Binary Search Trees

1 Overview

Binary search trees (BSTs) are a versatile data structure. They can be used to store ordered data in a way that allows for easy searching, and can also be used to map an ordered set of keys to a set of values. The tree structure also lends itself well to parallelism. However, the costs of operations on BSTs can sometimes be hard to reason about. With n keys in a perfectly balanced BST, the height of the tree will be $O(\log n)$. However, in the worst case, we can have the same set of keys in a completely unbalanced BST, leading to an $\Omega(n)$ height tree.

Thus, computer scientists have developed various ways to ensure that BSTs stay approximately balanced, ensuring that we have logarithmic time complexity on basic operations. You have already seen in previous classes some data structures, such as AVL trees and red-black trees, which are used to make sure that BSTs stay balanced. In this class, we will consider **treaps**, which are BSTs which use randomness to ensure they stay balanced with high probability.

2 The Binary Search Tree Data Structure

Definition: Functional Binary Search Tree

From a functional point of view, a **binary search tree** is a recursive data structure:

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

A generic BST is *parameterised* over the type of its keys `K` and its values `V`. The base case comes when our BST is equal to `Empty`. Our recursive case is that of `Node`(L, k, v, s, R). A node carries two subtrees, L and R, as well as a **key**, which is what we use to determine the ordering of our BST. For a BST to be valid, it must satisfy the **BST invariant**.

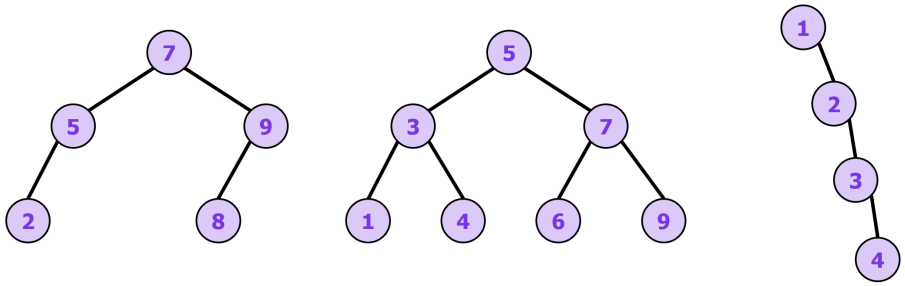
Definition: BST Invariant

A binary tree with root key k and left/right subtrees L and R satisfies the BST invariant if k is greater than the key of every node in L and less than the key of every node in R.

For this to make sense, we require that the type of keys K is totally ordered, meaning that for any two keys $k_1, k_2 \in K$, we can either conclude that $k_1 < k_2$, $k_1 = k_2$, or $k_1 > k_2$.

Each node also contains a **value** v. The type of values does not affect the ordering of the BST. These values are simply stored at each node and associated with the corresponding key. Lastly, each node also has a `size` parameter that tells us how many nodes are in the subtree of the BST, rooted at that Node.

Here are three examples of binary search trees. We omit the values here to emphasize that the tree is structured based on the keys. Additionally, some of the nodes do not have two children. In each of these cases, the missing child(ren) are simply `Empty`.



Here are some types of common operations on binary search trees, as well as their descriptions.

- **size**(`T`): Returns the number of elements in T
- **find**(`T, k`): Finds the value associated with key k (or returns that none exists)
- **insert**(`T,k,v`): Return a new tree resulting from inserting key k with value v into T
- **delete**(`T,k`): Return a new tree resulting from deleting key k from T
- **first**(`T`): Return the key and value of the element with the least key
- **last**(`T`): Return the key and value of the element with the greatest key
- **next**(`T,k`): Return the key and value of the element with the least key greater than k
- **prev**(`T,k`): Return the key and value of the element with the greatest key less than k
- **join**(`L,R`): Return a tree containing the contents of both L and R, where $L < R$
- **union**(`T1,T2`): Return a tree containing the union of the contents of T_1 and T_2
- **intersection**(`T1,T2`): Return a tree containing the intersection of the contents of T_1 and T_2
- **difference**(`T1,T2`): Computes the set difference of the contents of two trees
- **split**(`T,k`): Returns two trees corresponding to splitting T into the keys less than k, and keys greater than k, as well as the value corresponding to k (if it exists)
- **getRange**(`T,k1,k2`): Return a tree containing the keys k in T for which $k_1 < k < k_2$

Note that these operations suffice to implement the **Sorted Set** and the **Sorted Dictionary** interfaces! We'll cover the implementation of most of these functions. An extremely important

note is that all of the *update* operations on our BST do not modify the given tree! It is a persistent data structure API.

Definition: Persistent Data Structures

A persistent data structure preserves previous versions when modified. Operations like insert and delete return new trees while sharing unchanged subtrees with the original. This allows multiple versions to coexist efficiently and is especially valuable in functional and parallel algorithms.

2.1 Maintaining Balance

How can we implement the above operations for our binary search trees? After all, we might be using particular balancing invariants in our BSTs to keep the costs of these operations low. Thus, our implementations of these functions would need to maintain these invariants somehow. One possibility would be to write different versions of these functions for each BST implementation. This would mean we would need separate implementations for AVL trees, red-black trees, and so on, which could become very tedious!

Instead, it turns out that there is an elegant way to implement *all* of these operations such that all of the balancing logic goes in a *single function*. What we will do is write these functions in terms of another function called `rebalance`. The `rebalance` function takes a BST and rebalances its root node. Using this function, we can implement each of the above functions. In this way, we only have to implement one function, `rebalance`, for each variety of balanced tree (AVL, red-black, Treap, etc)!

Definition: rebalance

The `rebalance` function takes a BST and rebalances its root node. More specifically, if the tree is `Empty`, it returns `Empty`. Otherwise, if the tree is `Node`(L, k, v, s, R) where L and R both satisfy the implementation's balance invariant, it returns a BST consisting of the same (key,value) pairs but which satisfies the implementation's balance invariant.

No tree value that violates the balance invariant is ever returned by an operation; any temporary violation is repaired by calling `rebalance` before returning.

Remark: Rebalance rebalances the root

Note the critical subtlety here that `rebalance` *only* rebalances the root node; it assumes that L and R are already balanced, while the root may not be. It does not balance an entirely unbalanced tree! This works because all of our algorithms are recursive and will return balanced trees, so therefore when we construct a new tree from an existing pair of balanced trees L, R and a new root node, the only place in which there can be a violation of the balance invariant is at the root, which we correct by calling `rebalance`.

As we now have our rebalance function, we will assume henceforth that we are working with balanced binary search trees. So what does balanced mean? The height of a binary search tree is at least $\lceil \lg(n+1) \rceil$, where n is the number of nodes. As it turns out, this height bound is

achievable for any BST if we do a sufficient amount of rotation. This can be expensive, though. Therefore, instead of aiming for perfect balance, we aim for asymptotic balance.

Definition: Balanced Binary Search Tree

A balanced binary search tree is a binary search tree on n nodes with height $O(\log n)$.

In other words, a balanced BST can be a constant factor off from its ideal height. Indeed, red-black trees have height approximately equal to $2\log n$, while AVL trees have height around $1.618\log n$. Thus, expecting a good balanced binary search tree implementation to have height around twice that of the optimal height is reasonable.

3 Implementing Binary Search Trees

Now that we've defined some of our types and functions for binary search trees, it's time to discuss more of the things that happen under the hood. Note that particular balancing schemes might need to add additional information to the nodes. For example, red-black trees must also store the color. We will therefore use a helper function for creating nodes:

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

The purpose of `makeNode` is to maintain bookkeeping invariants, such as the `size` field. Specific balancing schemes may extend this function to initialize additional metadata, for example, the color bit in a red–black tree or the priority in a treap[1].

Note that `makeNode` does not perform any rebalancing; restoring balance invariants is handled explicitly and separately by calls to `rebalance`. This makes it clear when rebalancing occurs.

You may notice that the code that follows is much shorter than typical AVL or red–black tree implementations. In that code, the update operations contain the algorithms for rebalancing the trees when the new element causes the balance invariants to be violated. Isolating that complex logic inside the `rebalance` function allows our code to stay concise and generic!

3.1 Read-only operations

3.1.1 Size

The size function returns the size of a tree, i.e., the number of elements in it.

Algorithm: BST: size

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

[1] In an object-oriented language, this role would naturally be handled by a constructor for the `Node` type. In a functional setting, such initialization logic must instead be expressed using helper functions like `makeNode`.

3.1.2 Find

BSTs are called "search trees" because they make searching for a particular key efficient. Thus, we begin to define tree operations by searching through a binary search tree:

Algorithm: BST: find

```
fun find(T: BST<K,V>, k: K) -> option<V>:
  match T with:
    case Empty: return NONE
    case Node(L,k',v,_,R):
      if k < k': return find(L, k)
      else if k > k': return find(R, k)
      else: return SOME(v)
```

At each node, the algorithm compares the node's key to the target key, and if they match we are done. Otherwise, we recursively search the left or right subtree depending on whether the target key is less or greater than the node's key. This version of the `find` function finds the value corresponding to a particular key if it exists in the BST. If we have a tree that only stores keys at each node (no values) then we can return true or false instead of SOME or NONE, respectively.

3.2 Single-update operations

We now give implementations of the *update* operations on BSTs. Critically, remember that none of these functions ever modify the input tree. Instead, each operation returns a *new* tree reflecting the updated state, while the original tree remains unchanged and can still be used.

3.2.1 Insert

`insert` follows the structure of `find`; to insert into a binary search tree, we find the location where that key ought to exist based on order, then add it.

Algorithm: BST: insert

```
fun insert(T: BST<K,V>, k: K, v : V) -> BST:
  match T with:
    case Empty:
      return rebalance(makeNode(Empty, k, v, Empty))
    case Node(L,k',v',_,R):
      if k < k':
        return rebalance(makeNode(insert(L, k, v), k', v', R))
      else if k > k':
        return rebalance(makeNode(L, k', v', insert(R, k, v))
      else:
        return rebalance(makeNode(L, k, v, R))
```

In this particular implementation, if k already exists in the tree, we simply *replace* its value at its node. You might instead prefer that insert does nothing if the key exists and write a separate function for updating values of existing keys. That would be equally simple to implement.

3.2.2 first

`first` needs to find the minimum-key node in the tree. This is just the node that is "leftmost", i.e., it is the leftmost node of the left child of the root, which will be the first node on the left spine of the tree that itself has no left child.

To implement `first`, we therefore just recursively search the left children until we hit a node that has an `EMPTY` left child, and that is the answer.

Algorithm: BST: first

```
fun first(T : BST<K,V>) -> option<(K,V)>:
  match T with:
    case Empty: return NONE
    case Node(Empty,k,v,_,_):
      return SOME((k,v))
    case Node(L,_,_,_,_):
      return first(L)
```

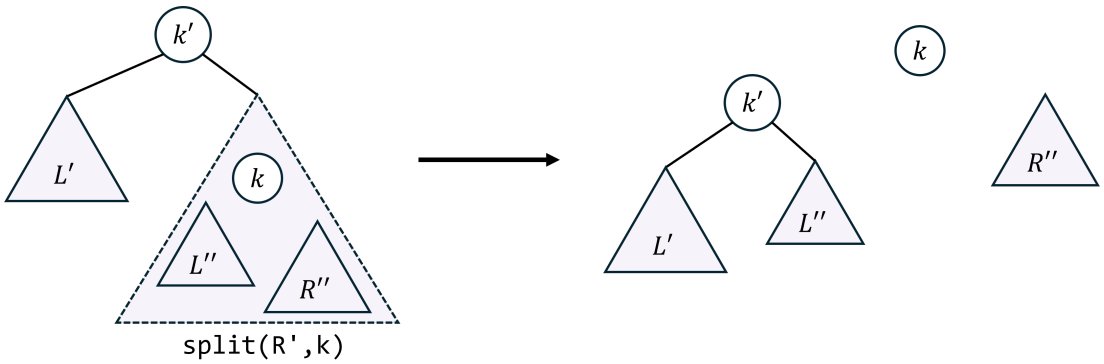
You can imagine the implementation of `last` would be pretty much the same but the opposite.

3.3 Bulk tree modification: Split and Join

3.3.1 Split

One of the core functions for *parallel* BST operations is the **split** operation. Many other complex parallel BST algorithms will use `split` and `join` as subroutines (indeed, `join` itself uses `split`).

The `split` function takes a tree T and a key k and returns (L, x, R), where L is a binary search tree containing all keys in T that are less than k (as well as their associated values), and R is a binary search tree containing the keys that are greater than k. The middle value x is equal to SOME v if k exists in the tree and is associated with value v, and is equal to NONE otherwise.



We implement the `split` function recursively. Given a tree and a key, we check whether the root of the tree has a key k' that is less than, greater than, or equal to k. Suppose, as in the picture above, that $k > k'$, then the algorithm has to recursively split the right subtree. After

splitting the right subtree, it joins the left part of the split back onto the old root and its left subtree. When $k < k'$ the mirror opposite happens. The base cases are when we find that $k = k'$ in which case the split simply returns the left and right subtrees and the value of the root, or the trivial base case of an empty tree.

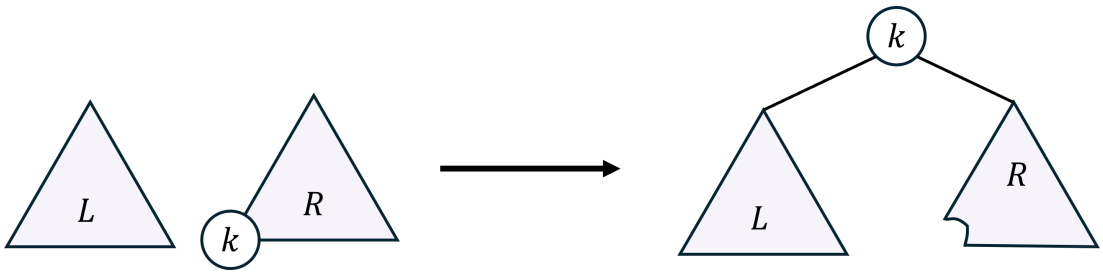
Algorithm: BST: split

```
fun split(T: BST<K,V>, k: K) -> (BST, option<V>, BST):
  match T with:
    case Empty:
      return (Empty, NONE, Empty)
    case Node(L',k',v',_,R'):
      if k < k':
        L'', optV, R'' = split(L', k)
        return (L'', optV, rebalance(makeNode(R'', k', v', R')))
      else if k > k':
        L'', optV, R'' = split(R', k)
        return (rebalance(makeNode(L', k', v', L'')), optV, R'')
      else:
        return (L', SOME(v'), R')
```

3.3.2 Join

join is the counterpart to `split`, the other core operation for parallel BST algorithms; `split` takes a BST and breaks it into two, L and R, while `join` takes two BSTs L and R and glues them together into one BST. Note that it is strictly a precondition of `join` that all the keys in L are less than all the keys in R. Without this precondition, `join` can not be implemented efficiently.

Let us think about the algorithm design idea behind `join`. We are given two trees L and R and we want to make them part of a single BST. Well, if we had a root node to put between them, it would be easy, we would just make a new BST with L and R as the children, but we *don't* have a root node to put between them. So the trick is, let's steal one! We can remove the minimum element of R (or the maximum of L) and use that as the root of the new tree. We can use `first` to do the job of finding the minimum element of R.



We need to do more than just *find* the minimum element of course, we also need to remove it since otherwise we would duplicate the minimum element. We could write a new algorithm from scratch for this, but the easiest way is to realize that we can just reuse the `split` function and split on the minimum! Splitting on the minimum key k means we will always get

`(Empty, SOME(v), R')` where R' no longer contains k.

Algorithm: BST: join

```
fun join(L: BST<K,V>, R: BST<K,V>) -> BST:
  match first(R) with:
    case NONE: return L
    case SOME((k,v)):
      _, _, R' = split(R, k)
      return rebalance(makeNode(L, k, v, R'))
```

Of course, the resulting BST from gluing together L with the minimum of R and the remainder of R will satisfy the BST invariant, but it might not be balanced! Therefore we must remember to call `rebalance` on the resulting tree before returning it.

3.4 Bulk filtering

3.4.1 Get Range

The `getRange` function takes a tree T and two keys k_1 and k_2, where $k_1 < k_2$. It returns the portion of T which has keys between k_1 and k_2 (exclusive). It is implemented using two calls to the `split` function, first to get all elements with keys greater than k_1, then to get elements with keys less than k_2 (but greater than k_1).

Algorithm: BST: getRange

```
fun getRange(T : BST<K,V>, k1 : K, k2 : K) -> BST:
  _, _, R = split(T, k1)
  M, _, _ = split(R, k2)
  return M
```

If desired, the implementation can be modified to add keys k_1 and k_2 to the resulting BST to make the range k_1, k_2 inclusive instead of exclusive.

3.4.2 Filter

The `filter` function on binary search trees is similar to the filter function on sequences. It takes a predicate and a binary search tree, and returns a new tree that contains only the nodes for which the predicate is true. We can implement different variants of `filter` depending on whether we want the predicate to act on keys, values, or both.

Here, `filter` is such that the predicate function takes a **value** and returns true or false. We could define a similar function, `filterKey`, for which the predicate takes a key and a value.

Algorithm: BST: filter

```
fun filter(p : V -> bool, T: BST<K,V>) -> BST:
  match T with:
    case Empty: return Empty
```

```
    case Node(L,k,v,_,R):
      fL, fR = parallel (filter(p, L), filter(p, R))
      if p(v):
        return rebalance(makeNode(fL, k, v, fR))
      else:
        return join(fL, fR)
```

As with the previous functions, we case on whether the tree is empty, and then recursively filter the remaining parts of the tree. The left and right subtrees can be filtered in parallel because they are independent. The algorithm decides whether to keep the current (k, v) pair based on whether $p(v)$ is true, and then either joins the left and right halves together *without* (k, v) or stitching (k, v) back between the filtered left and right halves.

Theorem: Work and Span of Filtering

The `filter` function on a BST runs in $O(n)$ work and $O(\log^2 n)$ span assuming that `rebalance` takes $O(\log n)$ time[a] and that p can be evaluated in constant time.

[a] This is true for all reasonable balancing schemes (AVL, red–black, Treap).

Proof. We will use the fact that `join` has $O(\log n)$ work and span. Under the assumption that tree is balanced, the work of the filter function will have the recurrence

$$W(n) = 2W\left(\frac{n}{2}\right) + O(\log n)$$

This comes out to be leaf-dominated, with a solution of $O(n)$. The span is similar:

$$S(n) = S\left(\frac{n}{2}\right) + O(\log n)$$

This winds up being $O(\log^2 n)$, since there are $\log n$ levels of the recurrence and at each level the join or rebalance costs $O(\log n)$. □

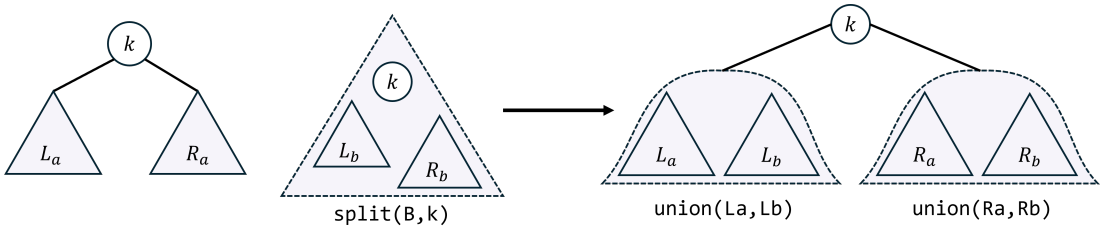
3.5 Bulk Set-Theoretic Operations

Now that we've seen some functions on binary search trees, we can consider one of their uses—as an implementation of the sorted set ADT. BSTs can be used to efficiently implement the `union`, `intersection`, and `difference` operations.

3.5.1 Union

One of the first operations to perform with two sets is to take their union. There is one caveat: what value do we return in our output set if a key appears in both of the input sets? Since we are mostly interested in the `union` operation for the set ADT, we don't particularly care about values in this case, so we will take the simplest solution, in which we will just take the value from the first set.

Essentially, `union` will operate by taking the key k at the root of the first tree, then splitting the other tree at that key, which allows it to recursively take the union of each side. The trees are then stitched back together at the end and balanced.



Algorithm: BST: union

```
fun union(A: BST<K,V>, B: BST<K,V>) -> BST:
  match (A, B) with:
    case (Empty, _): return B
    case (_, Empty): return A
    case (Node(La, k, v, _, Ra), _):
      Lb, _, Rb = split(B, k)
      L, R = parallel (union(La, Lb), union(Ra, Rb))
      return rebalance(makeNode(L,k,v,R))
```

The base cases for this function come when either tree is empty, in which case the non-empty tree is returned. Additionally, the case where A has size 1 can be logically viewed as a base case (and will be for our analysis): once this point is reached, splitting and reassembling the tree to include (k, v) is equivalent to simply inserting (k, v) into B.

Remark: Union with tiebreaking

A more complex tiebreaking mechanism in the presence of equal keys with different values is to have the `union` function take an additional argument which is a function `(V, V) -> V` which selects which of the two values to keep, or possible *combines* the two values in some way, such as adding them together.

Remark: Handwaving imminent

In our proof of the cost bounds of `union` below, we are going to make the additional assumption that A is not just balanced, but *perfectly balanced*. That is, we will assume that A is a binary search tree where every internal node has two children and every leaf node has the same depth.

Without this assumption—just under the assumption that the tree has approximate balance as induced by a balancing scheme such as AVL Trees, Red-Black Trees, Treaps, etc.—the proof goes from being under two pages to being a 40-page research paper, which you are welcome to read **here** if interested!

Theorem: Cost of Union

Given two balanced binary search trees A and B of size m and n respectively, assuming WLOG that $m \leq n$, union costs

$$O\left(m + m\log\left(\frac{n}{m}\right)\right)$$

work in $O(\log m \log n)$ span.

Proof. To satisfy the WLOG assumption that $m \leq n$, we can simply swap the two trees before calling union if A is larger than B. So, this assumption is implementable.

At each recursive call, the algorithm selects the root key of the first tree A, splits the second tree B at that key, and then recursively unions the left and right subtrees. The non-constant work performed locally at each call consists of a `split` of B and a `rebalance` of the result of the recursive calls. Splitting B costs $O(\log n)$ work, and rebalance costs at most $O(\log(m+n))$ work. Since $m \leq n$, this is $O(\log n)$ work.

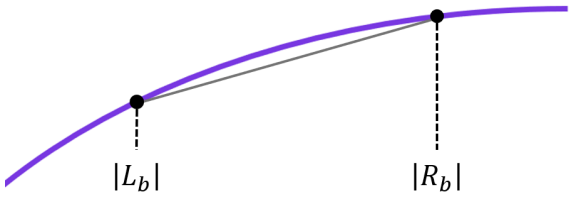
Let $W(m,n)$ denote the work to union trees of sizes m and n. If A is a singleton, the work is just $O(\log n)$ for the split and rebalance. Otherwise, we recurse on the left and right subtrees of A. Under our assumption that A is perfectly balanced, it gets split exactly in half, so we write

$$W(m,n) = \begin{cases} 1 + \log n & \text{if } m = 1, \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log n & \text{otherwise,} \end{cases}$$

where $|L_b| + |R_b| \leq n$. Note that B is *not* necessarily split exactly in half, since the sizes $|L_b|$ and $|R_b|$ depend on how many keys are less or greater than the root of A, which can be arbitrary.

We will prove however, is that B being split evenly is the *worst case*. This is rather counterintuitive; for most algorithms, splitting exactly in half is the best case. The reasoning turns is that we are paying the logarithms of the sizes of B (i.e., $O(\log n)$), and logarithm is a concave function.

More specifically, lets apply our favourite recurrence-solving technique, the **brick method**, and consider the cost of the children relative to the root of the recurrence. The local cost of the root is $\log n$, while the cost of the children is $\log|L_b| + \log|R_b|$ where $|L_b| + |R_b| \leq n$.



Since logarithm is a *concave* function, the cost at any point on the curve between $|L_b|$ and $|R_b|$ is higher than the average cost at $|L_b|$ and $|R_b|$. Therefore, the cost is maximum/worst when $|L_b| = |R_b| = n/2$. Formally, this is known as **Jensen's Inequality**.

Therefore, the worst-case work of the children is $2\log(n/2) = 2\log n - 2$. Asymptotically, this is a constant factor greater than the work of the root, and therefore this recurrence in the worst case is **leaf dominated**.

The recursion continues until the size of the A subtree reaches 1. Since the total number of nodes in A is m and we assume that A is perfectly balanced, there are $m/2$ leaves, each of which costs $W(1, b_i)$, where each b_i is the size of the corresponding subtree of B at that leaf. These values satisfy

$$\sum_{i=1}^{m/2} b_i \leq n.$$

From our recurrence, each such leaf costs $1 + \log b_i$. Therefore, the total work at the leaves is

$$\sum_{i=1}^{m/2} (1 + \log b_i) = \frac{m}{2} + \sum_{i=1}^{m/2} \log b_i.$$

Again, using the fact that the logarithm function is concave, this sum is maximized when the b_i are all equal (*Jensen's inequality*), so the summation is maximized as

$$\sum_{i=1}^{m/2} \log b_i \leq \frac{m}{2} \log\left(\frac{2n}{m}\right).$$

Since we are in a leaf-dominated recurrence, the work at internal nodes of the recursion tree is asymptotically dominated by the work at the leaves. Therefore, the total work of `union` is

$$O\left(m + m \log\left(\frac{n}{m}\right)\right).$$

For the span, note that there are $O(\log m)$ levels of recursion (corresponding to traversing A until reaching its leaves) and at each recursive call the local work is $O(\log n)$. Therefore the total span is $O(\log m \log n)$. □

It is possible to achieve an even better span of $O(\log n)$ for union using more advanced techniques beyond the scope of this course. If we use `union` in the implementation of future algorithms, we will therefore cite that it costs $O(\log n)$ span, even though we have not implemented or analyzed that version.

We can also use a similar algorithm for both intersection and set difference. The only changes we make are in handling what happens when k is and is not found in B. For intersection, we include k only if it is found in B. Inversely, for difference, we will exclude k if it is found in B.