Symbol Tables API

- key-value pair abstraction
- insert a value with specified key
- given a key search for the corresponding value

* Associative array abstraction - associate one value with each key

class ST

```
void put(Key key, Value val)
Value get(Key key)
void delete(Key key)
boolean contains(Key key)
boolean isEmpty()
int size()
Iterable<Key> keys()
```

Conventions

- no null values allowed
- get returns null if key is not present
- put() overwrites old value with new value

Key Assumptions

- assume keys are comparable / use compareTo
- assume keys are any generic type / use equals to test equality
- use hashCode() to scramble key
- use immutable types for symbol table keys

Elementary implementations

- unordered linked-list of key-value pairs
  - search : scan through all keys
  - insert : scan through all keys, no match → add it to front

Binary Search in an ordered array

- data structure: maintains an ordered array of key-value pairs
- rank helper function - how many keys < k
- to insert need to shift all greater keys over

Ordered Operations

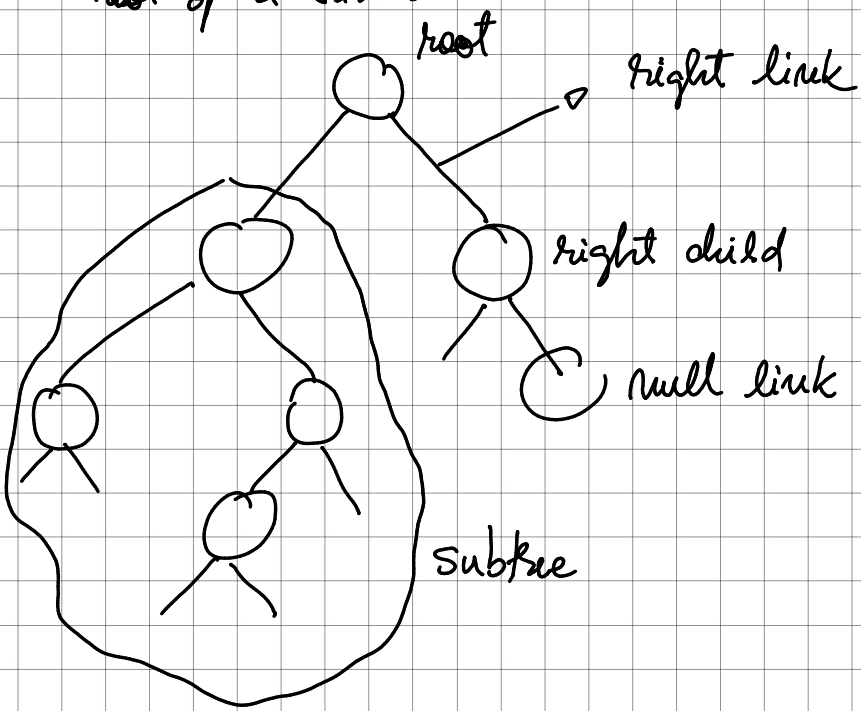
- min, max
- floor, ceiling
- rank
- select - key of rank k
- deleteMin, deleteMax
- int size (low, high)
- keys (low, high)

Binary Search Trees

BST = binary tree in symmetric order

binary tree is either:

- empty
- two disjoint binary trees (left and right)
- links can be null
- every node in a tree can be the root of a subtree



Symmetric Order

- each node has a key
- every node's key is:
  - larger than all keys in its left subtree
  - smaller than all keys in its right subtree

Java

Node
- key
- value
- left subtree (smaller keys)
- right subtree (larger keys)

BST = reference to a root Node

SEARCH: if less go left
if greater go right
if equal => hit

INSERT — if less go left
if greator go right
if null insert

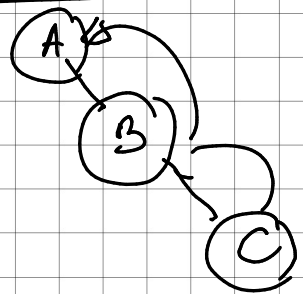
- search : cost

no. of compares is equal to 1+ depth of node

- put : associate value with key
  - search for key
    - key in tree → reset val
    - key not in tree → add
  - cost : no. of compares is equal to 1 + depth of node

root · null

put (A)
put (B)
put (C)

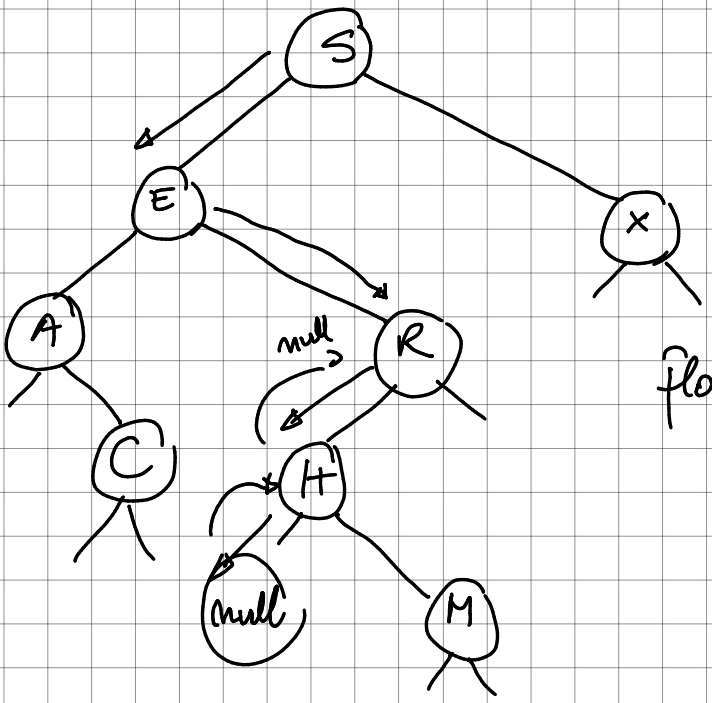


- tree shape depends on order of insertion
- if you insert N distinct keys into a BST in random order $\Rightarrow$ expected no. of compares for a search is $\boxed{\sim 2 \ln N}$
- worst case height is N

Operations

- minimum = smallest key in table — left link
- max = largest key in table — right link until null
- floor and ceiling
  - floor = Largest key $\leq$ to a given key
  - ceiling = smallest key $\geq$ to a given key

floor
1) k equals key at the root
2) k is less than key at root
3) k is greater than key at root



floor (6)

- rank, select, size - we store the number of nodes in the subtree rooted at that node

In order traversal

- traverse left subtree
- enqueue key
- traverse right subtree

Deletion in BST

- **lazy approach**
  - set the value to null
  - leave key in tree to guide searches (but don't consider it equal in search)
- **deleting the minimum**
  - go left until finding a node with a null left link
  - replace the node with right link
  - update subtree counts
- **Hibbard deletion**
  - to delete a node with key k: search for node t containing key k

  case 0: delete t by setting parent link to null (0 children)

  case 1: delete t by replacing parent link (1 child)

- case 2 - [2 children]
  - find successor x of t
  - delete the min in t's right subtree
  - put x in t's spot