Parallel And Sequential Data Structures and Algorithms

Extensions of Binary Search Trees

Learning Objectives

- Understand how Augmented BSTs make use of combining functions to support range queries
- Motivate and implement rank-based operations on BSTs

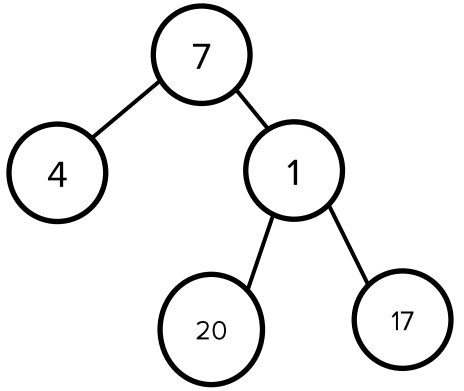
BSTs/Treaps Summary

- BSTs store key-value pairs
- Maintain a sorted order of keys
- Support operations: insert, delete, find, split/join
- Balanced BST operations cost: $O(\log n)$

Augmented BSTs

Motivation

- Range queries – We want to answer a query on a specified range efficiently
  - E.g. max value over a range of an array, sum of values over a range
- Idea: Add extra information to each BST node related to the query
  - `reduced`: combined/reduced value over subtree of node

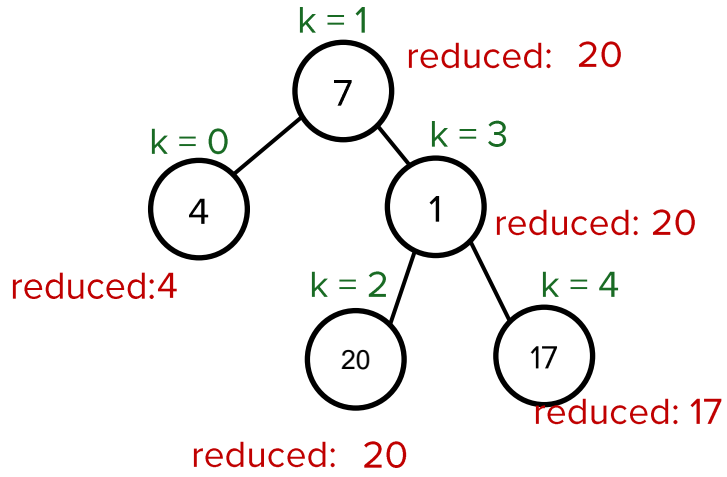


Example Queries: What is the max value in the example BST? What about max value in its right subtree?

```
type AugBST<K : Ordered, V> =
    Empty
  | Node { left: AugBST,
           key: K,
           value: V,
           reduced: V,
           size: int,
           right: AugBST }
```

Combining Function

- Define an associative "combining function"
  `combine: (V,V) -> V`
  - Similar to scan/reduce functions
  - Associative
  - Has **identity**
- Example: If `C` is `max`:
  - `V: int`
  - **Identity** `= -∞`



Using max as combining function …

Maintaining reduced value (rvalue)

```
fun makeNode(L:AugBST<K,V>,k:K,v:V,R:AugBST<K,V>)->AugBST<K,V>:
  return Node(L, k, v,
        combine(combine(reduce_val(L),v),reduce_val(R)),
        size(L)+1+size(R),
                        R)

fun reduce_val(T: AugBST<K,V>) -> V: (* returns reduced *)
  match T with:
    case Empty: return identity
    case Node(_,_,_,reduced,_,_): return reduced

fun size(T: AugBST<K,V>) -> int:
  match T with:
    case Empty: return 0
    case Node(_,_,_,_,s,_): return s
```

- Symmetry in how we compute `reduced` and `size` at a new node!

Invariant: At any node, **reduced** represents the result of applying **combine** to all the values in the node's subtree in-order

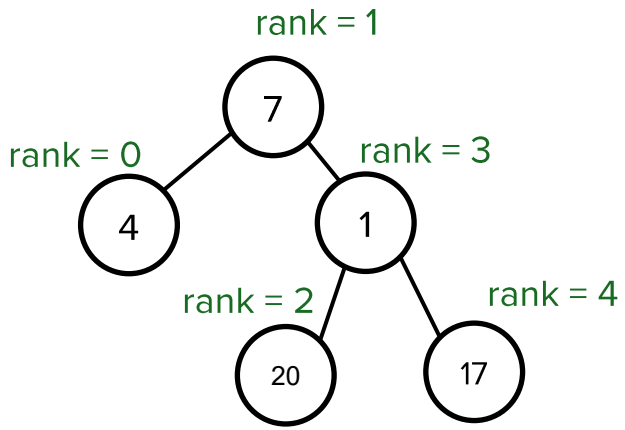
This is the only place where **combine** is used.

Rank-based Operations

Motivation

🚨 Every item in BST has a **rank** based on its position (left to right order) in the tree. Sometimes it is useful to use this method (instead of the keys) to address items. How can we efficiently access the item of rank i in the tree?

- Solution: Make use of the size fields
  - rank `i` in `[0, |T|-1]`
  - Requires new `split`, `join` operations using rank instead of key
    - Calculate rank using node's size
  - Can still access `ith` element (like sequences!)
  - Without giving up efficient BST operations
  - Still persistent data structures!!

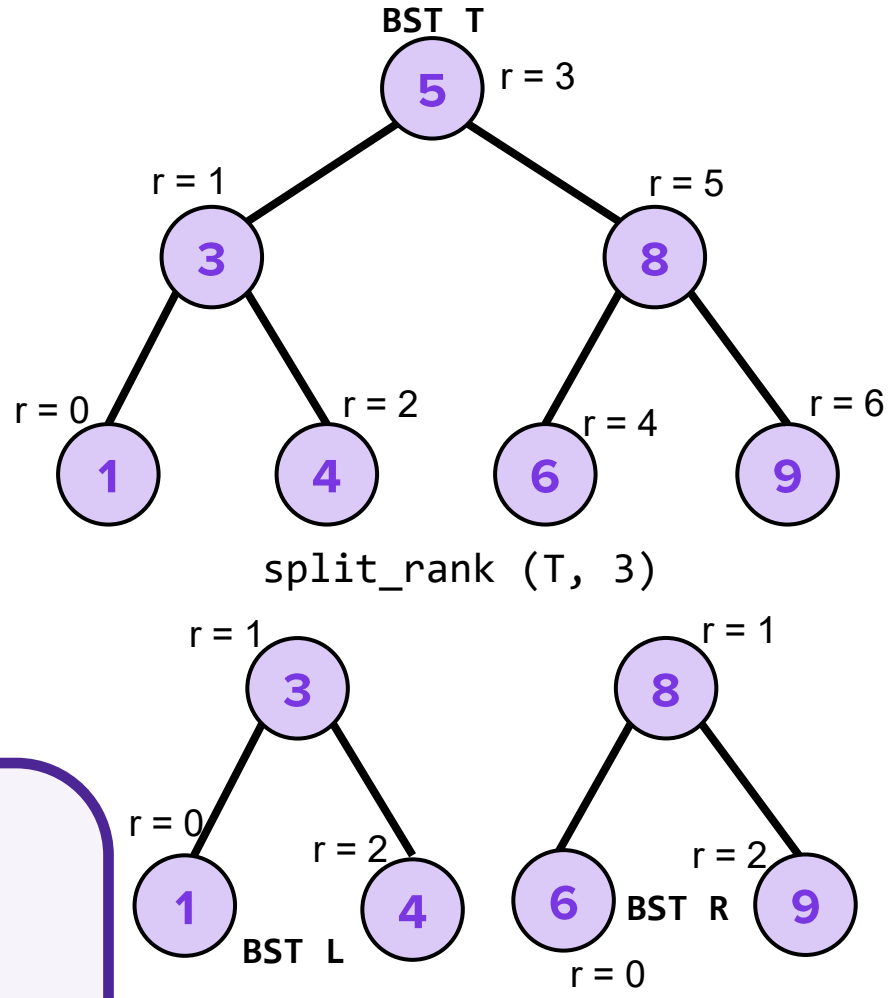


Each node still has a key and value! Just not depicted in this diagram.

split_rank Implementation

- `split_rank (T, i)` returns `(L, k, v, R)`
  - L: BST representing first `i` elements of T
  - `(k, v)` is the `ith` element of T
  - R: BST of remaining elements

```
fun split_rank(T: BST<K,V>, i: int) -> (BST, k, V, BST):
  match T with:
    case Node(L,k,v,_,_,R):
      if i = size(L):
        return(L, k, v, R)
      else if i < size(L):
        (L', k', v', R') = split_rank(L, i)
        return (L', k', v', rebalance(makeNode(R', k, v, R)))
      else:
        (L', k', v', R') = split_rank(R, i-size(L)-1)
        return (rebalance(makeNode(L, k, v, L')), k', v', R')
```



Analogous to typical BST split!

```
fun split_rank(T: BST<K,V>, i: int) -> (BST, k, V, BST):
  match T with:
    case Node(L,k,v,_,_,R):
      if i = size(L):
        return(L, k, v, R)
      else if i < size(L):
        (L', k', v', R') = split_rank(L, i)
        return (L', k', v', rebalance(makeNode(R', k, v, R)))
      else:
        (L', k', v', R') = split_rank(R, i-size(L)-1)
        return (rebalance(makeNode(L, k, v, L')), k', v', R')
```

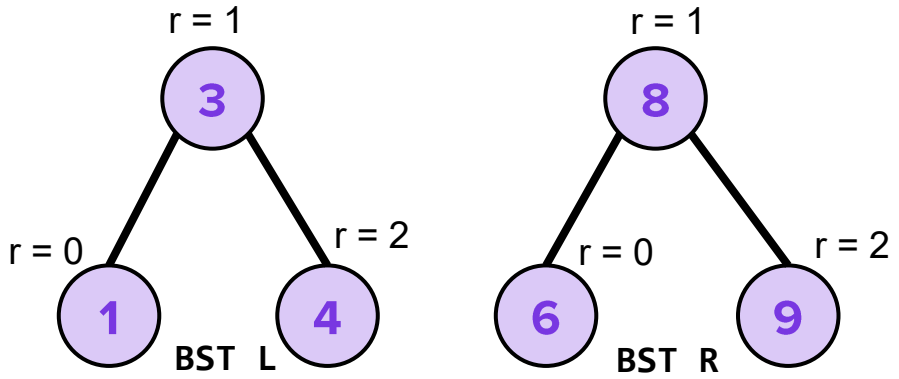
Case of i < |L|

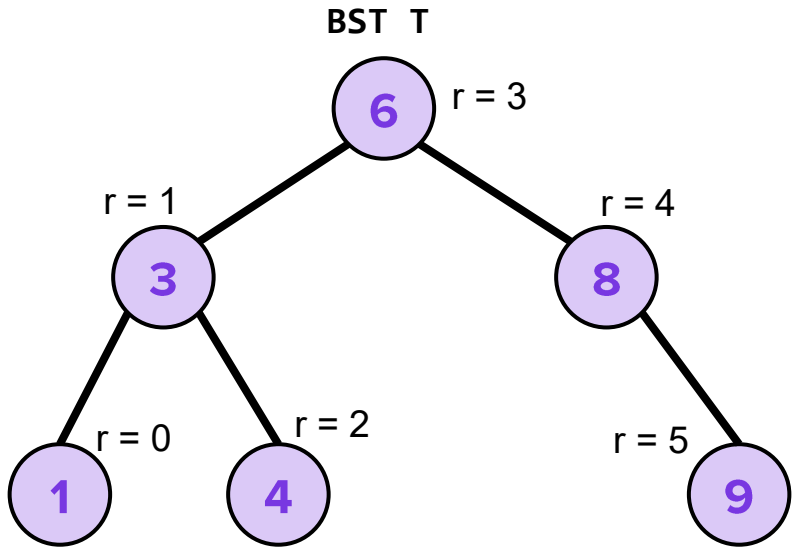
```
fun split_rank(T: BST<K,V>, i: int) -> (BST, k, V, BST):
  match T with:
    case Node(L,k,v,_,_,R):
      if i = size(L):
        return(L, k, v, R)
      else if i < size(L):
        (L', k', v', R') = split_rank(L, i)
        return (L', k', v', rebalance(makeNode(R', k, v, R)))
      else:
        (L', k', v', R') = split_rank(R, i-size(L)-1)
        return (rebalance(makeNode(L, k, v, L')), k', v', R')
```

Case of i > |L|

join_rank Implementation

- `join_rank (L, R)` returns T
  - T: BST representing sequence of (key, value) pairs in L followed by those in R
  - 🚨 Requires all keys in L to be less than all keys in R to preserve ordering constraint






```
fun join_rank(L: BST<K,V>, R: BST<K,V>) -> BST:
  if size(R) = 0:
    return L
  else:
    (_, k, v, R') = split_rank(R,0)
    return rebalance(makeNode(L,k,v,R'))
```

Analogous to typical BST join!

Order statistics operations

```
fun find_by_rank(T : BST<K,V>, i: int) -> (K,V):
  match T with:
    case Node(L,k,v,_,R):
      if i < size(L): return find_by_rank(L, i)
      else if i = size(L): return (k,v)
      else: return find_by_rank(R, i-size(L)-1)

fun rank_of_key(T : BST<K,V>, k : K) -> int:
  match T with:
    case Empty: return 0
    case Node(L,k',_,_,R):
      if k == k': return size(L)
      else if k < k': return rank_of_key(L, k)
      else: return size(L) + 1 + rank_of_key(R, k)
```

- `find_by_rank(T,i)` returns `(k,v)`
  - `(k,v)`: key-value pair at rank `i` of BST T
- `rank_of_key(T,k)` returns `i`
  - `i`: rank of key k of BST T

Other rank-based operations …

```
fun assign_rank(T: BST<K,V>, i: int, v: V)-> BST:
  (L,k,_,R) = split_rank(T,i)
  return rebalance(makeNode(L, k, v, R))

fun delete_rank(T: BST<K,V>, i: int) -> BST:
  (L,_,_,R) = split_rank(T,i)
  return join_rank(L,R)
```

- `assign_rank(T,i,v)` returns T’
  - T’: same BST as T, with value of element at rank `i` updated to v
- `delete_rank(T,i)` returns T’
  - T’: same BST as T, with element (key, value) at rank `i` deleted

Rank-based operations in action!

Consider the following:

- Want to insert keys $k_0, k_1, \ldots, k_{n-1}$ into a set at times `0,1,…,n-1`
- Then, want to answer `m` queries of the form `query(i, j)`: after insertion time `i`, return the key at rank `j`
- Use `O(nlogn)` space and `O(logn)` time per query

Example:

- Insert keys `[7, 5, 9, 1, 2]`
- `query(2, 1)`

After inserting at time 2:

- Set includes 7, 5, 9
- Key at rank 1 is 7

Solution

Main Idea: Make use of persistence and **STORE** the BST after each insertion

- We have n insertions – each stores an extra element and rebalances
- We **reuse** subtrees from previous BSTs that remain unchanged
- Thus, each insertion only takes (at most) `O(logn)` extra space

- For each query, we find the appropriate BST and call `find_rank`
- `find_rank` is `O(log n)`, so each query is `O(log n)`

```
t = An array of length n of
references to BSTs
B = Empty // An empty BST

for i=0 to n-1 do
  B = insert(B, k_i, k_i)
  t[i] = B

fun query(i,j):
  (answer,_) = find_by_rank(t[i], j)
  return answer
```

Dynamic Sequences

- Rank-based operations allow us to navigate the tree based on **rank**. So if there are no keys, just values, then:

- We can use this approach to represent just a sequence of values

  - Makes sequences **dynamic** by efficiently supporting operations such as splits and joins, as well as reduced values.

  - All operations are $O(\log n)$ work, space, functional, and persistent

  - Accessing is no longer O(1) work.

Example: The 6-7 problem

Suppose we want to maintain a collection of dynamic sequences of numbers under the following operations:

- splits, joins, inserts and deletes.
- Queries of the form: **contains67 S** which is true if the sequence **S** contains a 6 followed (somewhere after it) by a 7. (e.g. [2,6,5,7] satisfies this property but [2,3,7,6] does not.)
- All operations are O(log n) work where n is the length of the sequences involved.
- Try to solve this (hint: think about using reduced values)

Example: The 6-7 problem

Here's one solution:

keep in all nodes:

x6 : this subtree contains a 6

x7 : " " " " 7

x67 : " " " " "67"

c((x6, x7, x67), (y6, y7, y67)) =

((x6 ∨ y6), (x7 ∨ y7),

(x6 ∧ y7) ∨ x67 ∨ y67)

(Demo)

Summary

- Augmented BSTs allow us to answer maintain reduced values very efficiently.
- Persistence -- which allows access to previous versions of our data structures – can be used in algorithm design.
- Rank-based operations allow efficient position-based access of BSTs → potential dynamic sequence implementation
- Let’s do a DEMO