Map Implementation Summary

set

assuming $c_1 \leq \alpha \leq c_2$ (for large n) and hash fxn distributes keys evenly

	unsorted list (array/linked)	sorted array	sorted linked list	hash table	
contains/get (contains)	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$ expected, $O(n)$ worst case	$O(\log n)$
put (add)	$O(n)$	$O(\log n)$ if key present, $O(n)$ worst	$O(n)$	$O(1)$ expected, $O(n)$ worst case	$O(\log n)$
remove	$O(n)$	$O(n)$	$O(n)$	$O(1)$ expected, $O(n)$ worst case	$O(\log n)$
iterate through all (key,value)s	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
sort	$O(n \log n)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(n)$

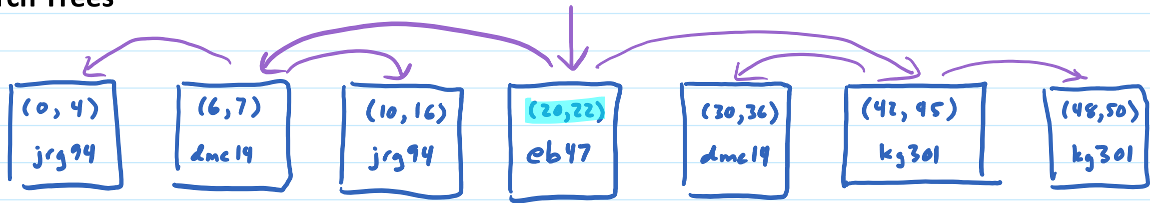
keys : disjoint intervals of integers
values : whatever

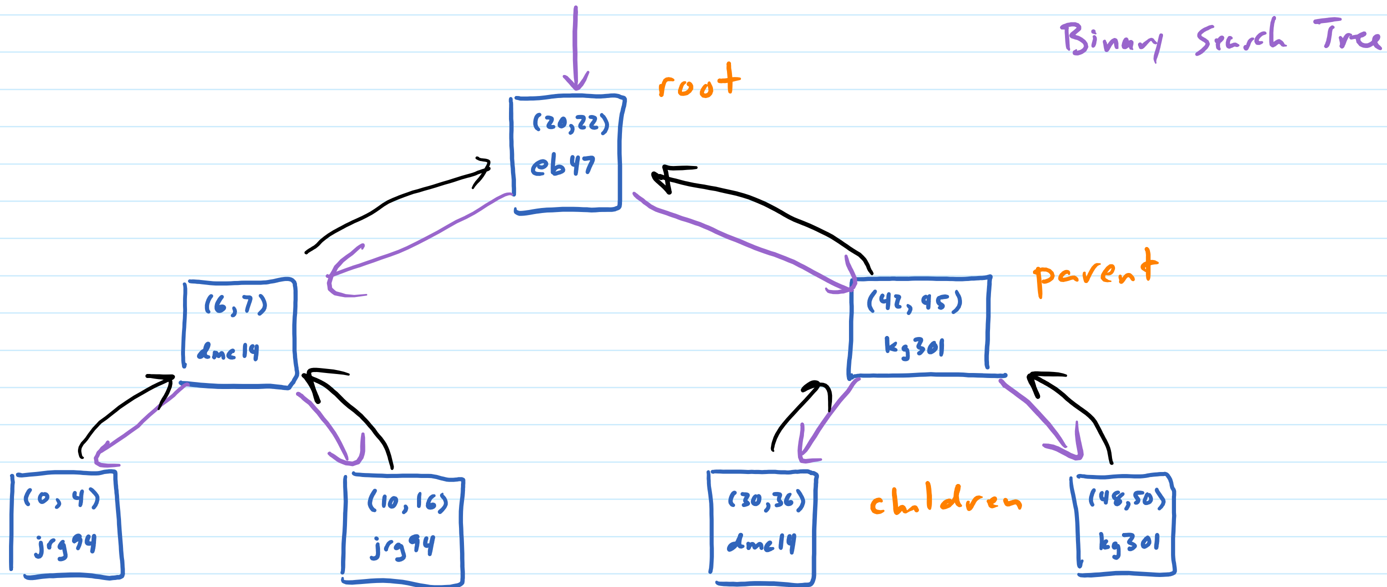
(10, 20)	jrg54
(30, 60)	kg301
(61, 100)	jrg54
(105, 120)	kg301

get : given int x, find value associated w/ interval containing x

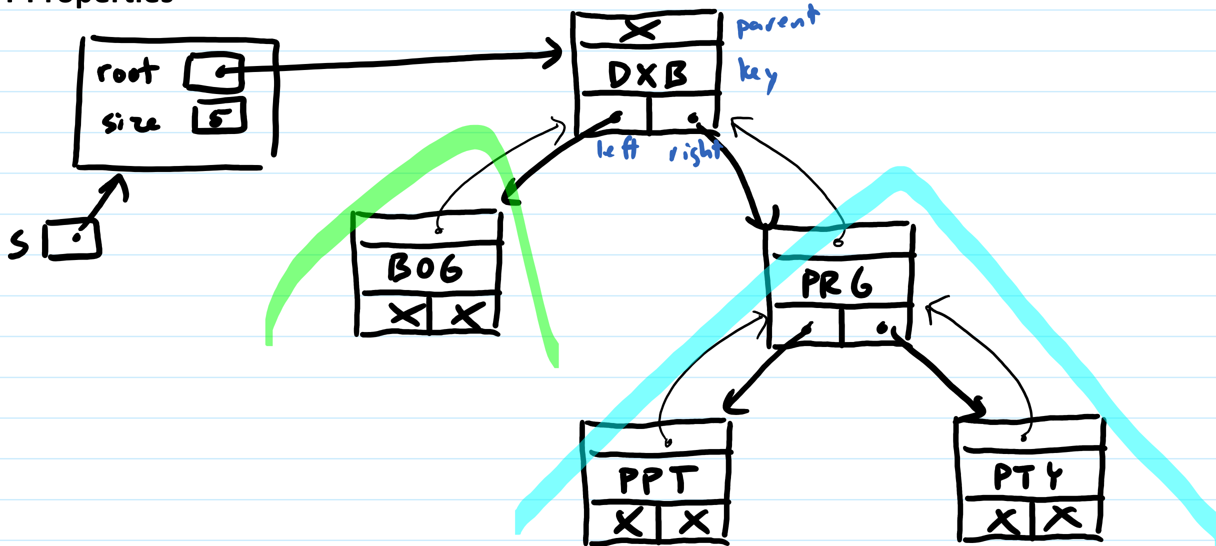
get(36) = kg301

Binary Search Trees





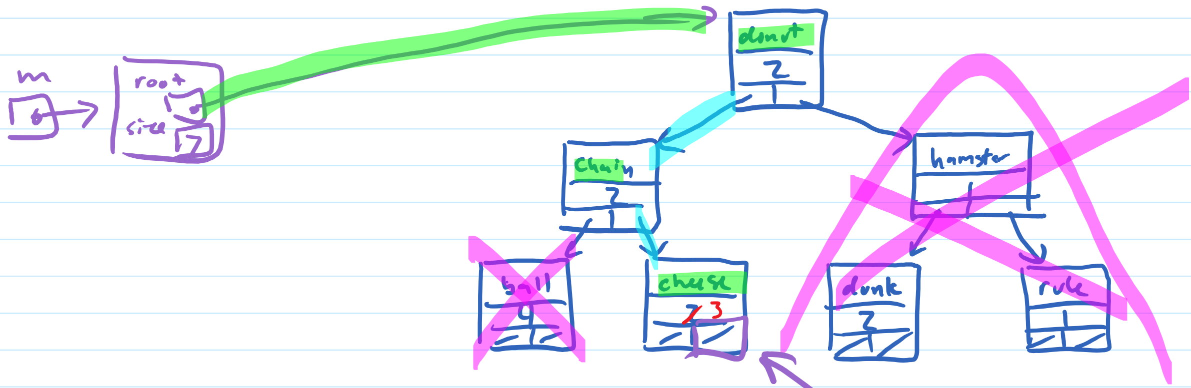
BST Properties



Properties: for every node n, values of keys in left subtree all < key in n

for every n, values in right subtree > value in n

Searching in a BST




```
bool smap_contains_key(smap *m, const char *key)
{
   smap_node *curr = m->root;
   while (curr != NULL && strcmp(key, curr->key) != 0)
   {
      if (strcmp(key, curr->key) < 0)
      {
         curr = curr->left;
      }
      else
      {
         curr = curr->right;
      }
   }
   return (curr != NULL);
}
```

Adding to a BST

```
bool smap_put(smap *m, const char *key, int value)
{
   smap_node *curr = m->root;
   smap_node *prev = NULL;
   while (curr != NULL && strcmp(key, curr->key) != 0)
   {
     prev = curr;
     if (strcmp(key, curr->key) < 0)
     {
       curr = curr->left;
     }
     else
     {
       curr = curr->right;
     }
   }
   if (curr == NULL)
   {
     // make a new node
     // link new node with existing nodes
     if (strcmp(key, prev->key) > 0)
     {
       prev->right = new_node;
       new_node->parent = prev;
     }
   }
}
```