CS 529: Advanced Data Structures & Algorithms
Assignment 1

Nathan Chapman, Hunter Lawrence, Andrew Struthers

January 11, 2024

Creating a 3-2 Tree

A binary search tree (BST) is usually an excellent choice of data structure because inserting, deleting, and searching all take $O(n)$. This linear complexity arises because the tree can be "completely" unbalanced, yielding a linear search. B-Trees circumvent this by rebalancing upon each insertion and deletion.

An example of how the re-balancing is implemented for B-Trees can be shown in the context of the simplest B-Tree: the 3-2 tree. The following algorithms implement insertion, traversal and "bumping and breaking" of the nodes of a 3-2 tree.

Recursive

The core functionality of what differentiates a B-tree from a BST (i.e. the automatic rebalancing) is encapsulated in `bumpAndBreak`.

Function bumpAndBreak(p)

Input: node p
Output: Null

```
1  if p has a parent then
2  |  insert p.keys[2] into p.parent.keys
3  else
4  |  create parent node with key p.keys[2]
5  remove p.keys[2]
6  create parent.left with key min p.keys
7  create parent.right with key max p.keys
8  remove parent.middle
```

We can use `bumpAndBreak` upon recursively traversing down to a leaf that, after insertion of the a new key, has 3 keys.

Function insertRecursive(p, k)

Input: node p, new key k
Output: 3-2 B-tree

```
1  if p is a leaf then
2  |  insert k into p.keys
3  |  sort p.keys
4  |  if length of p.keys = 3 then
5  |  |  bumpAndBreak(p)
6  else if k < p.keys[1] then
7  |  insertRecursive (p.left, k)
8  else if k > p.keys[end] then
9  |  insertRecursive (p.right, k)
10 else
11 |  insertRecursive (p.middle, k)
```

Finally, we can exectute the above algorithms for each element in a list of keys to create a 3-2 B-Tree.

Function buildTree(keyList)

Input: List of keys
Output: 3-2 Tree of given keys

```
1 create root node with key keyList[1]
2 foreach key ∈ keyList[2 : end] do
3 |  insertRecursive(root, key)
```

Iterative

We also wrote an interative version of the 3-2 Tree construction, seen below. For each key in the keys list, we call function `insertKey` with a reference to the head node of the tree and the key we want to insert. The algorithm then traverse iteratively down the tree from the head, finding the leaf where the key should be inserted. Once that leaf is found, insert the key into that leaf's keys, preserving the sorted order of the keys. Then, the algorithm handles the possible case where there are 3 keys in a node, by breaking up the current node into a left and right node, whose children and keys are assigned from the current node. The middle key gets inserted into the current node's parent, and then sets the current to the current node's parent. This process is repeated until the current node no longer has 3 keys. The tree grows at the root level, so all leaves are preserved at the bottom of the tree.

`traverseTree` runs in $O(\log n)$ time, because the algorithm traverses depth first into a tree structure. The traversal skips all invalid nodes by only moving to the appropriate child. `handleKeyOverflow` also runs in $O(\log n)$ time, since it traverses the tree from a single leaf up to the root node, which takes the same runtime as traversing from the root node to a single leaf. `breakNode` runs in constant time. Therefore, the insertion of a key into the 3-2 tree structure using an iterative algorithm is $O(\log n)$ time, which matches the expected insertion operation time.

Function insertKey(head, key)

Input: Head node of tree *head*, key to be inserted *key*

```
1 current ← traverseHead(head, key)
2 insert key into current keys
3 handleKeyOverflow(current)
```

Function traverseTree(head, key)

Input: Head node of the tree *head*, key to be inserted *key*
Output: Leaf node where *key* should be inserted

```
1 current ← head
2 while current node is not a leaf do
3 |  if key ≤ current first key then
4 |  |  current ← current left child
5 |  else if key ≥ current first key then
6 |  |  current ← current right child
7 |  else
8 |  |  current ← current middle child
```

Function handleKeyOverflow(current)

Input: Current node in the tree *current*

```
while number of keys in current = 3 do
    middle_key ← middle key of node
    if current parent node is null then
        create new parent node
        current.parent ← to new node
    insert middle_key to parent node
    create new left_node and right_node
    breakNode(current, left_node, right_node)
    delete middle_key from current keys
    left.parent ← current.parent
    right.parent ← current.parent
    current ← current.parent
    delete old current
```

Function breakNode(*current*, out *left*, out *right*)

Input: Current node in tree *current*, new left node *left*, new right node *right*
Output: Constructed *left* and *right* nodes as out parameters

```
left ← parent to current parent
right ← parent to current parent
insert first key from current to left
insert last key from current to right
if current is not a leaf then
    insert two left children from current as left children
    insert two right children from current as right children
```

Discussion on van Emde Boas Trees

Van Emde Boas (vEB) Trees are a data structure which stores a fixed number of $M = 2^m$ binary vectors in an index addressable array. Each index in this array represents the presence (1) or absence (0) of that element in the structure. From this list, it creates and stores a summary vector tree containing binary values identifying the presence of a positive bit within its vector. An example of a vEB tree of order 16 can be seen below.

Unlike other structures, the index addressable nature of the structure vEB trees are buit on allows for location, insertion, and deletion operations to be performed in constant $O(1)$ time simply by changing the bit at the index representing the value in the tree, then updating the summary vector. The major strength of vEB trees over other structures is how it can handle locating the next positive value from any given value in $O(\log \log M)$ time.

An example use case for vEB trees might include the location of the next nearest empty parking space. Assuming that 1) the car is near a parking space that is already taken, 2) that the spaces are ordered in a way that represents distance from one another (i.e. space #3 is closer to space #5 than space #6), and 3) that a 1 will represent an empty parking space. After locating the current spot in constant time, the ability of vEB trees to find a successor or predecessor of an empty spot in $O(\log \log M)$ time might prove useful.

Unfortunately, the limitations of vEB trees make other algorithms (such as Skip Lists) much more preferable for solving successor and predecessor problems. While the constant time insertion and deletion are ideal, the limitation that they may only represent integer values is a major limitation. The fixed size also restricts what the structure can be allowed to represent, since the data may only be dynamic within a certain range. And finally, the fixed space requirement of $O(\mathrm{M})$ makes the structure cumbersome. These drawbacks, combined with the fact that structures like Skip Lists can perform next nearest neighbor functions on more versatile objects in $O(\log n)$ time, makes them more useful over vEB trees.

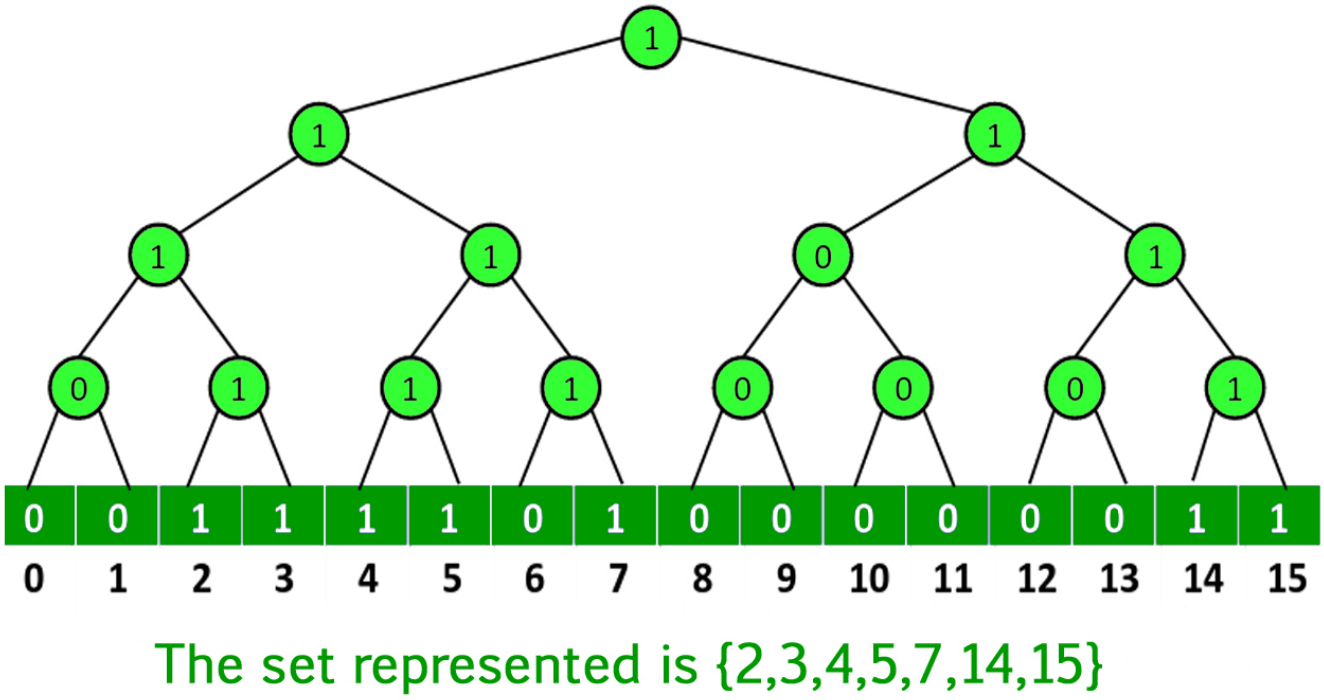

Figure 1: An example proto van Emde Boas Tree from GeeksForGeeks.org

Discussion on Skip Lists

Skip lists are an extension on the standard linked list. This data structure on average preserves the space efficiency of a linked list, but due to the random nature of this structure, could have worst case $O(n\log n)$ space efficiency. Skip lists also have the insertion/deletion ease of a linked list, where each node in the list has a pointer to the next (and sometimes previous) node, while also benefiting from the searching speed of various tree structures. This data structure is a randomized data structure, and it allows an average complexity of $O(\log n)$ for insertion and searching. This means that on average, a skip list has the best features of a sorted array, capable of random memory access while searching, while maintaining the linked list structure, meaning that insertion requires very low time complexity.

A skip list can be thought of as a layered linked list. The bottom layer is denoted as $L(0)$, and then higher layers are denoted as $L(1)$, $L(2)$, $\ldots$, $L(n)$. An element of the skip list has a $\frac{1}{2^m}\%$ chance of being on layer m. If an element appears on layer m, the element also must appear on layer $m-1$, $m-2$, $\ldots$, 0. An image of a skip list can be seen below

The hierarchical structure of the layers results in an average time complexity of $O(\log n)$ for search, insert, and delete operations and a worst case of $O(n)$. This structure uses randomness and a layered linked list to skip over portions of the data when performing search operations, to a very similar effect as a binary search tree. The linked list structure is more beneficial than a binary search tree, however, because there is no need to rebalance the tree after every insertion or deletion with a skip list.

A common real-world representation of a skip list is buses in a busy city. In downtown Seattle, for example, there are many bus stops. Some buses go from stop to stop, whereas some buses stop at one high traffic stop, then cross town, skipping many lower traffic stops in the process. This is the same idea as a skip list, where the bottom row is analogous to the bus that stops at every destination, and the higher up rows in the skip list hierarchy are analogous to the buses that take people from one high traffic destination to another, skipping over some stops.

Skip lists have many applications, especially in high traffic operations where rebalancing a tree or linearly searching through a massive array would take too long. One of the common applications of skip lists in industry is using this data

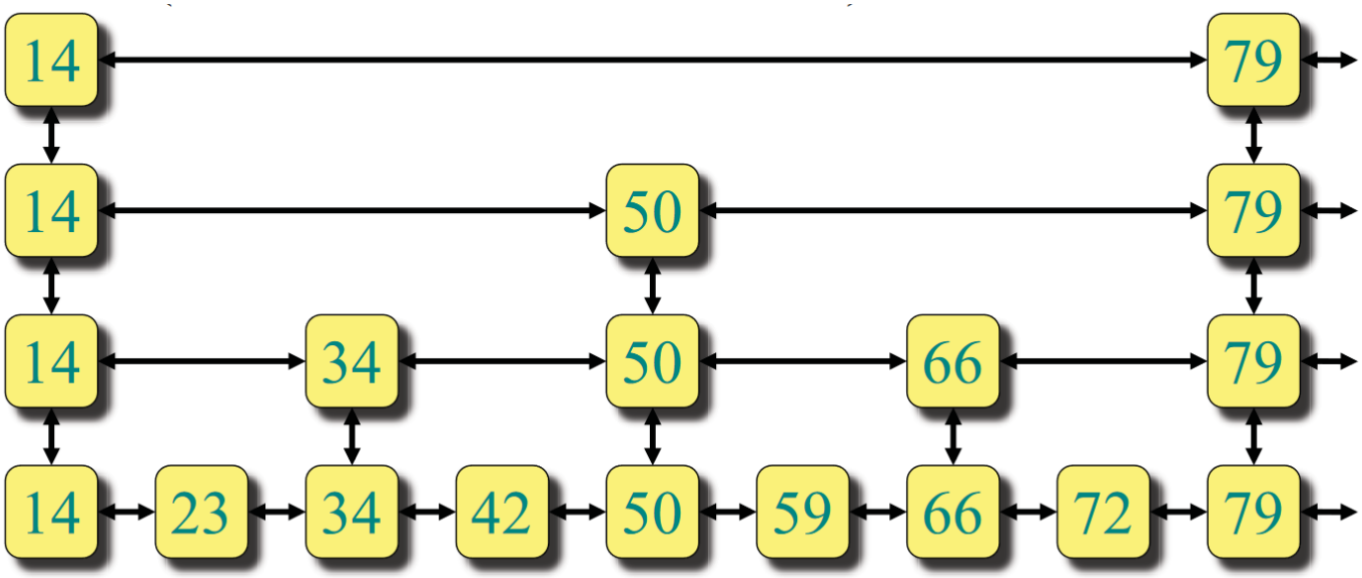

Figure 2: An example of a skip list from the MIT lecture slides

structure as a way to index databases and database tables. Another application of skip lists is in priority queues, where one specific example could be CPU job scheduling. CPU job scheduling is a very time-sensitive process, where jobs could be swapped out many times per second. Maintaining a priority queue with a skip list, where each node in the list is a priority of a job, allows the priority queue to operate in logarithmic time, where insertion and deletion is much faster than a standard queue implementation, and a tree structure doesn't need to be constantly maintained and rebalanced as jobs get added to or removed from the system.