# CS 529: Advanced Data Structures & Algorithms
# Assignment 1

Nathan Chapman, Hunter L., Andrew Struthers

January 10, 2024

## Creating a 3-2 Tree

**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
```

**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)
```

**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)
```

## Discussion on van Emde Boas Trees

Parking lots

## Discussion on Skip Lists

Skip lists are an extension on the standard linked list. This data structure preserves the space efficiency and 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

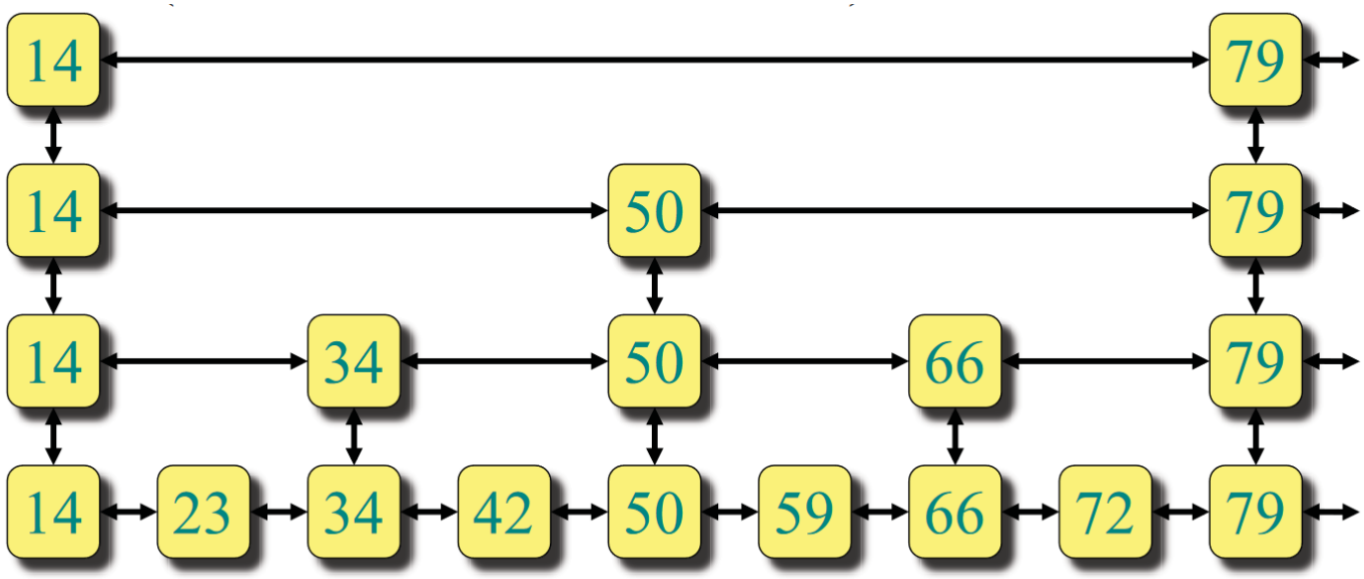


Figure 1: An example of a skip list from the MIT lecture slides

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 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.