree left, Tree right)

Properties:
- \( \forall x \in left, x\text{.compareTo}(elem) < 0 \)
- \( \forall x \in right, x\text{.compareTo}(elem) > 0 \)
- \( |height(left) - height(right)| = 1 \)

Complexity:
- add, remove, contains: \( \log_2(n) \) (aka “fast enough”).

Other nice property:
- Iterating in increasing order is a left right depth first traversal
- Iterating in decreasing order is a right left depth first traversal
Can you implement a Set using a BBT? Is it efficient?

TreeSet in Java.

Draw the tree created after inserting 5, 3, 6, 7, 8, 2, 4 in the empty tree.

Do the same after inserting 1, 2, 3, 4, 5, 6, 7, 8. What's the problem?

⇒ rebalancing is important!

Can you implement a Map using a BBT? Is it efficient?

⇒ TreeMap in Java.

Cons:

⇒ \( \log_2(n) \) is acceptable in many cases but still not super mega fast

⇒ need a total ordering over objects
Can you implement a Set using a BBT? Is it efficient?

⇒ TreeSet in Java.

Draw the tree created after inserting 5,3,6,7,8,2,4 in the empty tree. Do the same after inserting 1,2,3,4,5,6,7,8. What’s the problem?
Can you implement a Set using a BBT? Is it efficient?
⇒ TreeSet in Java.

Draw the tree created after inserting 5, 3, 6, 7, 8, 2, 4 in the empty tree. Do the same after inserting 1, 2, 3, 4, 5, 6, 7, 8. What’s the problem?
⇒ rebalancing is important!
Can you implement a Set using a BBT? Is it efficient?
⇒ TreeSet in Java.

Draw the tree created after inserting 5,3,6,7,8,2,4 in the empty tree.
Do the same after inserting 1,2,3,4,5,6,7,8. What’s the problem?
⇒ rebalancing is important!

Can you implement a Map using a BBT? Is it efficient?
Can you implement a Set using a BBT? Is it efficient?
⇒ TreeSet in Java.

Draw the tree created after inserting 5,3,6,7,8,2,4 in the empty tree.
Do the same after inserting 1,2,3,4,5,6,7,8. What’s the problem?
⇒ rebalancing is important!

Can you implement a Map using a BBT? Is it efficient?
⇒ TreeMap in Java.

Cons:

- \( \log_2(n) \) is acceptable in many cases but still not “super mega fast”
- need a total ordering over objects
Hashtable (1/3)

Implements Map, i.e. stores associations of keys and values. Needs:

- a hash function for keys
Hashtable (1/3)

Implements Map, i.e. stores associations of keys and values. Needs:

- a hash function for keys
- an equality function between keys
Hashtable (1/3)

Implements Map, i.e. stores associations of keys and values. Needs:

- a hash function for keys
- an equality function between keys

BFW: The whole behaviour depends on the hash function, its VERY tricky to get a correct hash function!

Basic data structure: Array of LinkedList (the cells of the array are often called slots and the lists bucket).
Hashtable (1/3)

Implements Map, i.e. stores associations of keys and values. Needs:
- a hash function for keys
- an equality function between keys

BFW: The whole behaviour depends on the hash function, its VERY tricky to get a correct hash function!

Basic data structure: Array of LinkedList (the cells of the array are often called slots and the lists bucket).

How does it work?
Suppose we want to associate strings with IP addresses (stored as arrays of integers).

Suppose 10 slots, initially filled with empty buckets.

We want to insert ("www.google.com", [209, 85, 171, 100]):

1. compute the hash of the key, $hash("www.google.com") = 2810$
2. maps the hash (2810) to a value between 0 and 10: $2810 \mod 10 = 0$
3. get the linked list at position 0 in the Hashtable
4. insert the pair (key, data) at the beginning of the list
 Hashtable (3/3)

What happens if two keys go into the same slot?
What happens if two keys go into the same slot?
What happens if a lot of keys go into the same slot?

Good properties of a hash function:
▶ Good distribution: all keys are hashed to different integers
▶ Fast

As for the BBT, we need to resize (rebalance) the Hashtable if the buckets are too large. Rebalancing needs to be fast/not too often.

Only if we have these properties, we get constant time for delete, add, exists

Can you implement Map using Hashtable? Is it efficient?
Can you implement Set using Hashtable? Is it efficient?
⇒ See HashSet and HashMap in Java. Cons:
The iterators are in unsorted order
What happens if two keys go into the same slot?
What happens if a lot of keys go into the same slot?
What happens if two keys have the same hash?
What happens if two keys go into the same slot?
What happens if a lot of keys go into the same slot?
What happens if two keys have the same hash?

Good properties of a hash function:
- Good distribution: all keys are hashed to different integers
- Fast
Hashtable (3/3)

What happens if two keys go into the same slot?
What happens if a lot of keys go into the same slot?
What happens if two keys have the same hash?

Good properties of a hash function:
  - Good distribution: all keys are hashed to different integers
  - Fast

As for the BBT, we need to resize (rebalance) the Hashtable if the buckets are too large. Rebalancing needs to be fast/not too often. Only if we have these properties, we get constant time for delete, add, exists

Can you implement Map using Hashtable? Is it efficient?
What happens if two keys go into the same slot?
What happens if a lot of keys go into the same slot?
What happens if two keys have the same hash?

Good properties of a hash function:

- Good distribution: all keys are hashed to different integers
- Fast

As for the BBT, we need to resize (rebalance) the Hashtable if the buckets are too large. Rebalancing needs to be fast/not too often. Only if we have these properties, we get constant time for `delete`, `add`, `exists`.

Can you implement `Map` using `Hashtable`? Is it efficient?
Can you implement `Set` using `Hashtable`? Is it efficient?
What happens if two keys go into the same slot?
What happens if a lot of keys go into the same slot?
What happens if two keys have the same hash?

Good properties of a hash function:
  ▶ Good distribution: all keys are hashed to different integers
  ▶ Fast

As for the BBT, we need to resize (rebalance) the Hashtable if the buckets are too large. Rebalancing needs to be fast/not too often. Only if we have these properties, we get constant time for delete, add, exists.

Can you implement Map using Hashtable? Is it efficient?
Can you implement Set using Hashtable? Is it efficient?
⇒ See HashSet and HashMap in Java. Cons: The iterators are in unspecified order
Recursive tree traversal

Remember tutorial 01, you have a class Node:

- `getFirstChild()` returns the first child of the node
- `getNextSibling()` returns the next sibling of a node
- an empty Node or NodeList is null

```java
void traverse(Node n) {
    if (n == null) return;
    else {
        Node n_c = n.getFirstChild();
        while (n_c != null) {
            traverse(n_c);
            n_c = n.getNextSibling();
        }
        return;
    }
}
```
Recursive tree traversal

Remember tutorial 01, you have a class Node:

- `getFirstChild()` returns the first child of the node
- `getNextSibling()` returns the next sibling of a node
- an empty Node or NodeList is `null`

A recursive tree traversal goes like this:

```java
void traverse(Node n) {
    if (n == null)
        return;
    else {
        Node nc = n.getFirstChild();
        while (nc != null) {
            traverse(nc);
            nc = n.getNextSibling();
        }
        return;
    }
}
```
Recursive tree traversal

Remember tutorial 01, you have a class Node:

- `getFirstChild()` returns the first child of the node
- `getNextSibling()` returns the next sibling of a node
- an empty Node or NodeList is null

A recursive tree traversal goes like this:

```java
void traverse(Node n) {
    if (n == null)
        return;
    else {
        Node nc = n.getFirstChild();
        while (nc != null) {
            traverse(nc);
            nc = n.getNextSibling();
        }
        return;
    }
}
```
Recursive tree traversal

Call stack:

Code:
traverse(n)
Recursive tree traversal

Code:
if (n == null) // false
Recursive tree traversal

Call stack:
- `n:0x00abcbd1` ➔ `{tag:'a', fs:0xaabc231, ns:null, ...}`
- `nc:0xaabc231` ➔ `{tag:'b', fs:null, ns:0x3942a2, ...}`

Code:
```
nc = n.getFirstChild();
```
Recursive tree traversal

Code:
while (nc != null) {
  //true
  traverse(nc)
Recursive tree traversal

Code:
```java
if (n == null)  // false
```
Recursive tree traversal

Call stack:
\[\begin{align*}
n &: 0x00abcd1 \quad \mapsto \quad \{ \text{tag: 'a', fs: 0xaabc231,} \,
\text{ns: null, ...} \} \\
c &: 0xaabc231 \quad \mapsto \quad \{ \text{tag: 'b', fs: null,} \,
\text{ns: 0x3942a2, ...} \} \\
n &: 0xaabc231 \quad \mapsto \quad \{ \text{tag: 'b', fs: null,} \,
\text{ns: 0x3942a2, ...} \} \\
nc &: \text{null} \\
\end{align*}\]

Code:
\[
nc = n.getFirstChild();
\]
Recursive tree traversal

```
// Call stack:
n:0x00abcbd1 ← {tag:'a', fs:0xaabc231, ns:null, ...}
nc:0xaabc231 ← {tag:'b', fs:null, ns:0x3942a2, ...}
n:0xaabc231 ← {tag:'b', fs:null, ns:0x3942a2, ...}
nc:0xaabc231 ← {tag:'b', fs:null, ns:0x3942a2, ...}
nc:null

Code:
while (nc != null) { ... } //false
return
```
Recursive tree traversal

```
Code:
nc = nc.getNextSibling();
```

Call stack:
```
n:0x00abcd1 ↦→ {tag:'a', fs:0xaabc231,
   ns:null, ...}
nc:0x3942a2 ↦→ {tag:'c', fs:0x12af89,
   ns:0xbbf638, ...}
```
Recursive tree traversal

Code:
while (nc != null) {
  //true
  traverse(nc)

Call stack:
n:0x00abcd1 ↦ {tag:'a', fs:0xaabc231,
          ns:null, ...}
nc:0x3942a2 ↦ {tag:'c', fs:0x12af89,
          ns:0xbbf638, ...}
# Recursive tree traversal

```plaintext
Code:
if (n == null)  // false

Call stack:
- n:0x00abcbd1 ↔ {tag:’a’, fs:0xaabc231, ns:null, ...}
- nc:0x3942a2 ↔ {tag:’c’, fs:0x12af89, ns:0xbbf638, ...}
- n:0x3942a2 ↔ {tag:’c’, fs:0x12af89, ns:0xbbf638, ...}

```

![Recursive tree traversal diagram]
Recursive tree traversal

```
Call stack:
n:0x00abcdbd1  ↦  {tag:'a', fs:0xaabc231, ns:null, ...}
nc:0x3942a2    ↦  {tag:'c', fs:0x12af89, ns:0xbbf638, ...}
n:0x3942a2    ↦  {tag:'c', fs:0x12af89, ns:0xbbf638, ...}
nc:0x12af89    ↦  {tag:'d', fs:null, ns:0x48ab2f, ...}
```

Code:
```
nc = n.getFirstChild();
```
Recursive tree traversal

Call stack:

- n:0x00abcbd1 ↦ {tag:'a', fs:0xaabc231, ns:null, ...}
- nc:0x3942a2 ↦ {tag:'c', fs:0x12af89, ns:0xbbf638, ...}
- n:0x3942a2 ↦ {tag:'c', fs:0x12af89, ns:0xbbf638, ...}
- nc:0x12af89 ↦ {tag:'d', fs:null, ns:0x48ab2f, ...}

Code:
```java
while (nc != null) { //true
    traverse(nc)
}
```
Recursive tree traversal

Code:
if (n == null) // false
Recursive tree traversal

Call stack:
- n:0x00abcbd1 ↦ {tag: ’a’, fs:0xaabc231, ns:null, ...}
- nc:0x3942a2 ↦ {tag: ’c’, fs:0x12af89, ns:0xbbf638, ...}
- n:0x3942a2 ↦ {tag: ’c’, fs:0x12af89, ns:0xbbf638, ...}
- nc:0x12af89 ↦ {tag: ’d’, fs:null, ns:0x48ab2f, ...}
- n:0x12af89 ↦ {tag: ’d’, fs:null, ns:0x48ab2f, ...}
- nc:null

Code:
nc = n.getFirstChild();
Recursive tree traversal

```
Code:
while (nc != null) { ... } //false
return
```
Recursive tree traversal

Call stack:

```
nc:0x3942a2  →  {tag:'c', fs:0x12af89, ns:0xbbf638, ...}
n:0x3942a2  →  {tag:'c', fs:0x12af89, ns:0xbbf638, ...}
n:0x00abcbd1  →  {tag:'a', fs:0xaabc231, ns:null, ...}
```

Code:
```
nc = nc.getNextSibling();
```
Recursive tree traversal

Call stack:
- n:0x00abcbd1 ↦ {tag: ‘a’, fs:0xaabc231, ns:null, ...}
- nc:0x3942a2 ↦ {tag: ‘c’, fs:0x12af89, ns:0xbbf638, ...}
- n:0x3942a2 ↦ {tag: ‘c’, fs:0x12af89, ns:0xbbf638, ...}
- nc:0x48ab2f ↦ {tag: ‘e’, fs:null, ns:null, ...}

Code:
```java
while (nc != null) {
    //true
    traverse(nc)
}
```
Recursive tree traversal

Code: