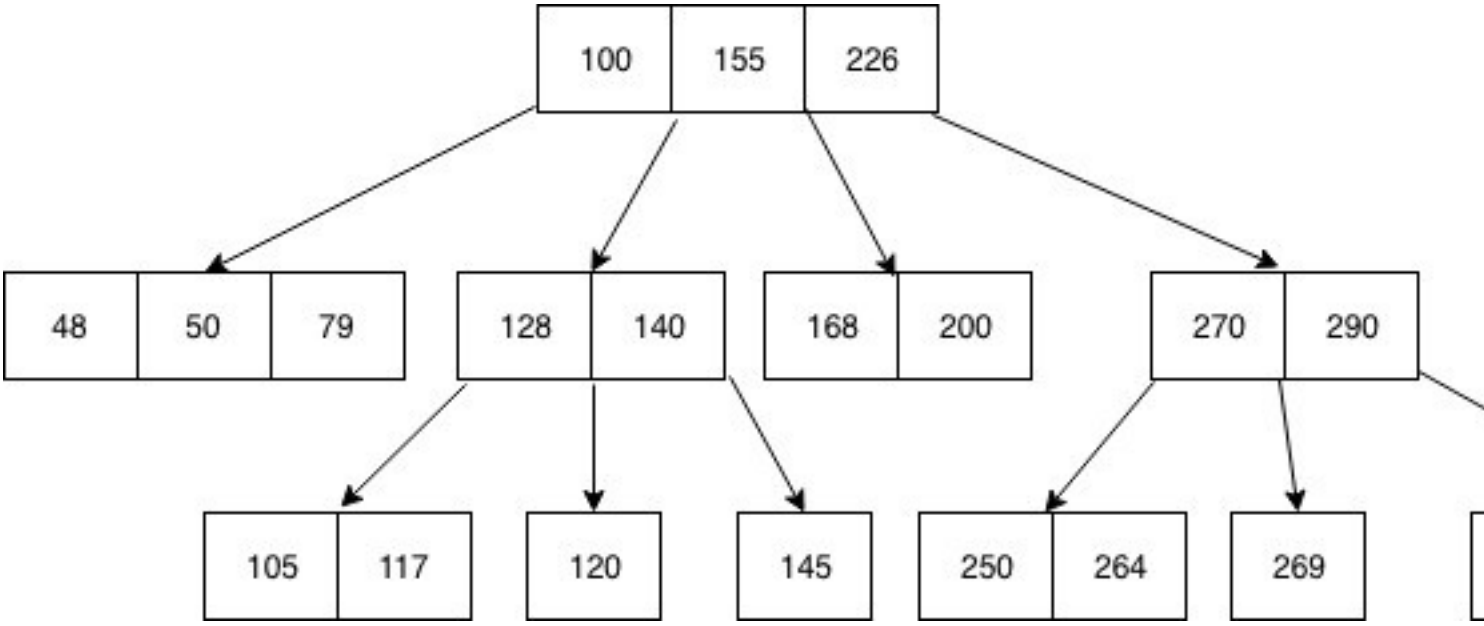

Figure 1: btree

Theme Song: Here Comes The Sun

You've surely encountered binary search trees before - data structures that allow quick ($O(logn)$) search. A similar search structure, the B+Tree, is like a binary tree generalized to `n` children, with extra optimizations to make search more efficient. In this assignment, you will implement a B+Tree from scratch and use it to store and retrieve data.

---

1 Background Knowledge

1.1 BTrees

To understand B+Trees, we should first cover what a BTree is. A BTree is a self-balancing tree data structure that allows logarithmic time search, insertion, and deletion. It is a generalization of the binary search tree in that each node can have more than 2 children. A BTree of order m is formally defined as a tree that satisfies the following properties:

- Each node has at most m children,
- Every non-leaf (internal) node has at least $\lceil m/2 \rceil$ children,
- The root has at least two children unless it is a leaf node,
- A non-leaf node with k children contains $k-1$ keys,
- All leaves appear on the same level and carry no information.

One way to visualize what an internal node looks like is an array of $2k-1$ values, alternating between a child node pointer and a key:

```
[node_1, value_1, node_2, value_2, ..., value_k-1, node_k]
```

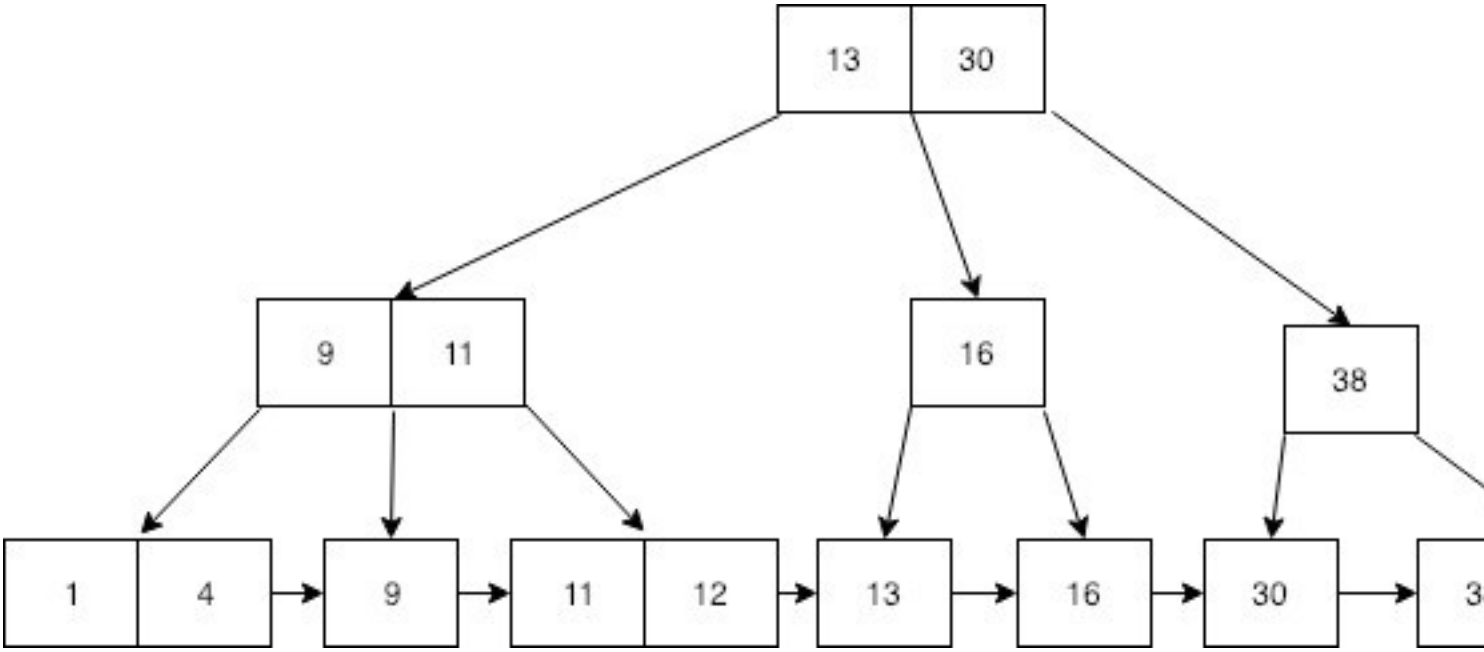

Figure 2: b+tree

Critically, all of the values of a node in between two values must be between those two values. So, there will never be a value less than `value_1` or greater than `value_2` in `node_2` or any of its descendants.

This data structure supports search, insertion, and deletion. Updates are supported as well, but can be thought of as a deletion followed by a subsequent insertion. We describe each of these algorithms.

Search - to search for a value, we start at the root node. If the value we're looking for is in this node, we return it. Otherwise, we binary search over the keys to find the branch that our value could be in, and repeat this process for that child node.

Insertion - to insert a value, we start at the root node. Then, we traverse to a leaf node and insert the value in the correct spot. If the leaf node *overflows* (i.e. has more than k values), then we need to *split* it so that our BTree is still valid. To split a leaf node, we create a new leaf node and transfer half of the values (not including the median) over to the new leaf node. Then, we take the median element and push that value up into the parent node to serve as a separator key for our two new nodes. If the parent node overflows, we repeat this process for that node.

Deletion - to delete a value, we first find the value (if it exists), then we remove it. This operation becomes rather complicated when nodes underflow (less than $\lceil m/2 \rceil$ children), or when deleting a value from an internal node; we leave it as an exercise to the reader to explore this operation.

To explore how some of these BTree operations work, try out this online visualization.

1.2 B+Trees

Now that we understand BTrees, it's time to explore what a B+Tree is. A B+Tree is a BTree with a few important changes:

- Each leaf node has a pointer to its right neighbor. This makes linear scans much easier, since we don't have to traverse the tree structure to get to the next leaf node.

- Internal nodes store duplicates of values in leaf nodes. This means that if we were to scan every leaf node, we would retrieve every value in the tree.

Otherwise, everything else is the same as in a BTree. This structure is better optimized for when data you're looking for is on disk, since internal nodes only need to store keys, and then all of the large values are kept on leaf nodes. In our scheme, each node corresponds to a page, meaning the number of pages we have to read in to find a particular value is equal to the height of our B+Tree.

The operations in a B+Tree are slightly different:

Search - to search for a value, we start at the root node and traverse all the way to the correct leaf node, using binary search to descend quickly. Once we arrive at the leaf node, binary search for the value.

Insertion - to insert a value, we start at the root node. Then, we traverse to a leaf node and insert the value in the correct spot. If the leaf node *overflows* (i.e. has more than k values), then we need to *split* it so that our BTree is still valid. To split a leaf node, we create a new leaf node and transfer half of the values (including the median) over to the new leaf node. Then, we take a copy of the median element and push that value up into the parent node to serve as a separator key for our two new nodes. If the parent node overflows, we repeat this process for that node. The main difference here is that we duplicate the median key when we split.

Deletion - to delete a value, we first find the value in a leaf node(if it exists), then we remove it. This operation becomes rather complicated when nodes underflow (less than $\lceil m/2 \rceil$ children), or when deleting a value from an internal node; we leave it as an exercise to the reader to explore this operation.

To explore how some of these B+Tree operations work, try out this online visualization.

1.3 Cursors

One of the B+Tree's biggest optimization is the existence of links between leaf nodes to allow faster linear scans. A cursor is an abstraction that takes advantage of these links to carry out selections quickly. A cursor typically has two operations: `next` and `output`. `next` steps the cursor over the next value; if we are in the middle of a node, point to the next value, otherwise, skip over to the next node. `output` returns the value that the cursor is currently pointing at for the surrounding program to handle. Critically, in a B+Tree, a cursor never needs to be in an internal node, making it an easy to understand and use abstraction.

---

2 Assignment Spec

2.1 Overview

In this project, you'll first do a little bit of setup, then you'll be writing a B+Tree, nodes, cursors, and all!

The following are the files you'll need to alter:

- `cmd/bumble/main.go`
- `pkg/db/db.go`
- `pkg/btree/node.go`
- `pkg/btree/cursor.go`

- `pkg/btree/btree.go`

And the following are the files you'll need to read:

- `pkg/db/db.go`
- `pkg/btree/entry.go`
- `pkg/btree/btree_subr.go`

2.2 Subroutine Code

We've provided subroutine code so that you don't have to worry about the fact that all of this data is actually being stored on pages. We handle the marshalling and unmarshalling of data, you handle the actual logic of the B+Tree.

You will need to read the `btree_subr.go` file to implement this assignment. You don't need to understand exactly how every function works, just understand what they do so that you know when you need to call them. If you ever find yourself manipulating bytes yourself, you have done something wrong and need to leverage the subroutine code more. You shouldn't change the subroutine code unless a TA has instructed you to do so.

A common pattern you will see is how we get nodes; we use defer to ensure that the page is returned to the pager eventually, and then use one of the following three converter functions to use the page as a header, leaf node, or internal node:

```
// Get page and put later:
page, err := table.pager.GetPage(pagenum)
if err != nil {
    return nil, err
}
defer page.Put()
// And then one of the following:
header := pageToNodeHeader(page)
leaf := pageToLeafNode(page)
internal := pageToInternalNode(page)
```

2.3 Part 0: Instrumenting Existing Code

Relevant files:

- `cmd/bumble/main.go`
- `pkg/db/db.go`

Before you get started, we have to provide you with a few stencil packages. You will download the `db`, `utils`, `config`, and `btree` zip files and extract the folder containing Go files; all of these should be placed in the `pkg` folder, replacing old versions of the package when necessary. When finished, your filestructure should look like this:

```
./
    - cmd/
        - bumble/
    - pkg/
        - btree/
        - config/
        - db/
        - utils/
        - list/
```

```
- pager/
- repl/
```

Next, you will need to uncomment all of the code blocks prefixed with `[BTREE]` in your `main.go`. If you're not sure if your `main.go` is working, we've provided a copy that should work below.

The `db` package serves as an interface between the B+Tree and the rest of your code; since a B+Tree is one of many potential index schemes (foreshadowing), this allows us to support multiple kinds of indexes rather easily. We recommend poking through the code to get a sense of what it does and how it works. Once you have added the above code, you should have access to the `dbREPL` functionality, which supports the following commands:

- `create <type> table <table>` - Creates a table with a particular type. For now, only supports type of "btree". You must create a table before using it.
- `find <key> from <table>` - Finds a particular key-value pair from the given table.
- `insert <key> <value> into <table>` - Inserts a particular key-value pair into the given table. Does not allow insertion of duplicates.
- `update <table> <key> <value>` - Updates a particular key-value pair in the given table.
- `delete <key> from <table>` - Deletes a particular key-value pair from the given table.
- `select from <table>` - Returns all of the key-value pairs from the given table.
- `pretty from <table>` - Pretty-prints the internal structure of the index.

2.4 Part 1: Node Functions

Relevant files:

- `pkg/btree/node.go`
- `pkg/btree/btree_subr.go`

We'll start by implementing B+Tree functions on the node level. Note that a lot of these functions will require you to use functions defined in the `btree_subr` file - read through this file thoroughly first, then begin the rest of the assignment. We've implemented `get` for you, since it deals with entries which you shouldn't have to worry about yet. Implement the following functions:

```
func (node *LeafNode) search(key int64) int64
func (node *LeafNode) insert(key int64, value int64, update bool) Split
func (node *LeafNode) delete(key int64)
func (node *LeafNode) split() Split

func (node *InternalNode) search(key int64) int64
func (node *InternalNode) insert(key int64, value int64, update bool) Split
func (node *InternalNode) insertSplit(split Split) Split
func (node *InternalNode) delete(key int64)
func (node *InternalNode) split() Split
```

Note that we do not expect you to handle coalescing underflowing nodes on deletion. It's perfectly fine for your delete implementation to simply remove a value from the leaf node; don't worry about maintaining the "at least $\lceil m/2 \rceil$ values" invariant.

You may be wondering what a `Split` is - a `Split` is a struct that indicates whether the operation below triggered a node split, and if it did, the node that was pushed up, so that the calling function can receive it. Let's say we're at node A and we call insert on our child; our child comes back and tells us that it has split, and pushes a new node up to us. We then add that node to node A, and potentially split again to our parent. In this project, we handle root node splitting for you; however, you must handle the cases where internal and leaf nodes split!

Some hints to get you started:

- Use the `sort` package's implementation of binary search instead of writing it yourself.
- The following node functions should be used to get and manipulate keys and values: `getKeyAt`, `getValueAt`, `updateKeyAt`, `updateValueAt`, `updateNumKeys`, `updatePNAt`, `modifyCell`.
- To create a new leaf node, use `createLeafNode(node.page.GetPager())`. To create a new internal node, use `createInternalNode(node.page.GetPager())`.
- To get a node's child, use `getChildAt`. Note: as the function's name and documentation suggests, this helper function increases the pinCount of the page storing the child, and hence the node must be Put accordingly after use.
- After using any function that creates or gets a node, be sure to run `defer newNode.getPage().Put()` so that you properly release the new page you just read in.
- B+Tree leaf nodes have a right sibling; to change it, use `setRightSibling`.

2.5 Part 2: Cursor Functions

Relevant files:

- `pkg/btree/cursor.go`
- `pkg/btree/btree.go`

Next, we'll want to take advantage of the B+Tree structure and define a cursor that can scan over leaf nodes. We've defined the basic cursor functions for you and filled out `tableStart` so you have a sense of how to approach the rest of the functions. With that, implement the following functions in `pkg/btree/cursor.go`:

```
func tableEnd(table *BTreeIndex) (*Cursor, error)
func tableFind(table *BTreeIndex, key int64) (*Cursor, error)
func tableFindRange(table *BTreeIndex, startKey int64, endKey int64)
```

And the following in `pkg/btree/btree.go`:

```
func (table *BTreeIndex) Select() ([]entry.Entry, error)
```

Some hints to get you started:

- Use the `tableStart` implementation as a guide to how to interact with nodes and their underlying page representation.
- Remember to always run `defer page.Put()` whenever you traverse a node!
- To convert a cursor's current position into an Entry, use `getEntry`.

2.6 Testing

Run our unit tests on Gradescope. We've provided a subset of the tests here.

2.7 Getting started

To get started, download the following stencil packages: db, utils, config, and btree. We've provided an updated `main.go` file here. See Part 0 on how to integrate this with the rest of your code.

Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!