Graded Assignment 3 – DSA

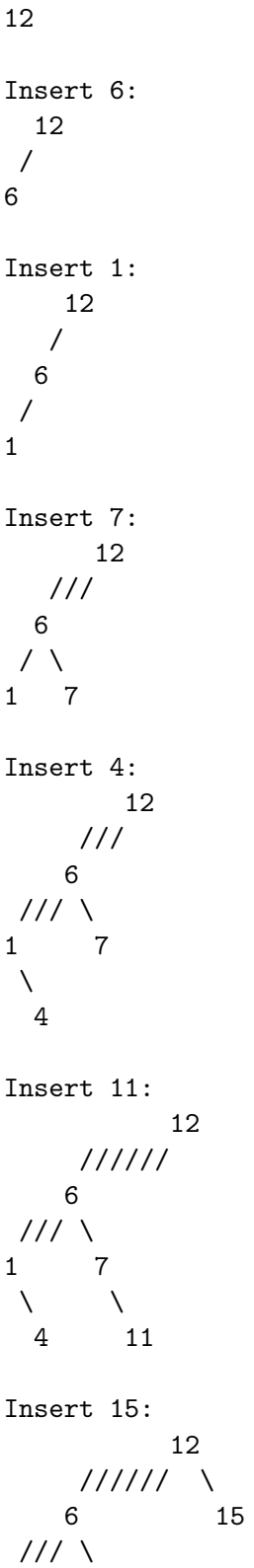
Claudio Maggioni

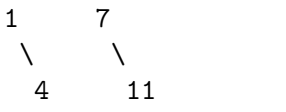
May 14, 2019

Contents

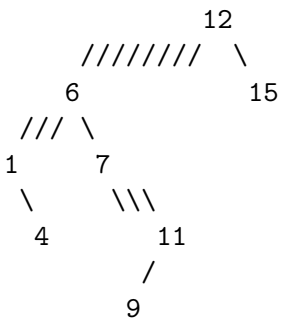
Listings

1 Exercise 1

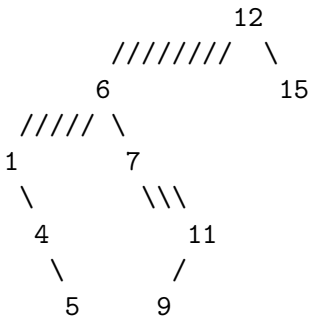

12
Insert 6:
12
/
6
Insert 1:
12
/
6
/
1
Insert 7:
12
///
6
/ \
1 7
Insert 4:
12
///
6
/// \
1 7
\
4
Insert 11:
12
//////
6
/// \
1 7
\ \
4 11
Insert 15:
12
////// \
6 15
/// \



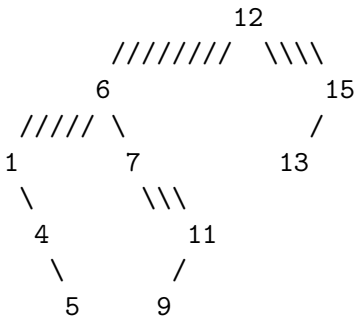
Insert 9:



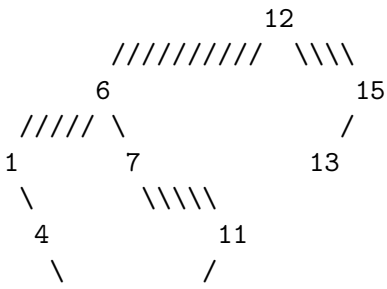
Insert 5:

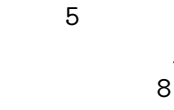


Insert 13:

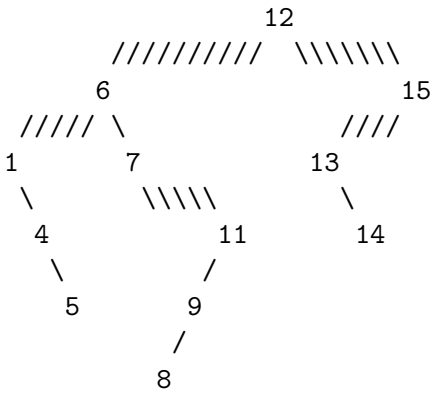


Insert 8:

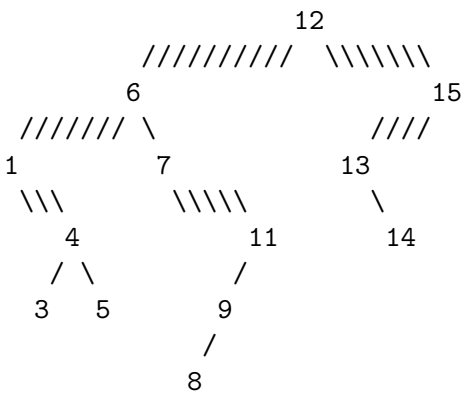




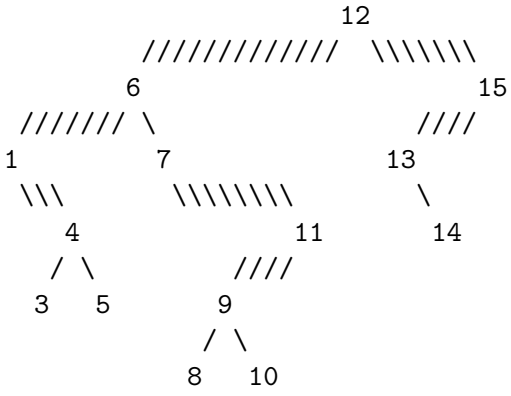
Insert 14:



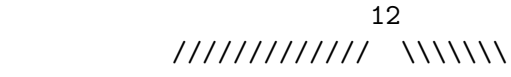
Insert 3:

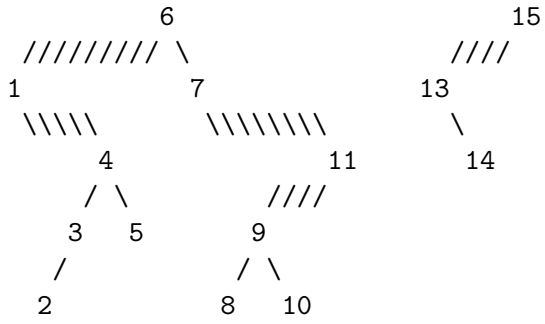


Insert 10:

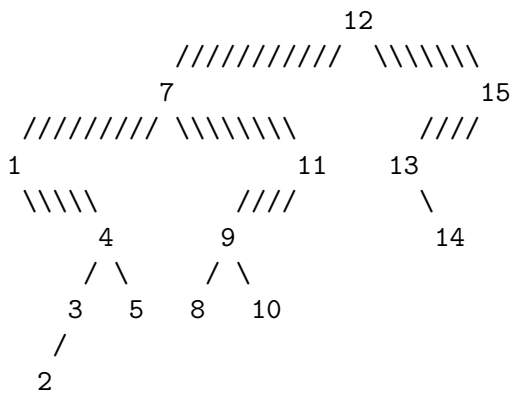


Insert 2:

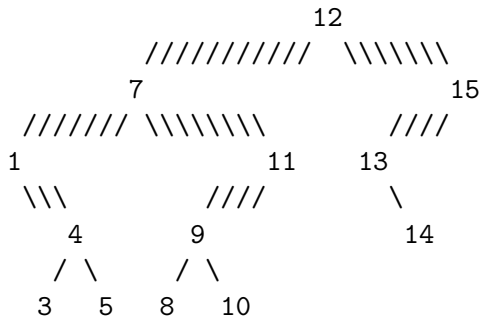




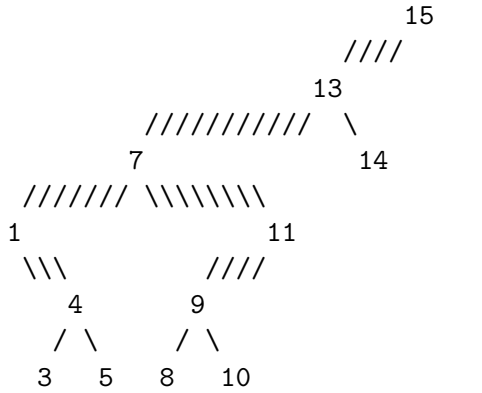
Delete 6:



Delete 2:



Delete 12:



The following printout was obtained by running the following BST implementation. with the following command:

```
./tree.py 12 6 1 7 4 11 15 9 5 13 8 14 3 10 2 \| 6 2 12
```

```
#!/usr/bin/env python3
# vim: set ts=2 sw=2 et tw=80:

import sys

class Node:
  def __init__(self, k):
    self.key = k
    self.left = None
    self.right = None
    self.parent = None

  def set_left(self, kNode):
    kNode.parent = self
    self.left = kNode

  def set_right(self, kNode):
    kNode.parent = self
    self.right = kNode


def search(tree, k):
  if tree is None:
    return None
  elif tree.key == k:
    return tree
  elif k < tree.key:
    return search(tree.left, k)
  else:
    return search(tree.right, k)


def insert(t, k):
  insert_node(t, Node(k))


def insert_node(t, node):
  if node.key < t.key:
    if t.left is None:
```

```
      t.set_left(node)
    else:
      insert_node(t.left, node)
  else:
    if t.right is None:
      t.set_right(node)
    else:
      insert_node(t.right, node)


def root_insert(t, k):
  if t is None:
    return k
  if k.key > t.key:
    t.set_right(root_insert(t.right, k))
    return left_rotate(t)
  else:
    t.set_left(root_insert(t.left, k))
    return right_rotate(t)


def unlink_me(node, to_link):
  if node.parent == None:
    tr = node
    node.key = to_link.key
    node.left = to_link.left
    node.right = to_link.right
    return node
  elif node.parent.left == node:
    node.parent.left = to_link
    return to_link
  else:
    node.parent.right = to_link
    return to_link


def delete(t, k):
  to_delete = search(t, k)
  if to_delete is None:
    return
  elif to_delete.left is None:
    unlink_me(to_delete, to_delete.right)
  elif to_delete.right is None:
    unlink_me(to_delete, to_delete.left)
  else:
    if abs(to_delete.left.key - to_delete.key) < abs(to_delete.right.key -
```

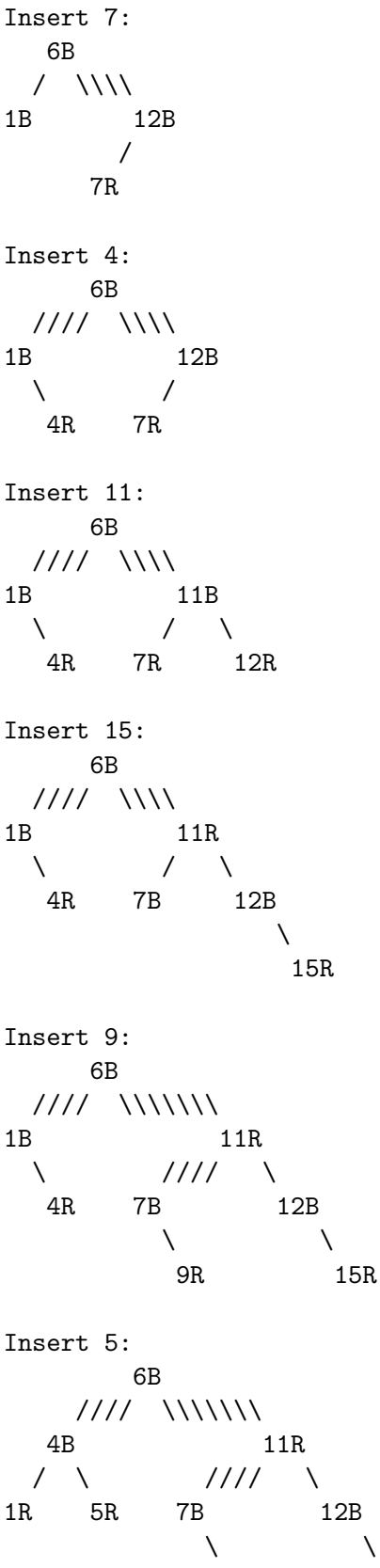
```
      to_delete.key):
    ins = to_delete.right
    new_branch = unlink_me(to_delete, to_delete.left)
    insert_node(new_branch, ins)
  else:
    ins = to_delete.left
    new_branch = unlink_me(to_delete, to_delete.right)
    insert_node(new_branch, ins)


if __name__ == "__main__":
  args = [x for x in sys.argv[1:]]
  T = Node(int(args[0]))
  for i in range(1, len(args)):
    if args[i] == '|':
      break
    print_tree(T)
    print("\nInsert " + str(args[i]) + ":")
    insert(T, int(args[i]))
  for i in range(i+1, len(args)):
    print_tree(T)
    print("\nDelete " + str(args[i]) + ":")
    delete(T, int(args[i]))
  print_tree(T)
```

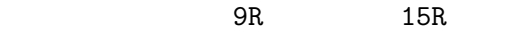
Listing 1: BST implementation

2 Exercise 2

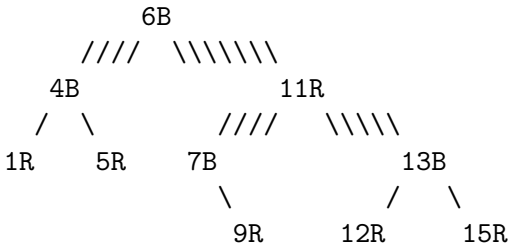
2.1 Point A

```
12B

Insert 6:
   12B
  /
6R

Insert 1:
   6B
  /  \
1R    12R
```

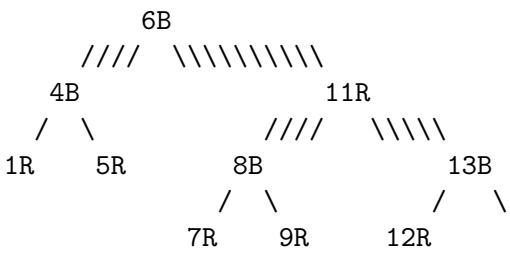

Insert 7:
6B
1B
12B
7R
Insert 4:
6B
1B
12B
4R
7R
Insert 11:
6B
1B
11B
4R
7R
12R
Insert 15:
6B
1B
11R
4R
7B
12B
15R
Insert 9:
6B
1B
11R
4R
7B
12B
9R
15R
Insert 5:
6B
4B
11R
1R
5R
7B
12B



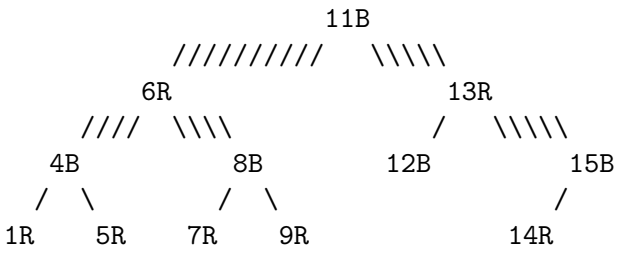
Insert 13:



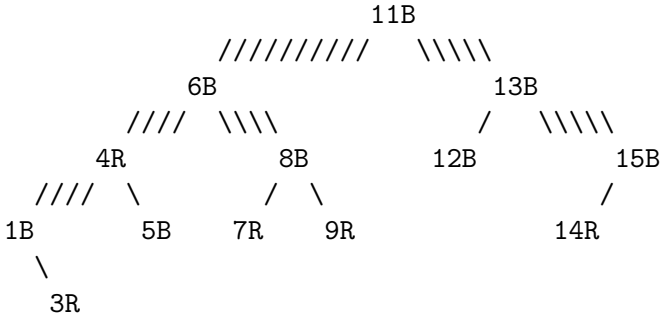
Insert 8:



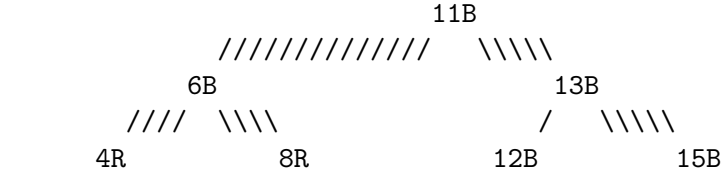
Insert 14:



Insert 3:



Insert 10:



```
  ////  \        /  \                    /
1B       5B    7B    9B                14R
  \                    \
   3R                   10R
```

Insert 2:

```
                                11B
                  //////////////   \\\\\
                6B                      13B
              ////  \\\\                /   \\\\\
           4R           8R           12B         15B
      ////  \         /  \                      /
   2B        5B    7B    9B                  14R
  /  \                     \
1R    3R                   10R
```

Assume every empty branch has as a child a black leaf node.
The following printout was generated by this red-black tree implementation in python with the following command:

```
python3 redblack.py 12 6 1 7 4 11 15 9 5 13 8 14 3 10 2
```

```
#!/usr/bin/env python3
# vim: set ts=2 sw=2 et tw=80:

import sys

class Tree:
  def __init__(self, root):
    self.root = root
    root.parent = self

  def set_root(self, root):
    self.root = root
    root.parent = self


class Node:
  def __init__(self, k):
    self.key = k
    self.isBlack = True
    self.left = None
    self.right = None
    self.parent = None
```

```
  def set_left(self, kNode):
    if kNode is not None:
      kNode.parent = self
    self.left = kNode

  def set_right(self, kNode):
    if kNode is not None:
      kNode.parent = self
    self.right = kNode


def is_black(node):
  return node is None or node.isBlack


def insert(tree, node):
  y = None
  x = tree
  # Imperatively find place to insert node
  while x is not None:
    y = x
    if node.key < x.key:
      x = x.left
    else:
      x = x.right

  node.parent = y
  if y is None:
    tree = node
  elif node.key < y.key:
    y.left = node
  else:
    y.right = node
  node.isBlack = False
  insert_fixup(tree, node)


def sibling(node):
  if node.parent.left is node:
    return node.parent.right
  else:
    return node.parent.left


def uncle(node):
```

```
  return sibling(node.parent)


def right_rotate(x):
  p = x.parent
  t = x.left
  x.set_left(t.right)
  t.set_right(x)
  if isinstance(p, Tree):
    p.set_root(t)
  elif p.left is x:
    p.set_left(t)
  else:
    p.set_right(t)


def left_rotate(x):
  p = x.parent
  t = x.right
  x.set_right(t.left)
  t.set_left(x)
  if isinstance(p, Tree):
    p.set_root(t)
  elif p.left is x:
    p.set_left(t)
  else:
    p.set_right(t)


def insert_fixup(tree, node):
  if isinstance(node.parent, Tree): # if root
    node.isBlack = True
  elif is_black(node.parent):
    # no fixup needed
    pass
  elif not isinstance(node.parent.parent, Tree) and not is_black(uncle(node)):
    node.parent.parent.isBlack = False
    node.parent.isBlack = True
    if sibling(node.parent) is not None:
      sibling(node.parent).isBlack = True
    insert_fixup(tree, node.parent.parent)
  else:
    if node.parent.parent.left is node.parent:
      if node.parent.right is node:
        left_rotate(node.parent)
        node = node.left
```

```
      right_rotate(node.parent.parent)
    else:
      if node.parent.left is node:
        right_rotate(node.parent)
        node = node.right
      left_rotate(node.parent.parent)
    node.parent.isBlack = True
    if sibling(node) is not None:
      sibling(node).isBlack = False


if __name__ == "__main__":
  args = [x for x in sys.argv[1:]]
  T = Tree(Node(int(args[0])))
  for i in range(1, len(args)):
    print_tree(T.root)
    print("\nInsert␣" + str(args[i]) + ":")
    insert(T.root, Node(int(args[i])))
  print_tree(T.root)
```

Listing 2: Red-black tree implementation

2.2 Point B

A red-black tree of n distinct elements has an height between $\log(n)$ and $2\log(n)$ (as professor Carzaniga said in class) thanks to the red-black tree invariant. The worst-case insertion complexity is $\log(n)$ since finding the right place to insert is as complex as a regular tree (i.e. logarithmic) and the "fixup" operation is logarithmic as well (the tree traversal is logarithmic while operations in each iteration are constant). In asymptotic terms, the uneven height of leaves in the tree does not make a difference since it is a constant factor (since the maximum height is $2\log(n)$ and the minimum one is $\log(n)$.

3 Exercise 3

```
FUNCTION JOIN-INTERVALS(X)
  if X.length == 0:
    return

  for i from 1 to X.length:
    if i % 2 == 1:
      X[i] ← (X[i], 1)
```

```
  else:
    X[i] ← (X[i], -1)

sort X in place by 1st tuple element in O(n log(n))

c ← 1
n ← 0
start ← X[1][1]
for i from 2 to X.length:
  c ← c + X[i][2]
  if c == 0:
    X[2 * n + 1] ← start
    X[2 * n + 2] ← X[i][1]
    if i < X.length:
      start ← X[i+1][1]
    n ← n + 1
X.length ← 2 * n
```

Listing 3: Solution for exercise 3

The complexity of this algorithm is $O(n \log(n))$ since the sorting is in $\Theta(n \log(n))$ and the union operation afterwards is $\Theta(n)$.

4 Bonus

The number of possible trees with n nodes can be defined recursively. The number of trees with 0 or 1 node is clearly 1, since there is no freedom in arranging any remaining elements as children. For n nodes, this number can be defined as the the sum, for each $x, y \in N_0$ s.t. $x + y = n - 1$, of the number of trees with x nodes times the number of trees with y nodes. We consider only $n - 1$ nodes since one node must be the root of the tree. Using math notation, the number T_n of trees with n nodes can be expressed as:

$$T_n = 1 \qquad n = 0 \vee n = 1$$

$$T_n = \sum_{i=0}^{n-1} T_i T_{n-1-i} \qquad n \geq 2$$