# BINARY SEARCH TREES

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

# Agenda

1 Trees

2 Dictionaries implemented as BSTs

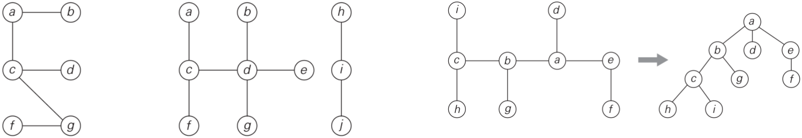
3 Binary tree traversals

4 Bibliography

# Trees[1]

A free tree: a connected acyclic graph

- Forest: acyclic graph not necessarily connected
- Rooted tree (root typically on the top)



Applications: implement dictionaries, fault analysis, etc.

[1] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

# Trees: terminology for rooted trees

- Parent
- Siblings
- Subtree
- Internal vertices
- Ancestors / descendants (proper)

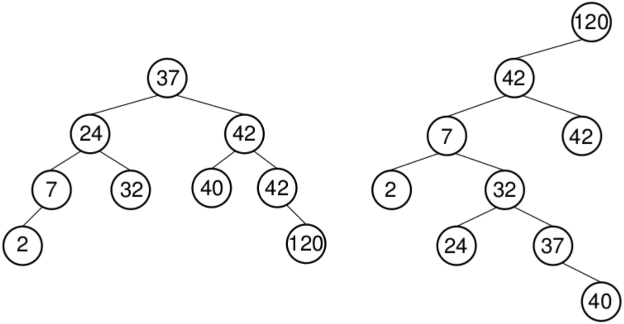
# Trees: terminology for rooted trees

- Leaf: no children
- Depth/level of $v$: length of the unique path from the root to $v$
- Height: length of the longest unique path from the node to a leaf
- $m$-ary tree: every internal vertex has no more than $m$ children
  - Complete $m$-ary tree: levels filled from top to bottom, left to right
  - Full $m$-ary tree: exactly $m$ children
  - $m$-ary tree, where $m = 2$: binary tree
- Ordered tree
  - Ordered binary tree: binary search tree (BST)
- Balanced tree

# Binary search trees[2]

Left tree: insertion order = 37, 24, 42, 7, 2, 40, 42, 32, 120

Right tree: insertion order = 120, 42, 42, 7, 2, 32, 37, 24, 40



[2] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

# Agenda

1 Trees

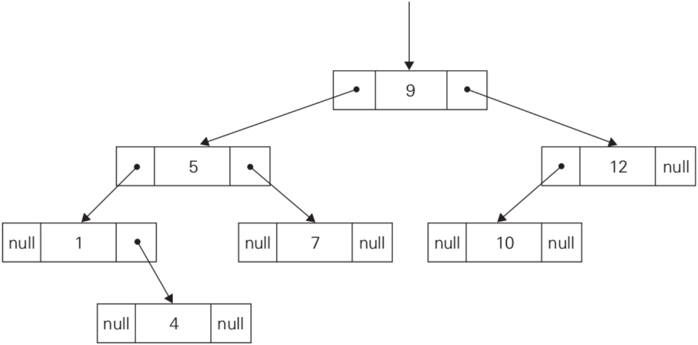
2 Dictionaries implemented as BSTs

3 Binary tree traversals

4 Bibliography

# Dictionaries implemented as BSTs[3]

Typical implementation: based on references (pointers)



[3] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

# Dictionaries implemented as BSTs

Composite type (BSTNode):

```
Key key;
E element;
BSTNode left;                    // left child
BSTNode right ;                  // right child
```

**Algorithm:** BSTNode create_bstnode(Key k, E e)

```
$n.key \leftarrow k$;
$n.element \leftarrow e$;
$n.left \leftarrow n.right \leftarrow NULL$;
return $n$;
```

# Dictionaries implemented as BSTs

Composite type (BST):

```
BSTNode *root*;
int *nodecount*;                    // number of elements
```

**Algorithm:** BST create_bst()

```
$bst.root \leftarrow NULL$;
$bst.nodecount \leftarrow 0$;
**return** $bst$;
```

# Dictionaries implemented as BSTs

```
Algorithm: E find(BST bst, Key k)

return $findhelp(bst.root, k)$;
```

```
Algorithm: E findhelp(BSTNode rt, Key k)

if $rt = NULL$ then  return $NULL$ ;
if $rt.key > k$ then
    return $findhelp(rt.left, k)$;
else if $rt.key = k$ then
    return $rt.element$;
else
    return $findhelp(rt.right, k)$;
```

# Dictionaries implemented as BSTs

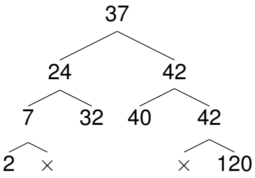
**Algorithm:** E find(BST bst, Key k)

```
return $findhelp(bst.root, k)$;
```

**Algorithm:** E findhelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$ ;
if $rt.key > k$ then
    return $findhelp(rt.left, k)$;
else if $rt.key = k$ then
    return $rt.element$;
else
    return $findhelp(rt.right, k)$;
```

find(_,32)

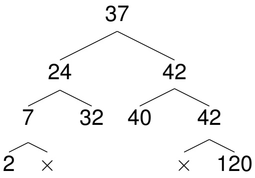


# Dictionaries implemented as BSTs

```
Algorithm: E find(BST bst, Key k)
return findhelp(bst.root, k);
```

```
Algorithm: E findhelp(BSTNode rt, Key k)
if $rt = NULL$ then return $NULL$ ;
if $rt.key > k$ then
    return $findhelp(rt.left, k)$;
else if $rt.key = k$ then
    return $rt.element$;
else
    return $findhelp(rt.right, k)$;
```

find(_,32)

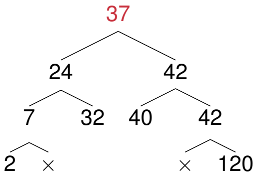


# Dictionaries implemented as BSTs

```
Algorithm: E find(BST bst, Key k)
return $findhelp(bst.root, k)$;
```

```
Algorithm: E findhelp(BSTNode rt, Key k)
if $rt = NULL$ then return $NULL$ ;
if $rt.key > k$ then
    return $findhelp(rt.left, k)$;
else if $rt.key = k$ then
    return $rt.element$;
else
    return $findhelp(rt.right, k)$;
```

find(_,32)



# Dictionaries implemented as BSTs

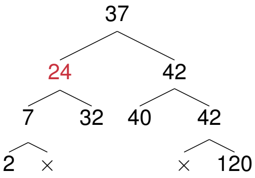
**Algorithm:** E find(BST bst, Key k)

```
return $findhelp(bst.root, k)$;
```

**Algorithm:** E findhelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$ ;
if $rt.key > k$ then
    return $findhelp(rt.left, k)$;
else if $rt.key = k$ then
    return $rt.element$;
else
    return $findhelp(rt.right, k)$;
```

find(_,32)

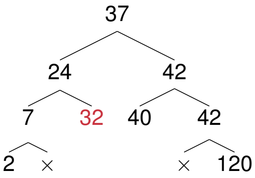


# Dictionaries implemented as BSTs

```
Algorithm: E find(BST bst, Key k)
return findhelp(bst.root, k);
```

```
Algorithm: E findhelp(BSTNode rt, Key k)
if $rt = NULL$ then return NULL ;
if $rt.key > k$ then
    return findhelp(rt.left, k);
else if $rt.key = k$ then
    return rt.element;
else
    return findhelp(rt.right, k);
```

find(_,32)



Returning element associated with $k = 32$

# Dictionaries implemented as BSTs

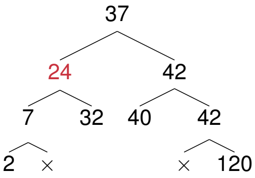
```
Algorithm: E find(BST bst, Key k)

return $findhelp(bst.root, k)$;
```

```
Algorithm: E findhelp(BSTNode rt, Key k)

if $rt = NULL$ then return $NULL$ ;
if $rt.key > k$ then
    return $findhelp(rt.left, k)$;
else if $rt.key = k$ then
    return $rt.element$;
else
    return $findhelp(rt.right, k)$;
```

`find(_,32)`



Returning element associated with $k = 32$

# Dictionaries implemented as BSTs

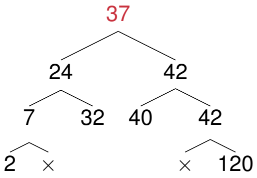
```
Algorithm: E find(BST bst, Key k)

return findhelp(bst.root, k);
```

```
Algorithm: E findhelp(BSTNode rt, Key k)

if $rt = NULL$ then return $NULL$ ;
if $rt.key > k$ then
    return findhelp(rt.left, k);
else if $rt.key = k$ then
    return rt.element;
else
    return findhelp(rt.right, k);
```

find(_,32)



Returning element associated with $k = 32$

# Dictionaries implemented as BSTs

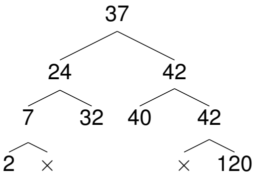
**Algorithm:** E find(BST bst, Key k)

```
return $findhelp(bst.root, k)$;
```

**Algorithm:** E findhelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$ ;
if $rt.key > k$ then
    return $findhelp(rt.left, k)$;
else if $rt.key = k$ then
    return $rt.element$;
else
    return $findhelp(rt.right, k)$;
```

find(_,32)



Returning element associated with $k = 32$

# Dictionaries implemented as BSTs

**Algorithm:** void insert(BST bst, Key k, E e)

```
$bst.root \leftarrow inserthelp(bst.root, k, e)$;
$bst.nodecount{+}{+}$;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return $create\_bstnode(k, e)$ ;
if $rt.key > k$ then
    $rt.left \leftarrow inserthelp(rt.left, k, e)$;
else
    $rt.right \leftarrow inserthelp(rt.right, k, e)$;
return $rt$;
```

Important: repeated keys
go to the right subtree

# Dictionaries implemented as BSTs

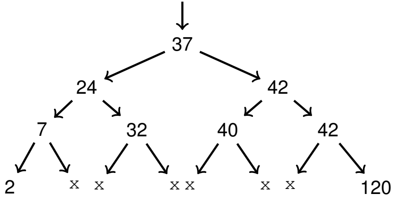
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode(k, e) ;
if $rt.key > k$ then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

`insert(_,25,_)`



# Dictionaries implemented as BSTs

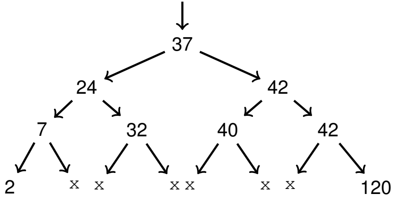
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode(k, e) ;
if $rt.key > k$ then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

insert(_,25,_)



# Dictionaries implemented as BSTs

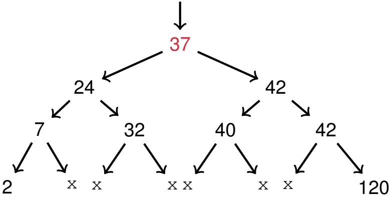
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if rt = NULL then return create_bstnode(k, e) ;
if rt.key > k then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

insert(_,25,_)



# Dictionaries implemented as BSTs

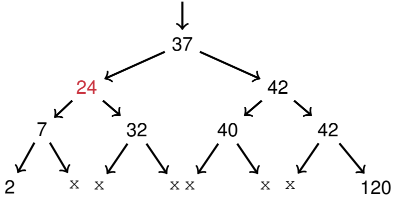
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode(k, e) ;
if $rt.key > k$ then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

`insert(_,25,_)`



# Dictionaries implemented as BSTs

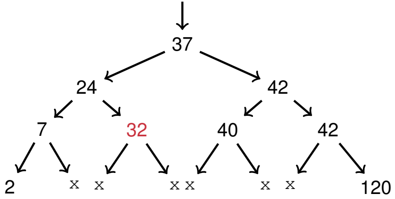
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode(k, e) ;
if $rt.key > k$ then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

insert(_,25,_)



# Dictionaries implemented as BSTs

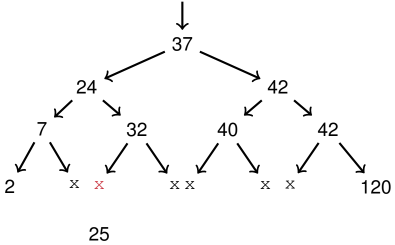
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if rt = NULL then return create_bstnode(k, e) ;
if rt.key > k then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

insert(_,25,_)



Returning the reference to the new node

# Dictionaries implemented as BSTs

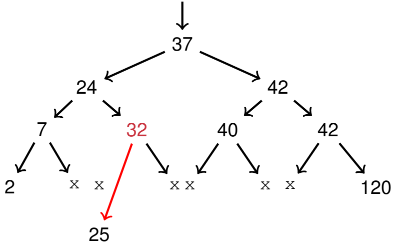
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode($k, e$) ;
if $rt.key > k$ then
    $rt.left \leftarrow inserthelp(rt.left, k, e)$;
else
    $rt.right \leftarrow inserthelp(rt.right, k, e)$;
return $rt$;
```

insert(_,25,_)



Returning the reference to the current node

# Dictionaries implemented as BSTs

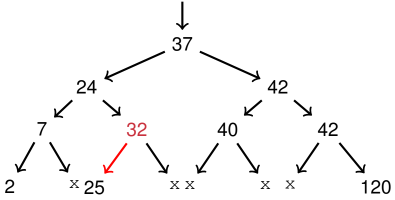
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode(k, e) ;
if $rt.key > k$ then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

`insert(_,25,_)`



Better presented this way

Returning the reference to the current node

# Dictionaries implemented as BSTs

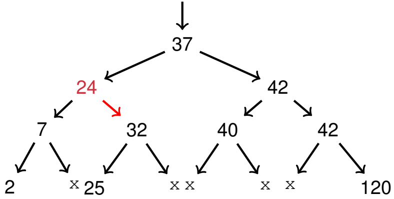
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode(k, e) ;
if $rt.key > k$ then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

`insert(_,25,_)`



Returning the reference to the current node

# Dictionaries implemented as BSTs

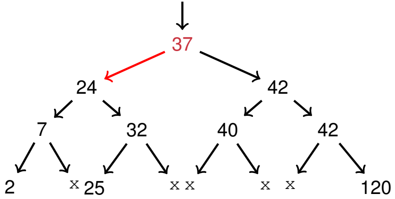
**Algorithm:** void insert(BST bst, Key k, E e)

```
bst.root ← inserthelp(bst.root, k, e);
bst.nodecount++;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return create_bstnode(k, e) ;
if $rt.key > k$ then
    rt.left ← inserthelp(rt.left, k, e);
else
    rt.right ← inserthelp(rt.right, k, e);
return rt;
```

insert(_,25,_)



Returning the reference to the current node (root)

# Dictionaries implemented as BSTs

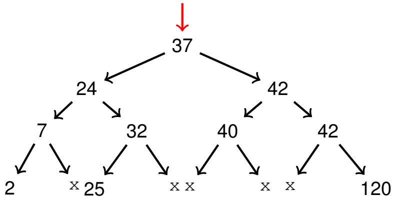
**Algorithm:** void insert(BST bst, Key k, E e)

```
$bst.root \leftarrow inserthelp(bst.root, k, e)$;
$bst.nodecount{++}$;
```

**Algorithm:** BSTNode inserthelp(BSTNode rt, Key k, E e)

```
if $rt = NULL$ then return $create\_bstnode(k, e)$ ;
if $rt.key > k$ then
    $rt.left \leftarrow inserthelp(rt.left, k, e)$;
else
    $rt.right \leftarrow inserthelp(rt.right, k, e)$;
return $rt$;
```

`insert(_,25,_)`



Updating the reference to the root node

# Dictionaries implemented as BSTs

**Algorithm:** E remove(BST bst, Key k)

```
E $temp \leftarrow findhelp(bst.root, k)$;
if $temp \neq NULL$ then
    $bst.root \leftarrow removehelp(bst.root, k)$;
    $bst.nodecount$--;
return $temp$;
```

Centro de Informática UFPE

Universidade Federal de Pernambuco

# Dictionaries implemented as BSTs

**Algorithm:** BSTNode removehelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return $rt.right$ ;
    else if $rt.right = NULL$ then return $rt.left$ ;
    else
        BSTNode $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow deletemin(rt.right)$;
return $rt$;
```

# Dictionaries implemented as BSTs

```
Algorithm: BSTNode removehelp(BSTNode rt, Key k)

if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return $rt.right$;
    else if $rt.right = NULL$ then return $rt.left$ ;
    else
        BSTNode $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow deletemin(rt.right)$;
return $rt$;
```

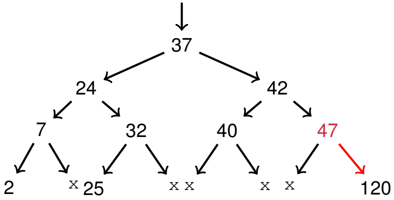
Let *rt* be the node to be removed, three cases:

(i) $rt.left = NULL$,

(ii) $rt.right = NULL$, and

(iii) otherwise.

`remove(_,47)`



# Dictionaries implemented as BSTs

**Algorithm:** BSTNode removehelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return $rt.right$ ;
    else if $rt.right = NULL$ then return $rt.left$ ;
    else
        BSTNode $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow deletemin(rt.right)$;
return $rt$;
```

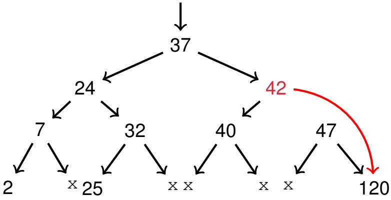
Let *rt* be the node to be removed, three cases:

(i) $rt.left = NULL$,

(ii) $rt.right = NULL$, and

(iii) otherwise.

`remove(_,47)`



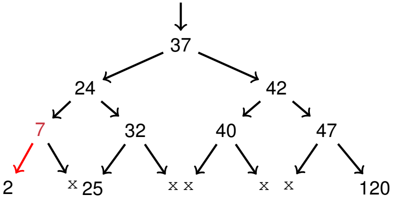
# Dictionaries implemented as BSTs

**Algorithm:** BSTNode removehelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow$
      $removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return
      $rt.right$ ;
    else if $rt.right = NULL$ then
      return $rt.left$;
    else
        BSTNode
          $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow$
          $deletemin(rt.right)$;
return $rt$;
```

Let *rt* be the node to be removed, three cases:
(i) $rt.left = NULL$,
(ii) $rt.right = NULL$, and
(iii) otherwise.

`remove(_,7)`



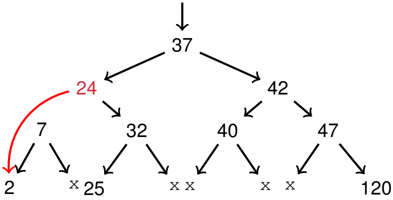
# Dictionaries implemented as BSTs

**Algorithm:** BSTNode removehelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return $rt.right$ ;
    else if $rt.right = NULL$ then return $rt.left$ ;
    else
        BSTNode $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow deletemin(rt.right)$;
return $rt$;
```

Let *rt* be the node to be removed, three cases:
(i) $rt.left = NULL$,
(ii) $rt.right = NULL$, and
(iii) otherwise.

`remove(_,7)`



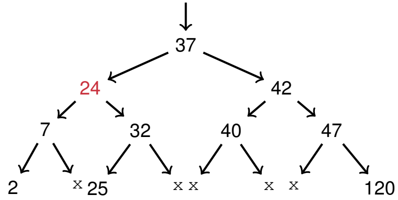
# Dictionaries implemented as BSTs

**Algorithm:** BSTNode removehelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return $rt.right$ ;
    else if $rt.right = NULL$ then return $rt.left$ ;
    else
        BSTNode $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow deletemin(rt.right)$;
return $rt$;
```

Let *rt* be the node to be removed, three cases:
(i) $rt.left = NULL$,
(ii) $rt.right = NULL$, and
(iii) otherwise.

`remove(_,24)`



*temp* = reference to node 25

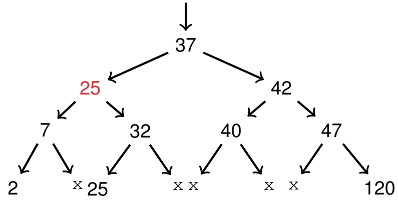
# Dictionaries implemented as BSTs

**Algorithm:** BSTNode removehelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return $rt.right$ ;
    else if $rt.right = NULL$ then return $rt.left$ ;
    else
        BSTNode $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow deletemin(rt.right)$;
return $rt$;
```

Let *rt* be the node to be removed, three cases:
(i) $rt.left = NULL$,
(ii) $rt.right = NULL$, and
(iii) otherwise.

`remove(_,24)`



*temp* = reference to node 25

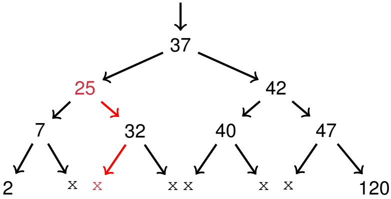
# Dictionaries implemented as BSTs

**Algorithm:** BSTNode removehelp(BSTNode rt, Key k)

```
if $rt = NULL$ then return $NULL$;
if $rt.key > k$ then
    $rt.left \leftarrow removehelp(rt.left, k)$;
else if $rt.key < k$ then
    $rt.right \leftarrow removehelp(rt.right, k)$;
else
    if $rt.left = NULL$ then return $rt.right$ ;
    else if $rt.right = NULL$ then return $rt.left$ ;
    else
        BSTNode $temp \leftarrow getmin(rt.right)$;
        $rt.element \leftarrow temp.element$;
        $rt.key \leftarrow temp.key$;
        $rt.right \leftarrow deletemin(rt.right)$;
return $rt$;
```

Let *rt* be the node to be removed, three cases:
(i) $rt.left = NULL$,
(ii) $rt.right = NULL$, and
(iii) otherwise.

`remove(_,24)`



*temp* = reference to node 25

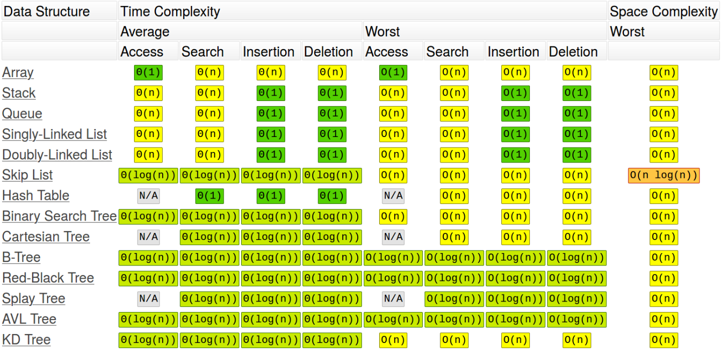
# Dictionaries implemented as BSTs

```
Algorithm: BSTNode getmin(BSTNode rt)
if $rt.left = NULL$ then return $rt$;
return $getmin(rt.left)$;
```

```
Algorithm: BSTNode deletemin(BSTNode rt)
if $rt.left = NULL$ then return $rt.right$;
$rt.left \leftarrow deletemin(rt.left)$;
return $rt$;
```

# Asymptotic efficiency of BSTs[4]

## Average case: considering a balanced BST



| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Queue | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Doubly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| B-Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| KD Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

[4] Source: http://bigocheatsheet.com/

# Agenda

1 Trees

2 Dictionaries implemented as BSTs

3 Binary tree traversals

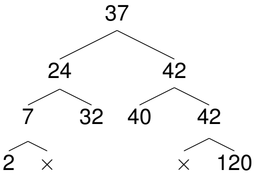
4 Bibliography

# Binary tree traversals: preorder

The root is processed before the left and the right subtrees

**Algorithm:** void preorder(BSTNode rt)

```
if $rt \neq NULL$ then
    // do something with $rt$
    preorder($rt.left$);
    preorder($rt.right$);
```



Let do something be printing the root's key

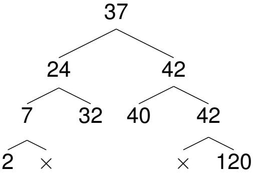
$37, 24, 7, 2, 32, 42, 40, 42, 120$

# Binary tree traversals: inorder

The root is processed after the left subtree, but before the right subtree

**Algorithm:** void inorder(BSTNode rt)

```
if $rt \neq NULL$ then
    inorder($rt.left$);
    // do something with $rt$
    inorder($rt.right$);
```



Let do something be printing the root's key

2, 7, 24, 32, 37, 40, 42, 42, 120

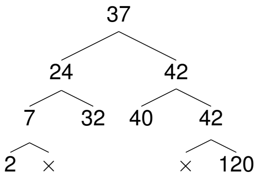
An inorder traversal visits the keys in a non-decreasing order

# Binary tree traversals: posorder

The root is processed after the left and the right subtrees

```
Algorithm: void posorder(BSTNode rt)

if $rt \neq NULL$ then
    posorder($rt.left$);
    posorder($rt.right$);
    // do something with $rt$
```



Let do something be printing the root's key

2, 7, 32, 24, 40, 120, 42, 42, 37

# Agenda

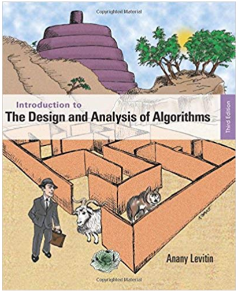
1 Trees

2 Dictionaries implemented as BSTs

3 Binary tree traversals

4 Bibliography

# Bibliography



**Chapter 1 (pp. 31–35)**
**Chapter 4 (pp. 163–164)**
**Chapter 5 (pp. 182–185)**
**Anany Levitin.**
*Introduction to the Design and Analysis of Algorithms*.
3rd edition. Pearson. 2011.

**Chapter 5 (pp. 145–170)**
**Clifford Shaffer.**
*Data Structures and Algorithm Analysis*. Dover, 2013.

Centro de Informática
UFPE

Universidade
Federal
de Pernambuco

# BINARY SEARCH TREES

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Centro de Informática
UFPE

Universidade Federal de Pernambuco