CSCI 204: Data Structures & Algorithms

Revised by Xiannong Meng based on textbook author's notes

Binary Search Tree

Revised based on textbook author's notes.

Search Trees

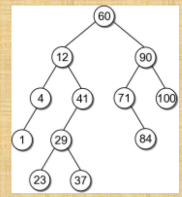
- The tree structure can be used for searching.
  - Each node contains a search **key** as part of its **data**.
  - Nodes are organized based on the relationship between the keys.
- Search trees can be used to implement various types of data structures.
  - Most common use is with the Map ADT.

Binary Search Tree (BST)

- A binary tree in which each node contains a search key and the tree is structured such that for each interior node **V**:
  - All keys less than the key in node **V** are stored in the left subtree of **V**.
  - All keys greater than the key in node **V** are stored in the right subtree of **V**.

BST Example

- Consider the example tree



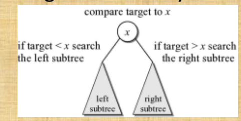
BST – ADT

bst.py

```
# We use an unique name to distinguish this version
# from others in the chapter.
class BST :
  def __init__( self ):
    self._root = None
    self._size = 0

  def __len__( self ):
    return self._size

# ...

# Storage class for the binary search tree nodes.
class _BSTNode :
  def __init__( self, key, data ):
    self.key = key
    self.data = data
    self.left = None
    self.right = None
```

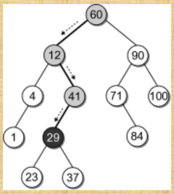
BST – Searching

- A search begins at the root node.
  - The target is compared to the key at each node.
  - The path depends on the relationship between the target and the key in the node.


compare target to x
x
if target < x search the left subtree
if target > x search the right subtree
left subtree
right subtree

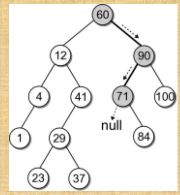
BST – Search Example

- Suppose we want to search for 29 in our BST.



BST – Search Example

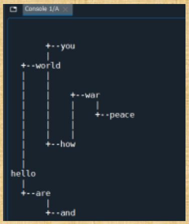
- What if the key is not in the tree? Search for key 68 in our BST.



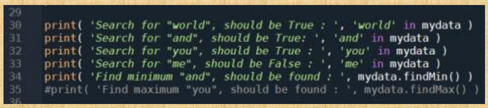
BST – Search Implementation

bst.py

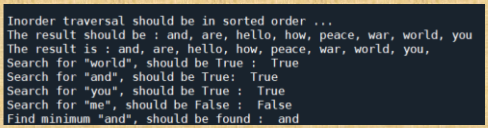
```
class BST :
# ...
  def __contains__( self, key ):
    return self._bstSearch( self._root, key ) is not None

  def valueOf( self, key ):
    node = self._bstSearch( self._root, key )
    assert node is not None, "Invalid map key."
    return node.value

  def _bstSearch( self, subtree, target ):
    if subtree is None :
      return None
    elif target < subtree.key :
      return self._bstSearch( subtree.left, target )
    elif target > subtree.key :
      return self._bstSearch( subtree.right, target )
    else :
      return subtree
```



```
          +--you
          |
     +--world
     |    |
     |    |
     |    |    +--war
     |    |    |    |
     |    |    |    +--peace
     |    |    |
     |    |    |
     |    +--how
     |
     |
  hello
     |
     +--are
          |
          +--and
```



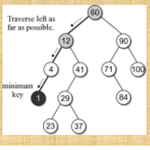
```
print( 'Search for "world", should be True : ', 'world' in mydata )
print( 'Search for "and", should be True: ', 'and' in mydata )
print( 'Search for "you", should be True : ', 'you' in mydata )
print( 'Search for "me", should be False : ', 'me' in mydata )
print( 'Find minimum "and", should be found : ', mydata.findMin() )
#print( 'Find maximum "you", should be found : ', mydata.findMax() )
```



```
Inorder traversal should be in sorted order ...
The result should be : and, are, hello, how, peace, war, world, you
The result is : and, are, hello, how, peace, war, world, you,
Search for "world", should be True :  True
Search for "and", should be True:  True
Search for "you", should be True :  True
Search for "me", should be False :  False
Find minimum "and", should be found :  and
```

BST – Min or Max Key

- Finding the minimum or maximum key within a BST is similar to the general search.
  - Where might the smallest key be located?
  - Where might the largest key be located?



BST – Min or Max Key

- The helper method below finds the node containing the minimum key.

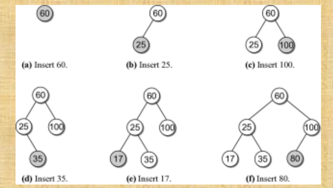
```
class BST :
# ...
  def _bstMinumum( self, subtree ):
    if subtree is None :
      return None
    elif subtree.left is None :
      return subtree
    else :
      return self._bstMinimum( subtree.left )
```

BST – Insertions

- When a BST is constructed, the keys are added one at a time. As keys are inserted
  - A new node is created for each key.
  - The node is linked into its proper position within the tree.
  - The search tree property must be maintained.

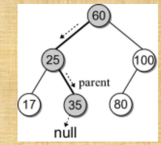
Building a BST

- Suppose we want to build a BST from the key list `60 25 100 35 17 80`



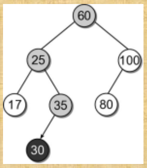
BST – Insertion

- Building a BST by hand is easy. How do we insert an entry in program code?
  - What happens if we use the search method from earlier to search for key 30?



BST – Insertion

- We can insert the new node where the search fell off the tree.



BST – Insert Implementation

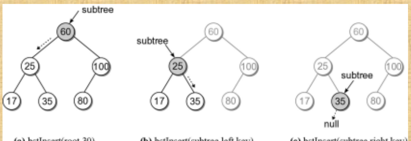
bst.py

```
class BST :
# ...
  def add( self, key, value ):
    node = self._bstSearch( key )
    if node is not None :  # just update the value
      node.value = value
      return False
    else :
      self._root = self._bstInsert( self._root, key, value )
      self._size += 1
      return True

  def _bstInsert( self, subtree, key, value ):
    if subtree is None :
      subtree = _BSTMapNode( key, value )
    elif key < subtree.key :
      subtree.left = self._bstInsert(subtree.left, key, value)
    elif key > subtree.key :
      subtree.right = self._bstInsert(subtree.right, key, value)
    return subtree
```

19

BST – Insert Steps

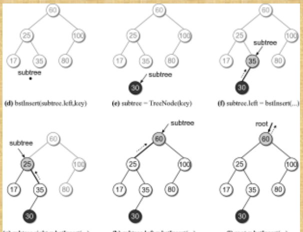
- Add 30 to our sample BST.



(a) bstInsert(root,30) **(b)** bstInsert(subtree.left,key) **(c)** bstInsert(subtree.right,key)

20

BST – Insert Steps



(d) bstInsert(subtree.left,key) **(e)** subtree = TreeNode(key) **(f)** subtree.left = bstInsert(...)

(g) subtree.right = bstInsert(...) **(h)** subtree.left = bstInsert(...) **(i)** root = bstInsert(...)

21