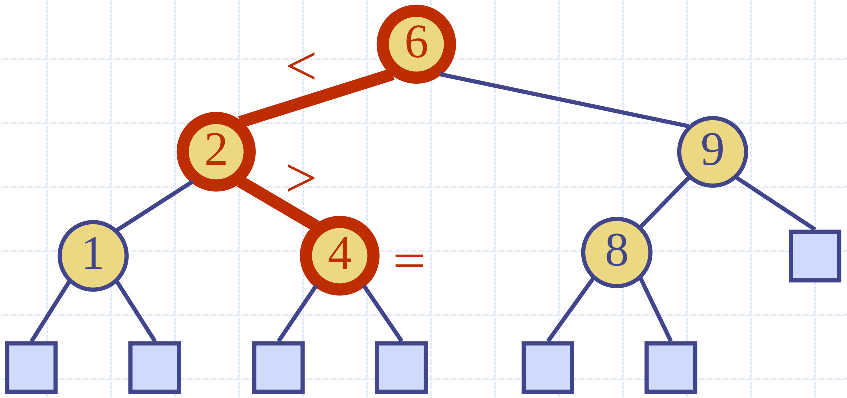
Modified version of the presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Binary Search Trees



What does it mean to “Search”?

- Find and retrieve information within a data structure
- Or figure out that it does not exists

- What is the index of a given value within an array?
- What is the value of a given key in a table?
- Guess the number I am thinking of

- Find a way to get to Beşiktaş from Koç University
  - In this case, you search for the path, will come up later
  - Yes, I am a BJK supporter

Search Examples

- Search an array for a value:
  - Pseudocode on the board!
- Guess the number I am thinking of
  - Pseudocode on the board!
- Complexities?
  - O(n) vs O(log n)
- Why log n?

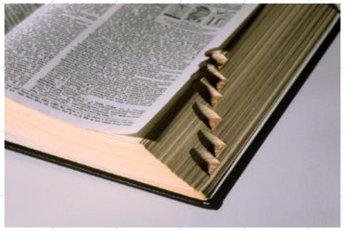
Ordered Sequence

- Ordered: Arranged in a “meaningful” way
  - Ascending order, by age, by height, by education level…
- Total Order: E.g. $\leq$ on $\mathbb{N}$
  - An order defined for all pairs of items of a set
  - A binary relation that is *reflexive*, *transitive*, *antisymmetric* and *total* defined on a set
  - Definitions on the board!
  - Counter example: Pairs on a 2D grid
- Allows us to efficient search, range search, nearest neighbor search etc.

Sidestep: Maps

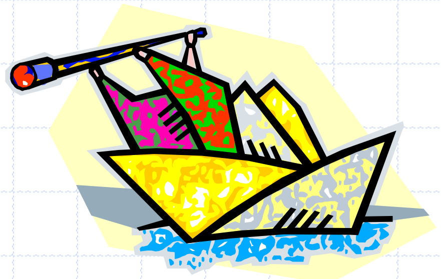
- A map is an ADT that is a collection of (*key*, *value*) pairs
  - Each key appears at most once
  - Also called dictionary, associative array, symbol table
- Operations:
  - Add (insert)
  - Remove (delete)
  - Modify
  - Lookup
- We will get back to these

Ordered Maps

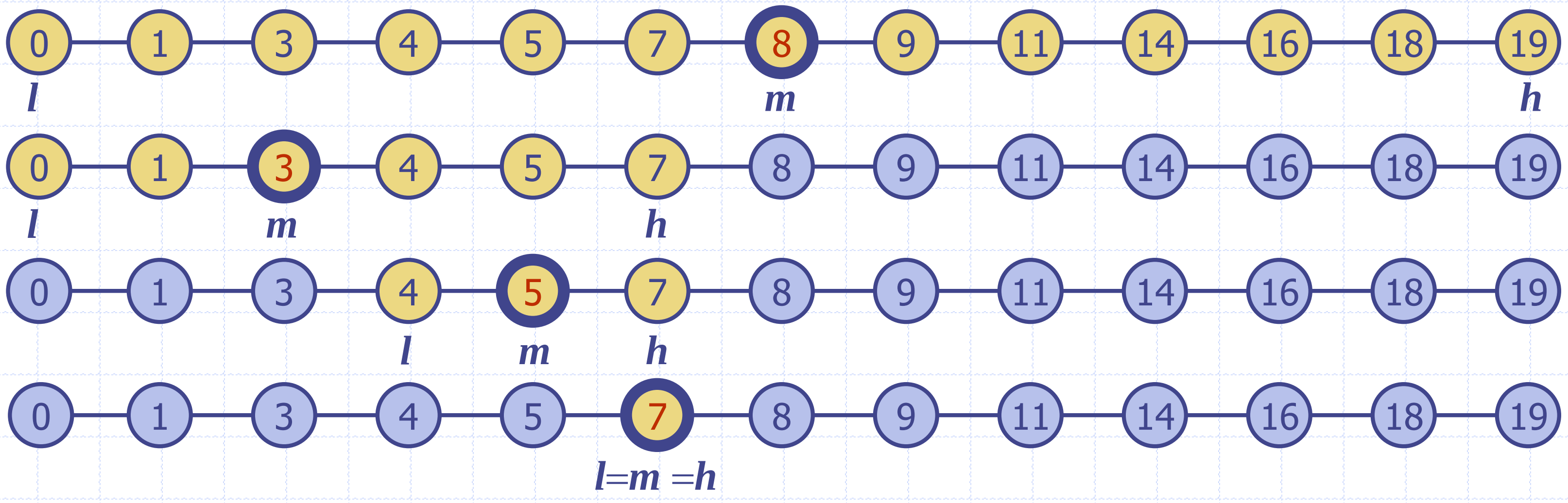


- Keys are assumed to come from a total order.
- Items are stored in order by their keys
- This allows us to support nearest neighbor queries:
    - Item with largest key less than or equal to k: ceiling
    - Item with smallest key greater than or equal to k: floor

Binary Search



- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children's game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)

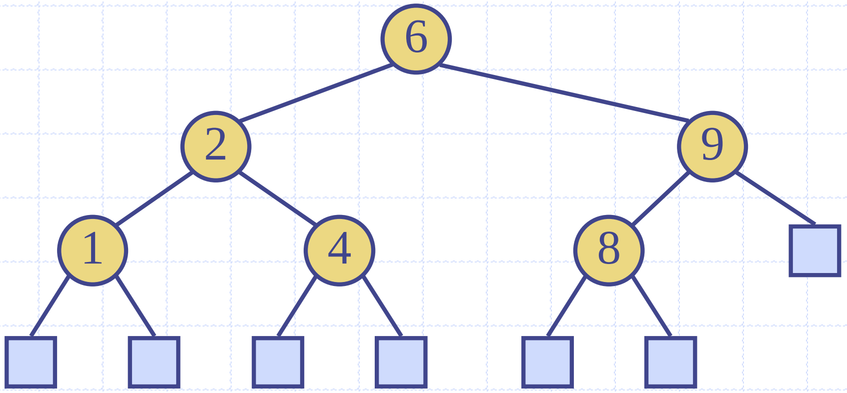


Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array-based sequence, sorted by key
  - We use an external **comparator** for the keys
- Performance:
  - Searches take $O(\log n)$ time, using binary search
  - Inserting a new item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to make room for the new item
  - Removing an item takes $O(n)$ time, since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The lookup table is effective only for ordered maps of small size or for maps on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Trees

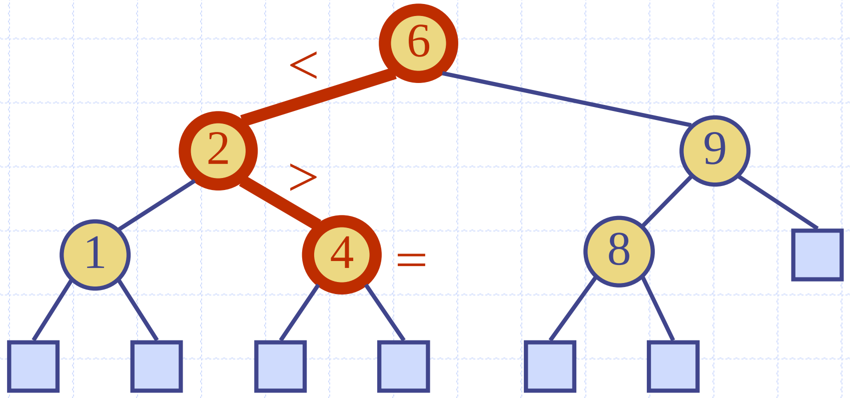
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
- An in-order traversal of a binary search trees visits the keys in increasing order



Search

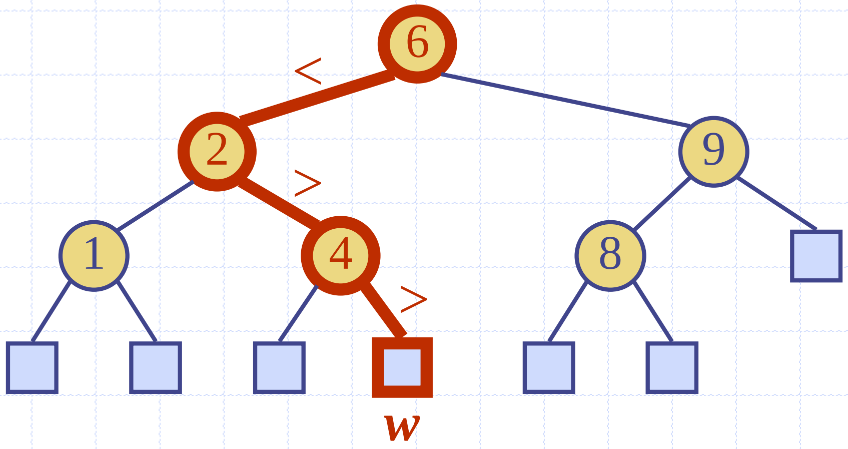
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found
- Example: get(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar

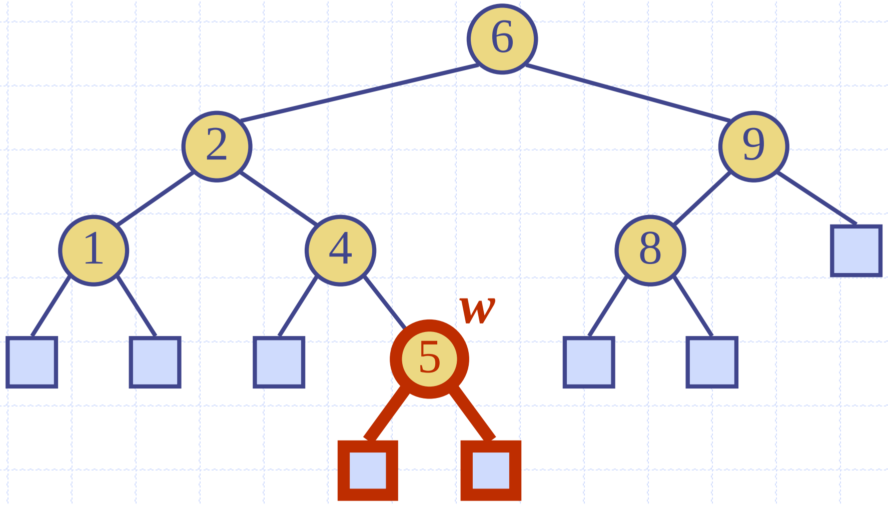
```
Algorithm TreeSearch(k, v)
    if isExternal (v)
        return v //Not found, return empty leaf
    if $k < key(v)$
        return TreeSearch(k, left(v))
    else if $k = key(v)$
        return v
    else  $k > key(v)$
        return TreeSearch(k, right(v))
```



Insertion

- To perform operation put(***k***, **o**), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let ***w*** be the leaf reached by the search
- We insert ***k*** at node ***w*** and expand ***w*** into an internal node
- Example: insert 5





Insertion

- Find where it belongs and insert it!
- Sticking with the book:

```
Algorithm TreeInsert(k, o)  //aka put
    w = TreeSearch(root, k)  //Returns the empty leaf if not found
    if isExternal (w)
        //Change the value of the leaf and adds 2 empty children:
        expandExternal(w, (k, o))
    else
        //Update the value if the location if not empty, could also throw an error
        update(w, o)
```

- Note that insertions always happen at the leaf nodes. This may “unbalance” the tree.
- We are going to see more advanced structures that deal with this problem

Deletion

- There are 3 cases to handle:
  1. A node with only leaf children (i.e. the child nodes do not store any values):

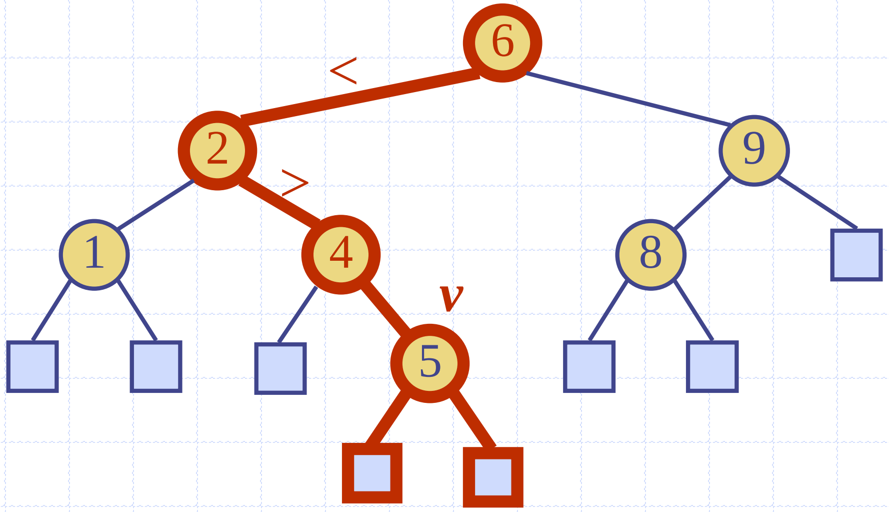
     Delete the node from the tree
  2. A node with only one child node that stores a value:

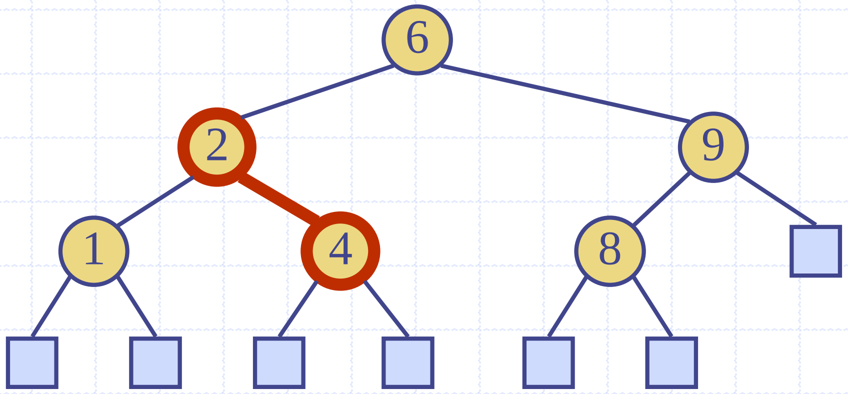
     Delete the node and replace it with its child
  3. A node with two child nodes that store a value

     Find its in-order successor or predecessor, call this w. Copy the value of w to the node to be deleted. Then delete w .

Deletion: Case 1

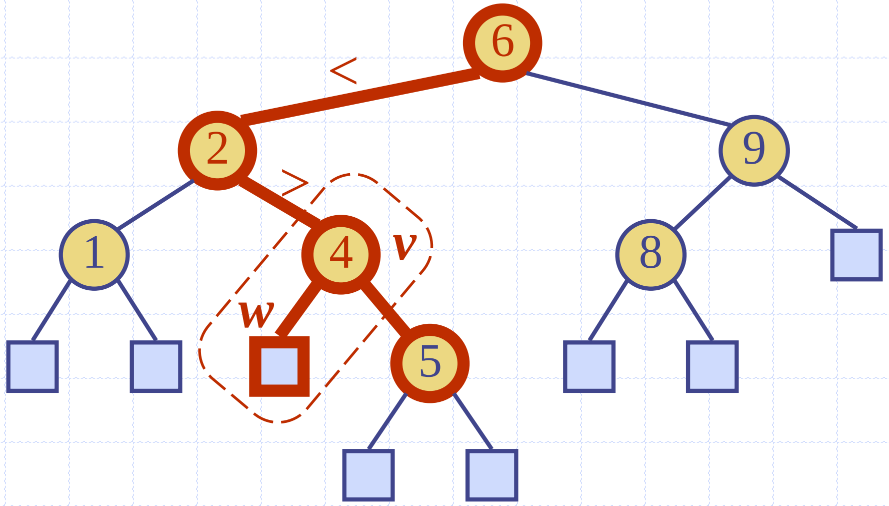
- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has only has leaf children, remove it and replace it with an empty leaf node
- Example: remove 5

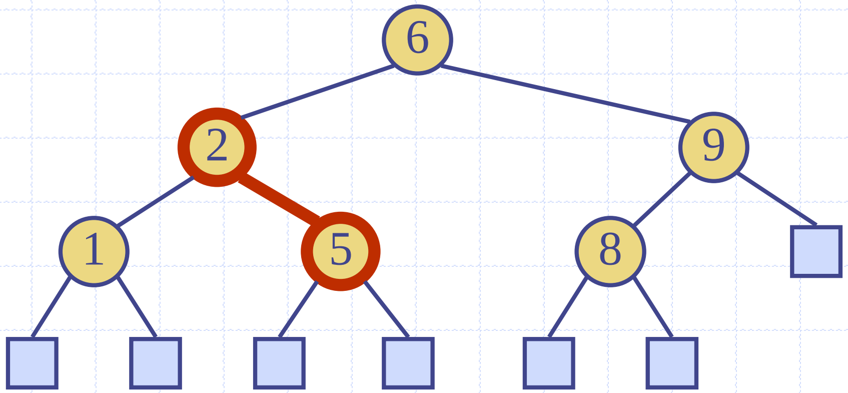




Deletion: Case 2

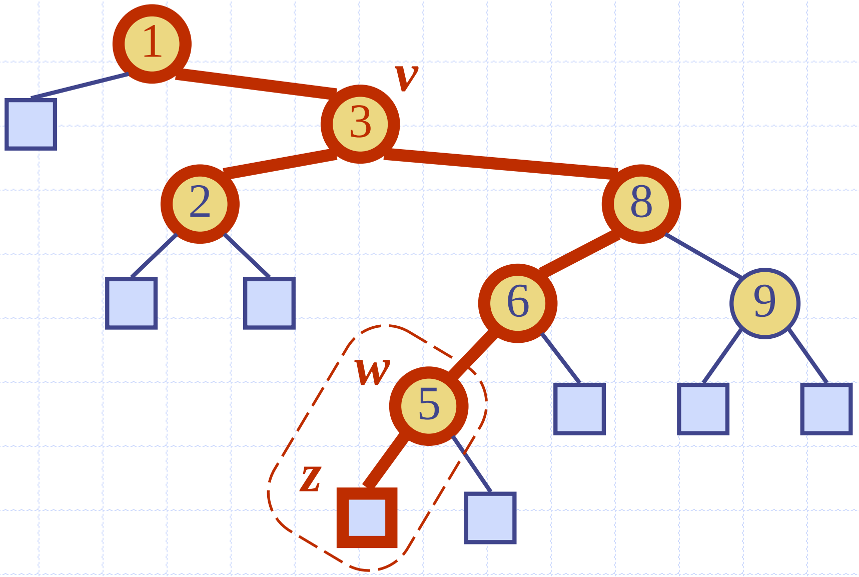
- To perform operation remove(***k***), we search for key ***k***
- Assume key ***k*** is in the tree, and let ***v*** be the node storing ***k***
- If node ***v*** has only one leaf child ***w***, we remove ***v*** and ***w*** from the tree and replace ***v*** with its internal child
- Example: remove 4

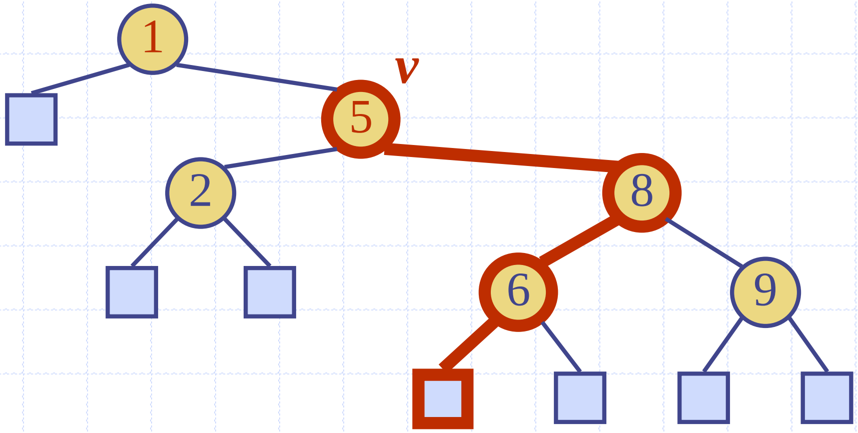




Deletion: Case 3

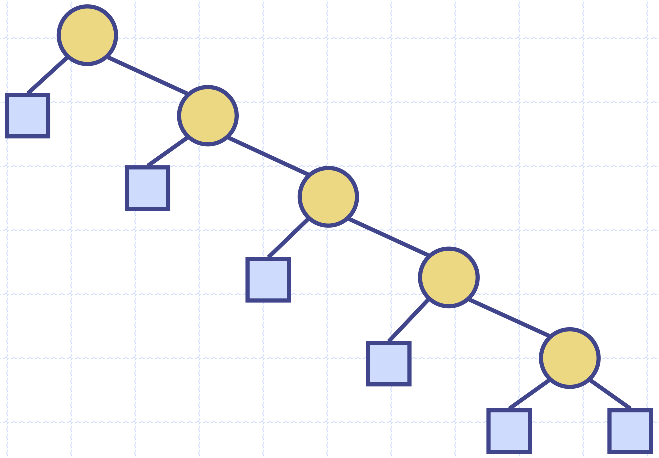
- We consider the case where the key ***k*** to be removed is stored at a node ***v*** whose children are both internal
  - we find the internal node ***w*** that follows ***v*** in an in-order traversal
  - we copy ***w*** into node ***v***
  - we remove node ***w*** and its left child ***z*** which must be a leaf (or
  - Example: remove 3
- Deletion could also be done by picking the preceding node

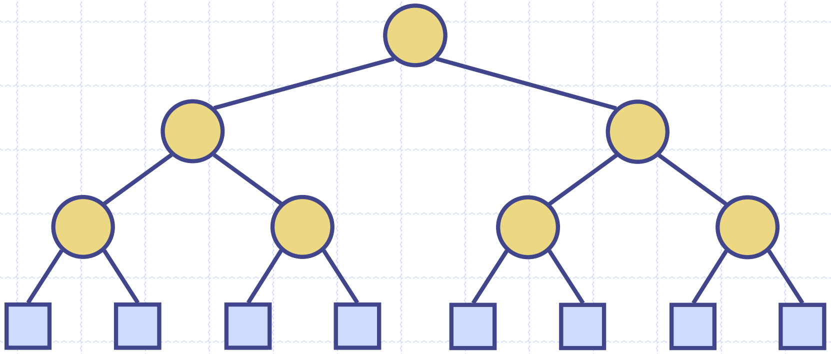




Performance

- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods get (search), put (insert) and remove (delete) take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case





Some Notes

- In implementing these trees, you do not have to use the position ADT as presented in the book. I encourage you to look online for alternatives.
- In the book, the leaves of a binary tree are always empty. However, you could just simply have them not exist. Having empty nodes as leaves help with keeping the position after a search operation, e.g., we used it in the insert operation.
- An application might call for a more specific implementation