7.2 dictionaries and BSTs

Thursday, October 20, 2022

Today:
- key-based applications
- dictionary ADT to store keys and values
- possible implementations
- removing from a BST

MOTIVATING APPLICATION

- Amazon has ~ 75 million Prime subscribers
- Instagram has > 1 billion users
- Tiktok has > 1 billion users
- Internet Archive has 625 billion pages

Each entry in the behind-the-scenes data storage has a unique KEY that identifies it (username, email address, URL, ...).

The application needs to find your individual data quickly when you log in to request it.

KEY $\xrightarrow{\text{maps to}}$ VALUE

ex: username data

We assume that every Key is unique.

For key-value data like this, we'll use the DICTIONARY ADT

templated with k = key type and v = value type

```
void insert(k, v)
v get(k)
v remove (k)
void update (k, v)
bool contains(k)
bool isEmpty()
int getSize()
vector<k> getKeys()
vector<pair<k,v>> getItems()
```

Note: "vector" is a built-in ArrayList in C++.

Dictionaries are behind real-world applications.
We need them to be efficient.
Which operations should be especially fast? get, update, insert, contains, remove

We need them to be efficient.

Which operations should be especially fast? get, update, insert, contains, remove
getSize, isEmpty, getKeys

If we use	ArrayList	sorted ArrayList	BST (assume balanced)
insert	$O(1)$ amortized	$O(n)$	$O(\log_2(n))$ idea: find where it goes, add it there
remove	$O(n)$	$O(n)$	??
update get contains	$O(n)$	binary search $O(\log_2(n))$	$O(\log_2(n))$

Dictionary ADT

- maps keys → values
- assumes keys are unique
- dictionaries are behind-the-scenes of many applications
- operations insert, get, update, remove, contains must be fast
- previous data structures like List will be too slow

Plan: implement dictionary ADT as BST

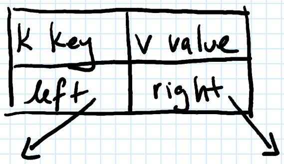
- operations on BST are $O(\text{height})$
- as long as BST is "balanced" (fully packed) the height of a size-n BST is $O(\log_2 n)$
- ... and $O(\log_2 n)$ is really fast! (recall $\log_2(1 \text{ billion}) \approx 30$)

BST : binary search tree

- binary tree (every node has left & right subtrees which can possibly be empty)
- binary search property invariant is true at every node
  - All keys in the left subtree of a node must be less than the key at that node. All keys in the right subtree of a node must be greater than the key at that node.

implementation details:

Linked BST Node



} contains 4 data members

```
private : // data
    K key
    V value
    LinkedBSTNode< K,V> *left
    LinkedBSTNode< K,V>* right
public: // methods
    getKey, setKey
    getValue, setValue
    getLeft, setLeft
    getRight, setRight
```

Linked BST

```
private: //data
    LinkedBSTNode<K,V>* root
    int size
```

Note: you can reach the entire tree by starting at the root.

Q: Suppose we had a BST that mapped Swarthmore ID#s to names.

What type would K be? int
What type would V be? string

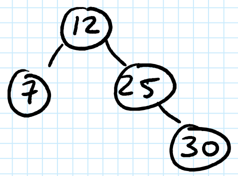
Q: Are these keys unique? yes

Q: Could we have a BST mapping names to ID#s? No. Names are not unique.

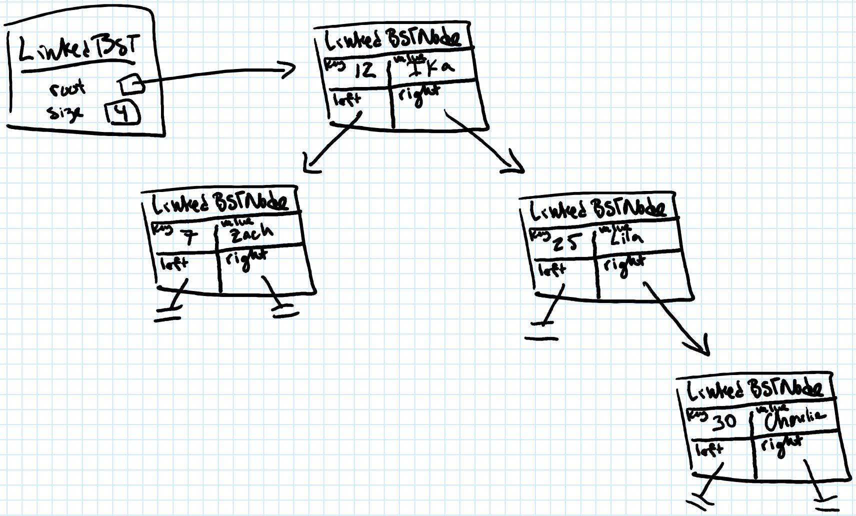
example:

12 → "Ika"
7 → "Zach"
25 → "Lila"
30 → "Charlie"

Simplified diagram of BST:



full detail memory diagram



IMPLEMENTING BST METHODS

Recall that BSTs are recursive: a BST is {
empty, or
a node with left and right subtrees satisfying the BST property

... so the best way to implement BST methods will use recursion.

Often with recursion we'll have one public method to start the recursion and a private helper method to do the work.

implementing the "get" method:

```
V get(K key)
```

This method searches through the BST for the given key and returns the associated value. If the key is not in the BST, this should throw an error.

pseudocode:

```
V get(K key)  // public method
    return findInSubtree(root, Key)

V findInSubtree(current, key)  // private helper
  // base cases
  if current is null
      throw error "key not found"
  if current's key is equal to key
      return current's value
  //recursive cases
  if key < current's key
      return findInSubtree(current's left, key)
  else // key > current's key
      return findInSubtree(current's right, key)
```

★ Remember: every case needs a return.

implementing the "insert" method:

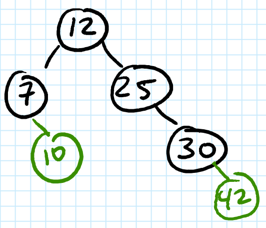
void insert(K key, V value)

Q: What should it do if the key is already in the BST?

Throw an error. Every key must be unique.

example:

insert(10, ...)



insert(25, ...) throw error!

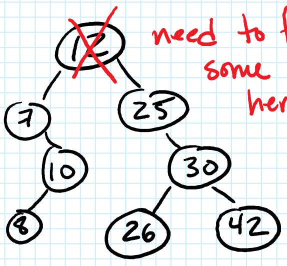
insert (42, ...)

pseudocode for insert

```
void insert( K key, V value) // public method
    // insert changes the structure of the tree, so we update root
    root = insertInSubtree(root, key, value)
    increment size

node * insertInSubtree(current, key, value) // private helper
    // base case
    if current is null
        create a new node to store key and value
        return a ptr to that node
    if key is equal to current key
        throw error " no duplicate keys allowed! "
    // recursive cases
    if key < current's key
        current's left = insertInSubtree(current's left, key, value)
        return current

    if key > current's key
        current's right = insertInSubtree(current's right, key, value)
        return current
```

the "remove" method

remove (12)



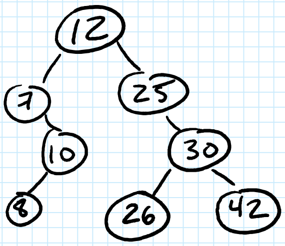
need to find some key to move here to fill the gap

IDEA:
- find predecessor of 12 which is 10
- replace (key, value) at node with (key, value) of predecessor
- remove predecessor

How do we find the predecessor key of a node in a BST?

- the predecessor must be the largest key in the node's left subtree
- this is the rightmost node in the left subtree

How do we find the successor key of a node in a BST?



predecessor of 12?

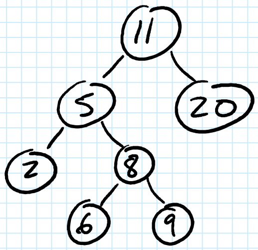
successor of 12?

pseudocode for remove

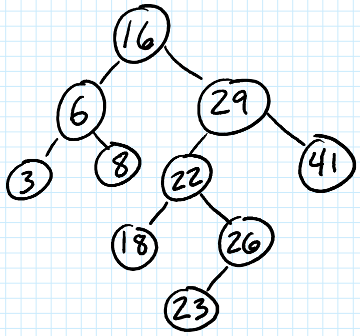
```
void remove(K key) // public method
    // remove changes tree structure, so update root
    root = removeFromSubtree(root, key)
    decrement size

node* removeFromSubtree(current, key) //private helper
    if current is null
        throw exception "key not found"
    if key < current's key
        current's left = removeFromSubtree(current's left, key)
```

```
    return current
else if key > current's key
    current's right = removeFromSubtree(current's right, key)
    return current
else // this must mean we found the key in the BST at current node
```

examples:



remove(5)



remove(29)