XML and Databases
Tutorial session 2: Basic datastructures

Kim.Nguyen@nicta.com.au

Week 3

Datastructure ?

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,...).
$\Rightarrow$ Using the proper data-structure for a task matters!

Datastructure ?

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

Abstract specification
(Interfaces)

- Collection

Datastructure ?

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

Abstract specification
(Interfaces)

- Collection
- Stack
- Set

Datastructure ?

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

Abstract specification
(Interfaces)

- Collection
- Stack
- Set
- Map

Datastructure ?

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

Abstract specification (Interfaces)

- Collection
- Stack
- Set
- Map

Concrete types (Classes)

- (Pair)

Datastructure ?

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

Abstract specification (Interfaces)

- Collection
- Stack
- Set
- Map

Concrete types (Classes)

- (Pair)
- List

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
- Map

Concrete types (Classes)

- (Pair)
- List
- Ordered Tree

Datastructure ?

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

Abstract specification (Interfaces)

- Collection
- Stack
- Set
- Map

Concrete types (Classes)

- (Pair)
- List
- Ordered Tree
- Hashtable

Datastructure ?

Various operations in XML are expensive (retrieve all the elements below a node, retrieve all the text nodes, retrieve all the tags,...).
$\Rightarrow$ 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-strcuture, allows to group several objects (called *elements*) in the same structure. Operations:

- `isEmpty()`, test for emptyness

Collection

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

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

Collection

Simplest abstract data-strcuture, 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

Collection

Simplest abstract data-strcuture, 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

Collection

Simplest abstract data-strcuture, 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-strcuture, 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

All the datastructures presented here support this!

Stack

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

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

Stack

LIFO data-strcuture, 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-strcuture, 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-strcuture, 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

Set

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`

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`

Map (or Dictionary)

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

$\{$

`"www.google.com"` $\rightarrow$

209	*85*	*171*	*100*

`"www.unsw.edu.au"` $\rightarrow$

149	*171*	*96*	*58*

$\ldots$

$\}$

Map (or Dictionary)

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

$$
\begin{array}{l}
\{ \\
\texttt{"www.google.com"} \rightarrow \boxed{\textit{209} \mid \textit{85} \mid \textit{171} \mid \textit{100}} \\
\texttt{"www.unsw.edu.au"} \rightarrow \boxed{\textit{149} \mid \textit{171} \mid \textit{96} \mid \textit{58}} \\
\ldots \\
\}
\end{array}
$$

- `put(K,E)` adds an element to the map with the specidied 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.:

$$
\begin{array}{l}
\{ \\
\texttt{"www.google.com"} \rightarrow \boxed{\mathit{209}}\boxed{\mathit{85}}\boxed{\mathit{171}}\boxed{\mathit{100}} \\
\texttt{"www.unsw.edu.au"} \rightarrow \boxed{\mathit{149}}\boxed{\mathit{171}}\boxed{\mathit{96}}\boxed{\mathit{58}} \\
\ldots \\
\}
\end{array}
$$

- `put(K,E)` adds an element to the map with the specidied 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"` $\rightarrow$

209	*85*	*171*	*100*

`"www.unsw.edu.au"` $\rightarrow$

149	*171*	*96*	*58*

...
}

- `put(K,E)` adds an element to the map with the specidied 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:

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 (2/2)

Implement it in java with *generics*:

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; }
}
```

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

(Linked)List (1/3)

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 begining 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?

(Linked)List (1/3)

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 begining 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)

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

(Linked)List (2/3)

```
LinkedList (E e) { LinkedList(e,null) };
LinkedList () { LinkedList(null,null) };

public E getFirst(){ returns content;}
```

(Linked)List (2/3)

```
LinkedList (E e) { LinkedList(e,null) };
LinkedList () { LinkedList(null,null) };

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;
 }
}
...
```

(Linked)List (3/3)

```
  public void removeFirst() {
    if (content == null)
       return;
   else {
     content = next.content;
     next = next.next;
  }
}
```

(Linked)List (3/3)

```
  public void removeFirst() {
    if (content == null)
        return;
   else {
      content = next.content;
      next = next.next;
  }
}
```

Can you implement this using only pairs ?

(Linked)List (3/3)

```
  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

(Linked)List (3/3)

```
  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 oredering of elements* (must implement the `Comparable` interface):
`int compareTo(O)`: `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`,...).

Balanced Binary Tree (2/3)

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:

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

Balanced Binary Tree (2/3)

Two types of Tree object:

- `EmptyTree`
- `Node(E elem, Tree left, Tree right)`

Properties:

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

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 oreder is a right left depth first traversal

Balanced Binary Tree (3/3)

- Can you implement a Set using a BBT? Is it efficient?

Balanced Binary Tree (3/3)

- 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?

Balanced Binary Tree (3/3)

- Can you implement a Set using a BBT? Is it efficient?

$\Rightarrow$ `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?
$\Rightarrow$ rebalancing is important!

Balanced Binary Tree (3/3)

- Can you implement a Set using a BBT? Is it efficient?

$\Rightarrow$ `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?
$\Rightarrow$ rebalancing is important!

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

Balanced Binary Tree (3/3)

- 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 ?

Hashtable (2/3)

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 begining of the list

Hashtable (3/3)

What happens if two keys go into the same slot?

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?

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?

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

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?

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?
Can you implement Set using Hashtable? Is it efficient?

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?
Can you implement Set using Hashtable? Is it efficient?
$\Rightarrow$ See `HashSet` and `HashMap` in Java. Cons:
The iterators are in *unsepcified 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`

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:

```
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

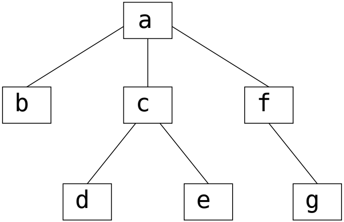
Remember tutorial 01, you have a class `Node`:

- `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:

```
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

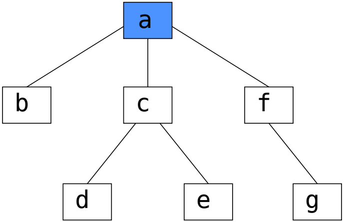


Call stack:

Code:

```
traverse(n)
```

Recursive tree traversal



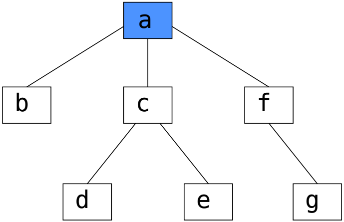
Call stack:

```
n:0x00abcbd1 ↦ {tag:'a', fs:0xaabc231,
                ns:null, ...}
```

Code:

```
if (n == null)  // false
```

Recursive tree traversal



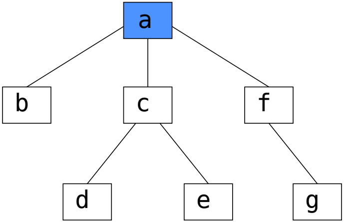
Call stack:

n:0x00abcbd1 $\mapsto$	{tag:'a', fs:0xaabc231, ns:null, ...}
nc:0xaabc231 $\mapsto$	{tag:'b', fs:null, ns:0x3942a2, ...}

Code:

```
nc = n.getFirstChild();
```

Recursive tree traversal



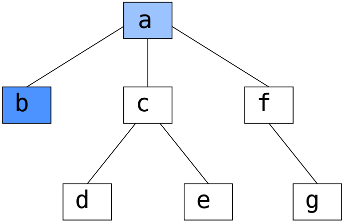
Call stack:

`n:0x00abcbd1` ↦	`{tag:'a', fs:0xaabc231, ns:null, ...}`
`nc:0xaabc231` ↦	`{tag:'b', fs:null, ns:0x3942a2, ...}`

Code:

```
while (nc != null) { //true
traverse(nc)
```

Recursive tree traversal



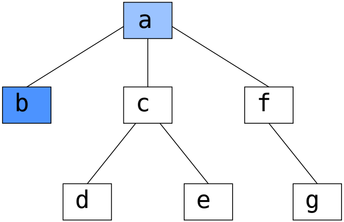
Call stack:

```
n:0x00abcbd1 ↦ {tag:'a', fs:0xaabc231,
                ns:null, ...}
nc:0xaabc231 ↦ {tag:'b', fs:null,
               ns:0x3942a2, ...}
n:0xaabc231 ↦  {tag:'b', fs:null,
                ns:0x3942a2, ...}
```

Code:

`if (n == null)` // false

Recursive tree traversal



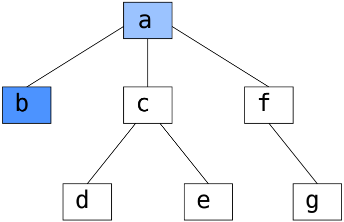
Call stack:

```
n:0x00abcbd1 ↦ {tag:'a', fs:0xaabc231,
                ns:null, ...}
nc:0xaabc231 ↦ {tag:'b', fs:null,
               ns:0x3942a2, ...}
n:0xaabc231 ↦  {tag:'b', fs:null,
                ns:0x3942a2, ...}
nc:null
```

Code:

```
nc = n.getFirstChild();
```

Recursive tree traversal



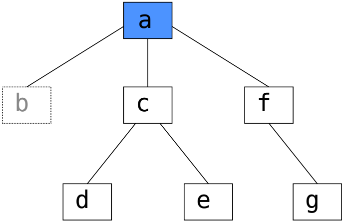
Call stack:

```
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:null
```

Code:

```
while (nc != null) { ...  }//false
return
```

Recursive tree traversal



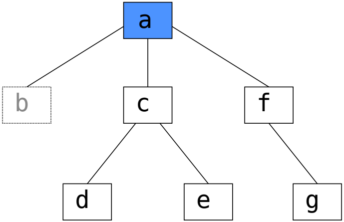
Call stack:

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

Code:

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

Recursive tree traversal



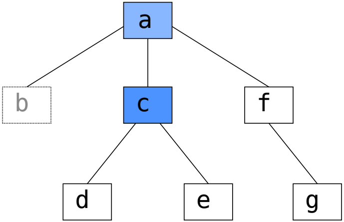
Call stack:

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

Code:

```
while (nc != null) { //true
traverse(nc)
```

Recursive tree traversal



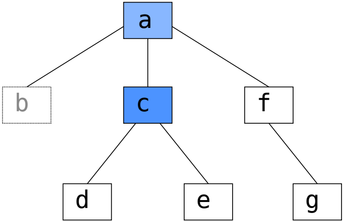
Call stack:

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

Code:

`if (n == null)` // false

Recursive tree traversal



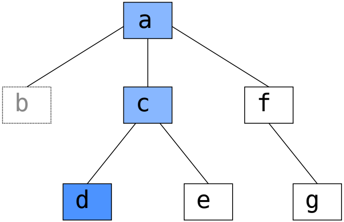
Call stack:

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

Code:

```
nc = n.getFirstChild();
```

Recursive tree traversal



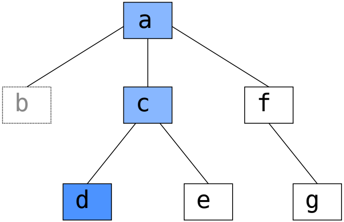
Call stack:

`n:0x00abcbd1` $\mapsto$	`{tag:'a', fs:0xaabc231, ns:null, ...}`
`nc:0x3942a2` $\mapsto$	`{tag:'c', fs:0x12af89, ns:0xbbf638, ...}`
`n:0x3942a2` $\mapsto$	`{tag:'c', fs:0x12af89, ns:0xbbf638, ...}`
`nc:0x12af89` $\mapsto$	`{tag:'d', fs:null, ns:0x48ab2f, ...}`

Code:

```
while (nc != null) { //true
traverse(nc)
```

Recursive tree traversal



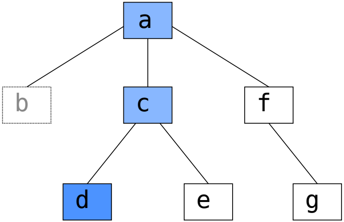
Call stack:

`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, ...}`

Code:

```
if (n == null)  // false
```

Recursive tree traversal



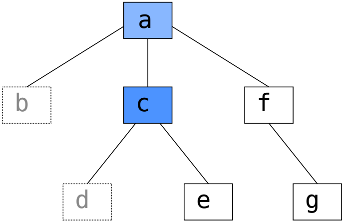
Call stack:

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

Code:

```
nc = n.getFirstChild();
```

Recursive tree traversal



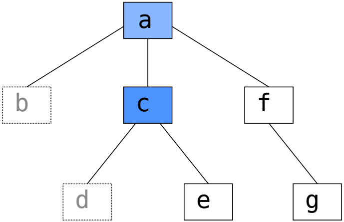
Call stack:

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:

```
while (nc != null) { ...  }//false
return
```

Recursive tree traversal



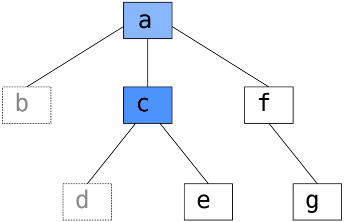
Call stack:

```
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:

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

Recursive tree traversal



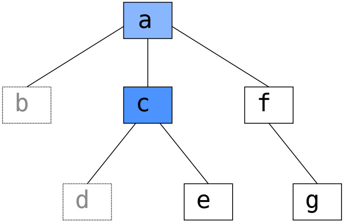
Call stack:

`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:

```
while (nc != null) { //true
traverse(nc)
```

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, ...}
```

Code: