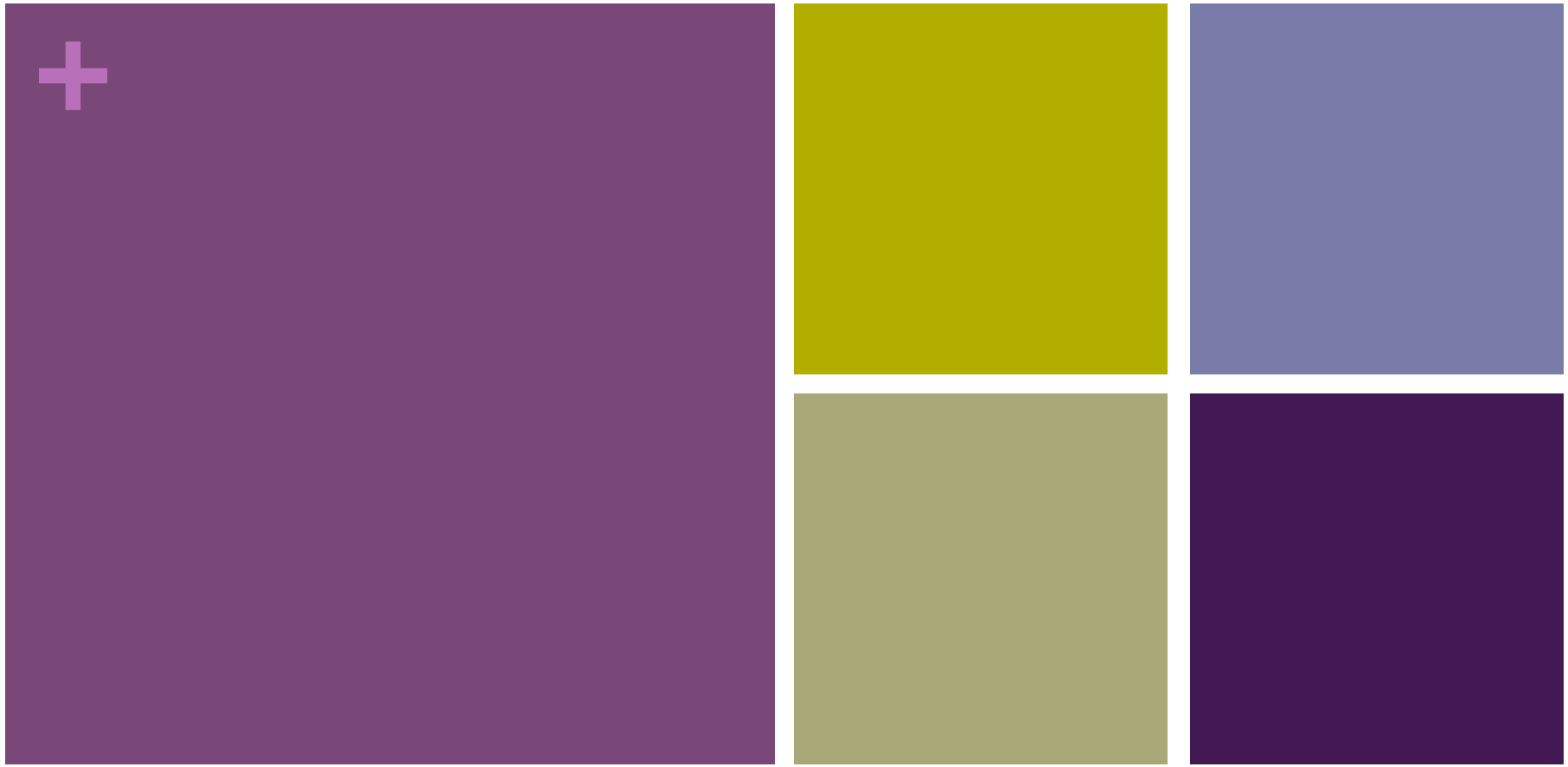


BinarySearchTree Implementation

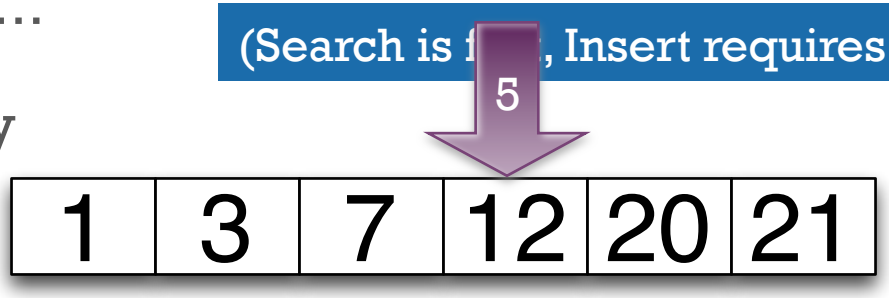
Efficient Dynamic Structure for Search?

- What is a typical data structure used for search?
  - Sorted Array
- What if we want the structure to be dynamic?
  - Will a linked list work?
- How do we make the structure efficient?

Insert Into a Sorted Structure

- Insert 5 into…
- Sorted Array

(Search is f…, Insert requires shift)



Shift 4 elements to right (4 copies)

- Sorted Double Linked List

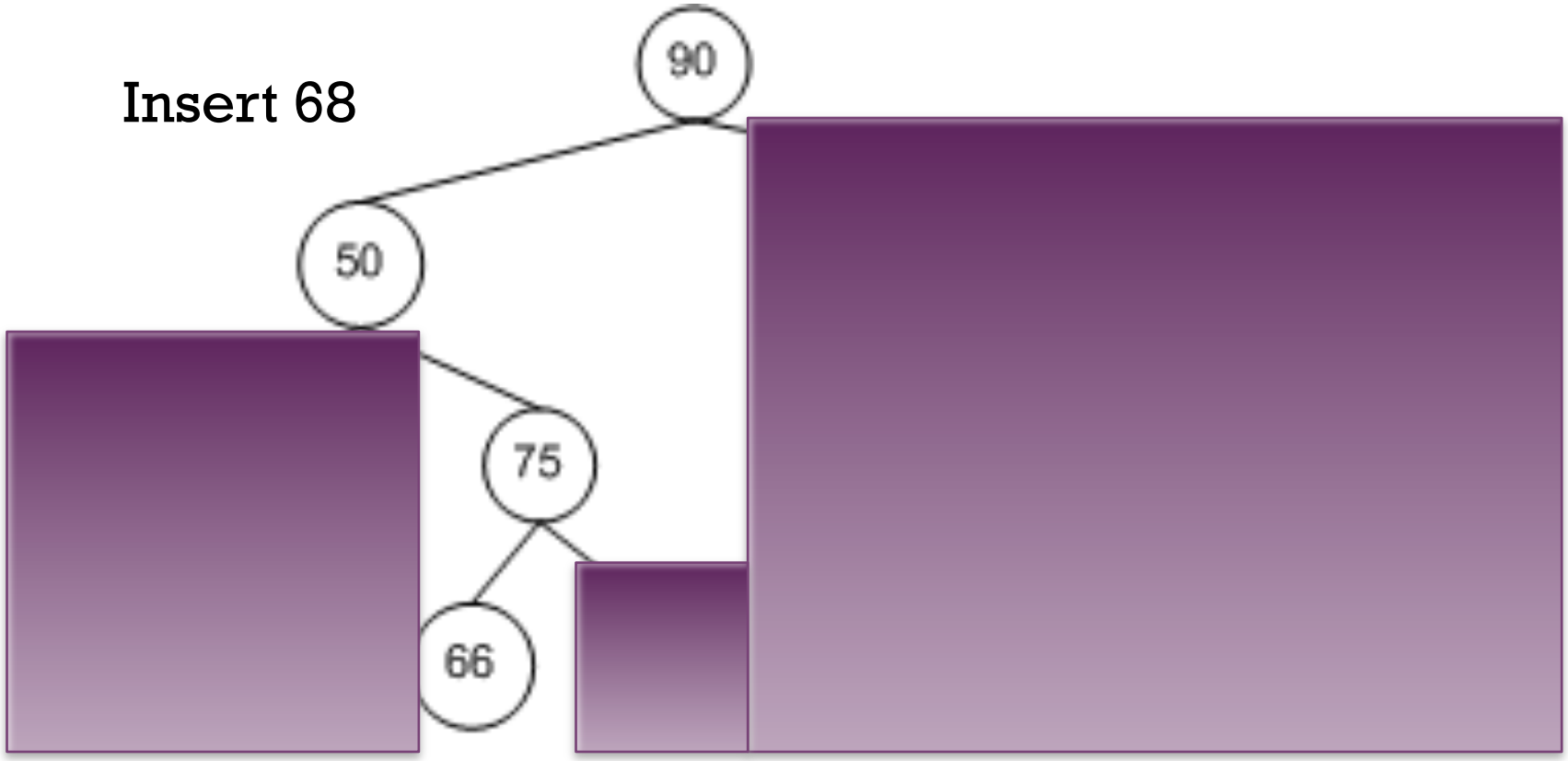
(Search is slow, insert is fast)

Can we have both fast search and fast insert?

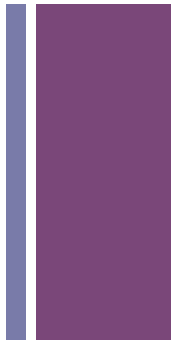
An Efficient Data Structure

- Imagine a data structure that only needs at most O(log n) comparisons to find an answer and O(log n) actions to insert or remove a value.
- What would it look like?
  - For search: Each comparison would need to eliminate half of the remaining options.
  - For insertion/removal: The location of the value would need to be log n steps from the initial starting place or cause log n items to shift.

Will a Binary Search Tree work?



Implementation



- add
- find
- min
- max
- successor
- predecessor
- remove

Add e

- If tree is empty
  - make a root node with e as its data
- else
  - add(e, root);

Add e, node

- if e < node.data
  - if (node.left == null)
    - make a left node with e as its data
  - else
    - add(e, node.left)
- else if e > node.data
  - if (node.right == null)
    - …
- else
  - return false // can't add equal items.

Find e

- If tree is empty
  - return null
- else
  - return find(e, root)
- find(e, node)
  - if e equals node.data
    - return node data
  - else if e < node.data
    - return find(e, node.left)
  - else
    - return find(e, node.right)

max()

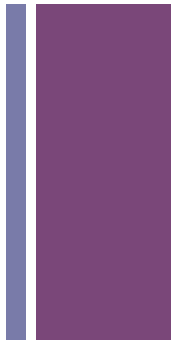
[note: min is similar using node.left instead of node.right]

- If tree is empty
  - return null
- else
  - return max(root)
- max(node)
  - if node.right == null
    - return node data
  - else
    - return max(node.rignt)

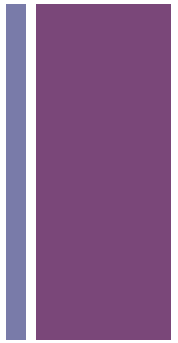
predecessor(node)

[note: successor is similar using node.right instead of node.left]

- return max(node.left)



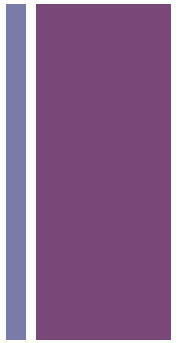
Traversal algorithms



- PreOrder(treeNode)
  - if TreeNode is empty
    - done
  - else
    - "visit" treeNode
    - PreOrder(treeNode.left)
    - PreOrder(treeNode.right)

- Simlarly for inOrder and postOrder
- Another example on board.

Exercise



- Get into groups
- do part 1, steps 1 to 3 from Assignment 6 in your group.