Binary Search Trees

Advanced Algorithms and Data Structures - Lecture 3A

Venanzio Capretta

Thursday 15 October 2020

School of Computer Science, University of Nottingham

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$
with algorithms to perform these operations:

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$ with algorithms to perform these operations:

- Search a key in the collection

$$\texttt{search}\ 4\ \{7, 5, 1, 10, 4, 6, 9\} = \text{true}$$
$$\texttt{search}\ 3\ \{7, 5, 1, 10, 4, 6, 9\} = \text{false}$$

  In concrete applications, every number/key would be associated with a value and the search function would return that value

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$ with algorithms to perform these operations:

- Search a key in the collection

$$\texttt{search}\ 4\ \{7,5,1,10,4,6,9\} = \text{true}$$
$$\texttt{search}\ 3\ \{7,5,1,10,4,6,9\} = \text{false}$$

  In concrete applications, every number/key would be associated with a value and the search function would return that value

- Insert an new element in the collection

$$\texttt{insert}\ 3\ \{7,5,1,10,4,6,9\} = \{7,5,1,10,4,6,9,3\}$$

  (where it is inserted depends on the representation)

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$ with algorithms to perform these operations:

- Search a key in the collection

  $$\texttt{search}\,4\,\{7, 5, 1, 10, 4, 6, 9\} = \text{true}$$
  $$\texttt{search}\,3\,\{7, 5, 1, 10, 4, 6, 9\} = \text{false}$$

  In concrete applications, every number/key would be associated with a value and the search function would return that value

- Insert an new element in the collection

  $$\texttt{insert}\,3\,\{7, 5, 1, 10, 4, 6, 9\} = \{7, 5, 1, 10, 4, 6, 9, 3\}$$

  (where it is inserted depends on the representation)

- Delete an element from the collection

  $$\texttt{delete}\,4\,\{7, 5, 1, 10, 4, 6, 9\} = \{7, 5, 1, 10, 6, 9\}$$

Dictionaries

In practice elements of a dynamic sets will be pairs:

A key used for searching, a value to be returned

Such a dynamic set is also called a dictionary

Example: In a database of students, the key could be the ID number, the value the name of the student (and all other relevant data)

$$\{(7, \text{Monica}), (5, \text{Richard}), (1, \text{Fang}), (10, \text{Wei}),$$
$$(4, \text{Jan}), (6, \text{Femke}), (9, \text{Clara})\}$$

Dictionaries

In practice elements of a dynamic sets will be pairs:

A key used for searching, a value to be returned

Such a dynamic set is also called a dictionary

Example: In a database of students, the key could be the ID number, the value the name of the student (and all other relevant data)

$$\{(7, \text{Monica}), (5, \text{Richard}), (1, \text{Fang}), (10, \text{Wei}), \\ (4, \text{Jan}), (6, \text{Femke}), (9, \text{Clara})\}$$

Searching the data base with a key returns the value:

$$\texttt{search}\, 6\, \{(7, \text{Monica}), \ldots, (9, \text{Clara})\} = \text{Femke}$$

Dictionaries

In practice elements of a dynamic sets will be pairs:
A key used for searching, a value to be returned
Such a dynamic set is also called a dictionary

Example: In a database of students, the key could be the ID number, the value the name of the student (and all other relevant data)

$$\{(7, \text{Monica}), (5, \text{Richard}), (1, \text{Fang}), (10, \text{Wei}), (4, \text{Jan}), (6, \text{Femke}), (9, \text{Clara})\}$$

Searching the data base with a key returns the value:

$$\texttt{search}\, 6\, \{(7, \text{Monica}), \ldots, (9, \text{Clara})\} = \text{Femke}$$

For the study of the algorithms just the keys are relevant
I will describe the algorithms just using a set of keys
Exercise: Extend them to include the values

Implementations

Possible representations are:

Implementations

Possible representations are:

- UNORDERED LIST: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list

Implementations

Possible representations are:

- UNORDERED LIST: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list
- ORDERED LIST: $[1, 4, 5, 6, 7, 9, 10]$
  Searching still takes $O(n)$ time
  Just a bit better: I may stop early

Implementations

Possible representations are:

- UNORDERED LIST: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list
- ORDERED LIST: $[1, 4, 5, 6, 7, 9, 10]$
  Searching still takes $O(n)$ time
  Just a bit better: I may stop early
- RANDOM-ACCESS ORDERED ARRAY
  Fast search in $O(\log n)$ time
  But insertion and deletion still take $O(n)$ time
  Resizing of the array and shifting elements is necessary and costly

Implementations

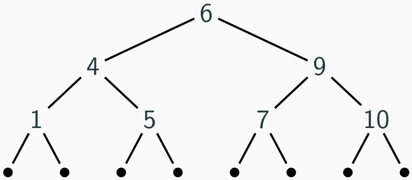
Possible representations are:

- UNORDERED LIST: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list
- ORDERED LIST: $[1, 4, 5, 6, 7, 9, 10]$
  Searching still takes $O(n)$ time
  Just a bit better: I may stop early
- RANDOM-ACCESS ORDERED ARRAY
  Fast search in $O(\log n)$ time
  But insertion and deletion still take $O(n)$ time
  Resizing of the array and shifting elements is necessary and costly

Can we find a data structure for which all three operations are efficient?

BST

Binary Search Trees



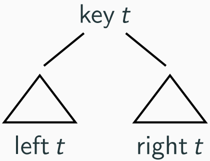
The operations of search, insert, delete
can be done in $O(k)$ time
where k is the depth of the tree

But tricky to keep k small: $k \sim O(\log n)$

There are more advanced variants that guarantee this: Red-Black Trees

The BST Property

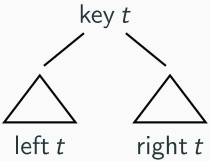
For every tree t, we use the notation:



(key t)	element in the root node
(left t)	left subtree of t
(right t)	right subtree of t

The BST Property

For every tree t, we use the notation:

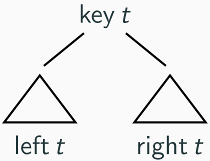


(key t)	element in the root node
(left t)	left subtree of t
(right t)	right subtree of t

The defining property of Binary Search Trees is:
All elements in (left t) are smaller than (key t) and
All elements in (right t) are larger than (key t)
(We assume that there are no repeated keys)

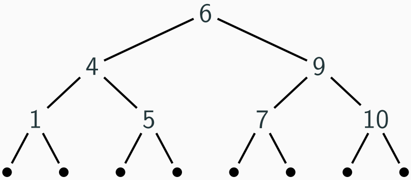
The BST Property

For every tree t, we use the notation:



(key t)	element in the root node
(left t)	left subtree of t
(right t)	right subtree of t

The defining property of Binary Search Trees is:
All elements in (left t) are smaller than (key t) and
All elements in (right t) are larger than (key t)
(We assume that there are no repeated keys)



$4, 1, 5 < 6$
$9, 7, 10 > 6$
recursively:
$1 < 4, 5 > 4$
$7 < 9, 10 > 9$

Trees with key-value pairs

In practical applications:
nodes will contain pairs $\langle k, v \rangle$ of
a key k and a value v

The key is used for searching

The value is the information we want to store

Trees with key-value pairs

In practical applications:
nodes will contain pairs $\langle k, v \rangle$ of
a key k and a value v

The key is used for searching
The value is the information we want to store

For example

- In a dictionary: keys = words; values = definitions
- In an address book: keys = names; values = addresses

Trees with key-value pairs

In practical applications:
nodes will contain pairs $\langle k, v \rangle$ of
a key k and a value v

The key is used for searching
The value is the information we want to store

For example

- In a dictionary: keys = words; values = definitions
- In an address book: keys = names; values = addresses

For the sake of the definition of the algorithms, we only use keys
Exercise: modify the algorithms with key-value pairs

The type of BSTs

BST is a type with two constructors:

The type of BSTs

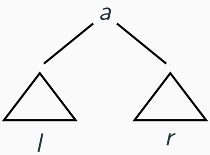
BST is a type with two constructors:

- Nil for leaves / empty trees

The type of BSTs

BST is a type with two constructors:

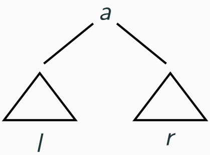
- Nil for leaves / empty trees
- Node for internal nodes
  containing elements from an ordered type Key
  (Node a l r) is the tree



The type of BSTs

BST is a type with two constructors:

- Nil for leaves / empty trees
- Node for internal nodes
  containing elements from an ordered type Key
  (Node a l r) is the tree



In functional programming it is defined as this inductive data type

```
data BST = Nil | Node Key BST BST
```

Component Functions

Functions to give the components
Defined by pattern-matching

Component Functions

Functions to give the components

Defined by pattern-matching

```
key :: BST → Key
key Nil = undefined
key (Node a l r) = a
```

Component Functions

Functions to give the components

Defined by pattern-matching

```
key :: BST → Key
key Nil = undefined
key (Node a l r) = a
```

```
left :: BST → BST
left Nil = undefined
left (Node a l r) = l
```

Component Functions

Functions to give the components

Defined by pattern-matching

```
key :: BST → Key
key Nil = undefined
key (Node a l r) = a
```

```
left :: BST → BST
left Nil = undefined
left (Node a l r) = l
```

```
right :: BST → BST
right Nil = undefined
right (Node a l r) = r
```

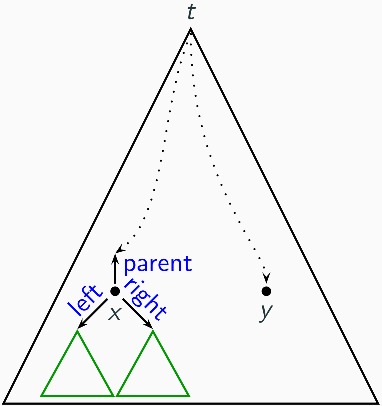
Imperative Implementation

In imperative programming, we use pointers to nodes, with methods giving the two children and the parent (See Ch.12 of IA)

(We can do it in functional programming too: using paths or defining them directly with those fields (Exercise))

Global Objects, Local Pointers

Work with a global tree t and with pointers x, y to subtrees/nodes



We can move around the tree with operations on pointers

- (parent x) the immediate precursor of x; Nil if x is the root
- (left x) and (right x) the children of x; Nil if they are leaves

Searching

Searching a tree t for a key k is done by following a single path
At each node x:

- If $k = \text{key}\, x$, we have found it!
- If $k < \text{key}\, x$, go to the left child of x;
- If $k > \text{key}\, x$, go to the right child of x.

Searching

Searching a tree t for a key k is done by following a single path
At each node x:

- If $k = \text{key}\, x$, we have found it!
- If $k < \text{key}\, x$, go to the left child of x;
- If $k > \text{key}\, x$, go to the right child of x.

Functional/Recursive Version:

```
search :: Key → BST → Bool
search k Nil = False
search k (Node x l r) =
     (k == x) || if (k < x) then search k l
                            else search k r
```

Imperative/Iterative Version

```
search (t,k):
  x := t
  while x /= Nil and k /= (key x)
    if k < (key x) then x := (left x)
                   else x := (right x)
  return k == (key x)
```

Imperative/Iterative Version

```
search (t,k):
  x := t
  while x /= Nil and k /= (key x)
    if k < (key x) then x := (left x)
                   else x := (right x)
  return k == (key x)
```

This algorithm just returns a Boolean value:
true if k is present in the tree, false otherwise

Exercise: Modify the algorithm for trees containing key-values pairs;
if the key is found, it must return the corresponding value.

Imperative/Iterative Version

```
search (t,k):
  x := t
  while x /= Nil and k /= (key x)
    if k < (key x) then x := (left x)
                   else x := (right x)
  return k == (key x)
```

This algorithm just returns a Boolean value:
true if k is present in the tree, false otherwise

Exercise: Modify the algorithm for trees containing key-values pairs; if the key is found, it must return the corresponding value.

Complexity: The search algorithm starts at the root and follows a specific path, until it reaches a node that matches the search key or a leaf. The time complexity is $O(h)$ where h is the height of the tree.

In-Order Traversal

Given a BST, generate a list containing all its elements in order

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller than the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller than the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

```
treeList :: BST → [Key]
treeList Nil = []
treeList (Node x l r) = treeList l ++ x : treeList r
```

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller than the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

```
treeList :: BST → [Key]
treeList Nil = []
treeList (Node x l r) = treeList l ++ x : treeList r
```

Complexity: The algorithm visits every node just once.
It runs in $O(n)$ time, where n is the number of elements.

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller than the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

```
treeList :: BST → [Key]
treeList Nil = []
treeList (Node x l r) = treeList l ++ x : treeList r
```

Complexity: The algorithm visits every node just once.
It runs in $O(n)$ time, where n is the number of elements.

Exercise: Implement the `insert` operation
Hint: Insert in a leaf in the correct position

Minimum and Maximum

The minimum element in a binary search tree is the leftmost one, the maximum is the rightmost

```
minimum :: BST → Maybe Key
minimum Nil = Nothing
minimum (Node x Nil _) = Just x
minimum (Node x l _) = minimum l
```

Minimum and Maximum

The minimum element in a binary search tree is the leftmost one, the maximum is the rightmost

```
minimum :: BST → Maybe Key
minimum Nil = Nothing
minimum (Node x Nil _) = Just x
minimum (Node x l _) = minimum l
```

In iterative style:

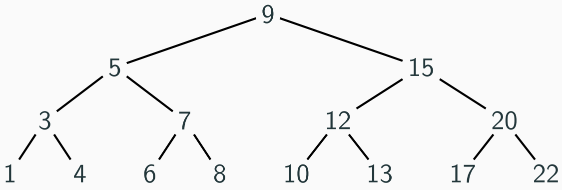
```
minimum (x):
  while (left x) /= Nil
    x := (left x)
  return (key x)
```

The maximum is defined similarly, going right instead of left.

Complexity: $O(h)$ where h is the height of the tree.

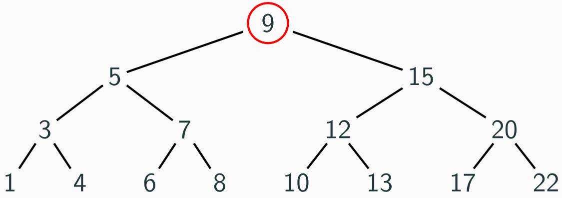
Delete

When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property



Delete

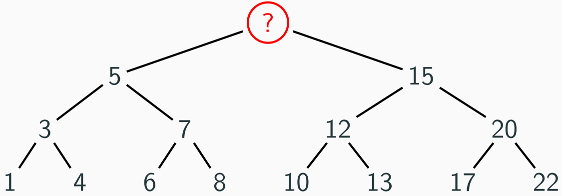
When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property



To delete the root 9:

Delete

When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property

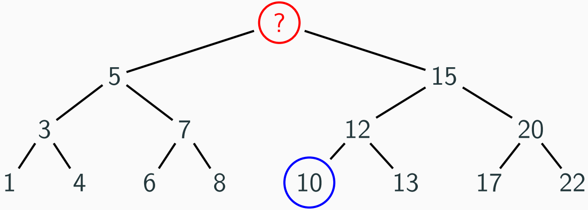


To delete the root 9:

- Remove the root

Delete

When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property



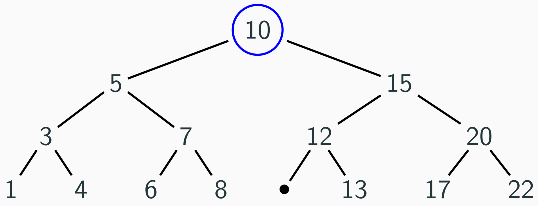
To delete the root 9:

- Remove the root
- Find the minimum of the right child

(We could also do it with the maximum of the left child)

Delete

When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property



To delete the root 9:

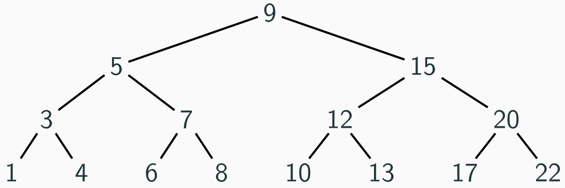
- Remove the root
- Find the minimum of the right child
- Place it at the root

(We could also do it with the maximum of the left child)

Predecessor and Successor

Extra Operations:

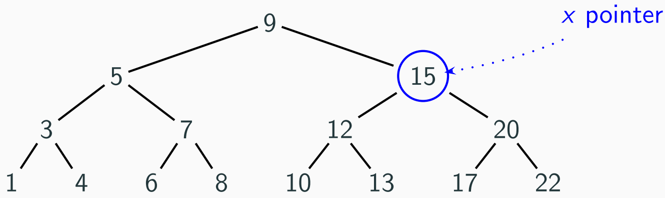
Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.



Predecessor and Successor

Extra Operations:

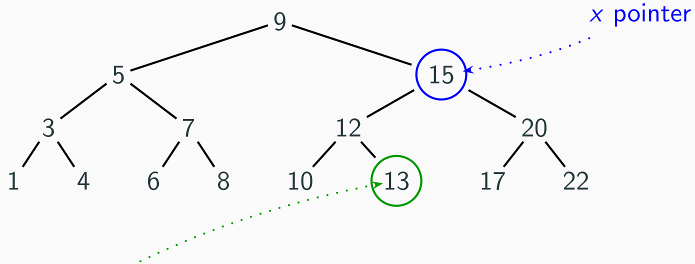
Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.



Predecessor and Successor

Extra Operations:

Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.

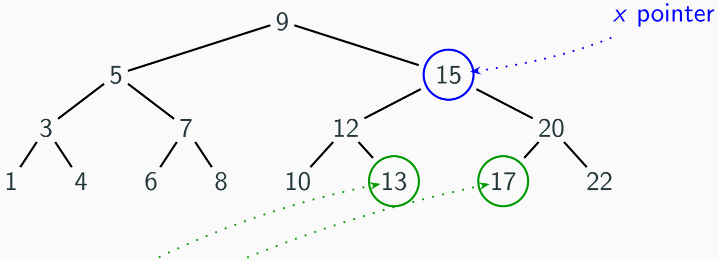


- The predecessor is the maximum of the left subtree of x;

Predecessor and Successor

Extra Operations:

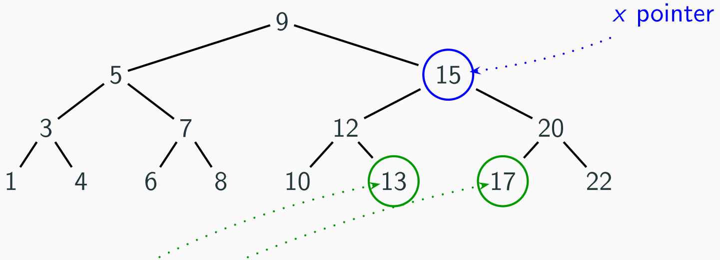
Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.



- The predecessor is the maximum of the left subtree of x;
- The successor is the minimum of the right subtree of x;

Predecessor and Successor

Extra Operations:
Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.



- The predecessor is the maximum of the left subtree of x;
- The successor is the minimum of the right subtree of x;

Exercise: What if the node doesn't have a left child (or a right child)?