CS 240 Module 2 Summary

Angel Zhang

Winter 2022

Heap Properties

- Structural Property
  - All levels are completely filled, except (possibly) for the last level
  - The filled items in the last level are left-justified
- Heap-order Property
  - For any node i, the key of the parent of i is larger than or equal to key of i
  - The maximum is at the root
  - The properties above are for a **max-oriented binary heap**
  - **Min-oriented binary heap** also exists
- Height is $\Theta(\log n)$

Heap Operations

- Insert
  - Place the new key at the first free leaf (after last element of array)
  - Fix-up if necessary
- DeleteMax
  - The maximum is the root
  - Swap the root with the last leaf and delete the last leaf
  - Fix-down if necessary

Heap Runtime

- Insert: $O(\log n)$
- DeleteMax: $O(\log n)$
- FindMax: $O(1)$
- Heapify: $\Theta(n)$
- HeapSort: $O(n \log n)$

CS 240 Module 3 Summary

Angel Zhang

Winter 2022

QuickSelect

Runtime

- Best case
  - First chosen pivot has pivot index k
  - Total cost from partition is $\Theta(n)$
  - No recursive calls
- Worst case
  - Search range decreases by 1 each time
  - $\Theta(n^2)$
- Expected
  - $\Theta(n)$
  - Generally the fastest implementation of a selection algorithm

QuickSort

Main Idea

- Choose pivot and partition
- Recursively sort the left part and the right part

Runtime

- Worst case: $O(n^2)$
- Expected: $O(n \log n)$
- Average: $O(n \log n)$

BucketSort

Main Idea

- Suppose all keys in A are integers in range $[0, \dots, L-1]$
- Use an auxillary bucket array $B[0, \dots, L-1]$ to sort
- Iterate through B and copy non-empty buckets to A
- Time: $\Theta(n)$

Analysis

- Time: $\Theta(L+n)$
- Space: $\Theta(L+n)$

MSD-Radix-Sort

Main Idea

- Sort multi-digit numbers from the most significant digit to the least significant digit
- Sort by the first digit
- Break down in groups by the first digit
- Recursively sort the rest of the digits
- Sort is not stable

Analysis

- Space: $\Theta(n+R+m)$
- Time: $O(mnR)$
  - $O(n)$ if the items are in limited range

LSD-Radix-Sort

Main Idea

- Apply single digit bucket sort from the least significant digit to the most significant digit
- Sort is stable

Analysis

- Time: $\Theta(m(n+R))$
- Space: $\Theta(n+R)$

CS 240 Module 4 Summary

Angel Zhang

Winter 2022

AVL Properties

- **Height-balance property**: The heights of the left and right subtree differ by at most 1
  - the height of an empty tree is defined to be -1
- If node v has left subtree L and right subtree R, then

$$balance(v) = height(R) - height(L) \in \{-1, 0, 1\}$$

- An AVL tree with n nodes has $\Theta(\log n)$ height

AVL Operations

- Insert
  - Insert KVP with the usual BST insertion and update height
  - Restructure if necessary (at most once)
- Delete
  - Delete KVP with the usual BST deletion and update height
  - Restructure if necessary (may need to keep rebalancing up to the root)

AVL Runtime

- Search: $\Theta(height)$
- Insert: $\Theta(height)$
  - Restructure restores the height of the subtree to what it was
  - Restructure at most once
- Delete: $\Theta(height)$
  - Restructure may be called $\Theta(height)$ times (all the way up to the root)
  - Restructure takes constant time so the total cost is still $\Theta(height)$
- Worst-case cost for all operations is $\Theta(height) = \Theta(\log n)$

CS 240 Module 5 Summary

Angel Zhang

Winter 2022

Skip List Properties

- A hierarchy S of ordered linked list $S_0, S_1, \ldots, S_h$
- Special keys $-\infty$ and $+\infty$ (**sentinels**)
- List S_0 contains the KVPs of S in non-decreasing order
- The other lists store only keys, or links to nodes in S_0
- List S_h contains only the sentinels
- The root is the left sentinel in S_h
- Usually has more nodes than keys

Skip List Randomization

- Repeatedly toss a coin until we get a tail
- If i is the number of heads, then i will be the height of tower of k
- $P(\text{tower of } k \text{ has height} \geq i) = \frac{1}{2^i}$

Skip List Analysis

- Expected space: $O(n)$
- Expected height: $O(\log n)$
- Search: $O(\log n)$ expected time
- Insert: $O(\log n)$ expected time
- Delete: $O(\log n)$ expected time

CS 240 Module 6 Summary

Angel Zhang

Winter 2022

Interpolation Search

- Works well if keys are uniformly distributed
- Recurrence relation is $T^{avg}(n) = T^{avg}(\sqrt{n}) + \Theta(1)$
- Resolves to $T^{avg}(n) \in \Theta(\log \log n)$
- Worst case is however $\Theta(n)$
  - Occurs when the keys are not uniformly distributed
- Trick
  - Use interpolation search for $\log n$ steps
  - If key is still not found, switch to binary search
  - Worst case $O(\log n)$, but could be $\Theta(\log \log n)$

Trie

Overview

- A dictionary for bit strings
- Let $|x|$ be the length of a string x
- Search, insert and delete all take $O(|x|)$ time
- Efficient for prefix search

Variation 1: No Leaf Labels

- Do not store actual keys at the leaves since they are stored implicitly along the path to the leaf
- Halves the amount of space

Variation 2: Allow Proper Prefixes

- Allow prefixes
- Internal nodes may now also represent keys, use a flag to indicate such nodes
- Remove $ children and replace them with flags
- More space efficient

Variation 3: Pruned Trie

- Stop adding nodes to trie as soon as the key is unique
- Save space if there are only a few long bitstrings
- Strings need to be stored at leaves
- Most efficient variation in practice

Compressed Tries

Overview

- Each internal node stores an index - next bit to be tested during a search
- Each internal node has at least 2 children
  - n leaf nodes (keys) means at most $n-1$ internal nodes
  - At most $2n-1$ nodes
  - Total space is $\Theta(n)$
- When searching, need to explicitly check whether the string stored at the leaf is x
- All operations are $O(|x|)$

Multiway Tries

Overview

- Represents strings over any fixed alphabet $|\Sigma|$
- Each node has at most $|\Sigma|+1$ children
- Inlcudes one child for the end-of-word character $

Children Storage

- Solution 1: Array of size $|\Sigma|+1$ for each node
  - $O(1)$ to time search child, $O(|\Sigma|n)$ space
- Solution 2: List of children for each node
  - $O(|\Sigma|)$ to time search child, $O(n)$ space
- Solution 3: AVL-tree of children for each node
  - $O(\log|\Sigma|)$ time to search child, $O(n)$ space
  - Best in theory, but not worth it in practice unless $|\Sigma|$ is very large

CS 240 Module 7 Summary

Angel Zhang

Winter 2022

Direct Addressing

- Special situation: any dictionary key k is integer with $0 \leq k < M$
- All operations are $O(1)$
- Total storage is $\Theta(M)$
- Drawbacks:
  - Space is wasteful if $n << M$
  - Keys must be integers

Hashing with Chaining

Worse Case

- In the worst case, all n items hash to the same array index.
  - *Insert*: $O(1)$
  - *Search*: $\Theta(n)$
  - *Delete*: $\Theta(n)$

Runtime

- The **Load factor** is defined as

$$\alpha = \frac{n}{M}$$

  where n is the number of items and M is the size of hash table

- α is the **average bucket size**
  - *Insert*: $\Theta(1)$
  - *Search* and *Delete*: $\Theta(1 + \text{size of bucket } T[h(k)])$
  - Average runtime is NOT $\Theta(1 + \alpha)$
- Under uniform hashing assumption, the expected size of bucket $T[h(k)]$ is at most $1 + \alpha$.
  - Expected runtime of *Insert* is $\Theta(1 + \alpha)$.
  - Expected runtime of *Search* and *Delete* is $\Theta(1 + \alpha)$.
  - Space is $\Theta(M + n) = \Theta(n/\alpha + n)$

Rehashing

- Keep a minimum and a maximum allowed load factor $0 < c_1 < c_2$
- Rehash whenever $\alpha < c_1$ (use smaller M) or $\alpha > c_2$ (user larger M).
- Runtime is $\Theta(M + n)$ but happens rarely

Linear Probing

- Entries tend to cluster into contiguous regions.
- Many probes for each *Search*, *Insert* and *Delete*

Double Hashing

- Double hashing with a good secondary hash function does not cause the bad clustering produced by linear probing.
- *Search*, *Insert*, *Delete* work as in linear probing, but with a different probe sequence.
- Linear probing is a special case of double hashing with $h_2(k) = 1$

Cuckoo Hashing

- Use independent hash functions h_0, h_1 and two tables T_0, T_1.
- An item with key k can only be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$

Operations

- *Insert* starts with T_0 and alternates between T_0 and T_1, kicking out the current occupant until no item is kicked out
- May lead to a loop of "kicking out", maximum $2n$ attempts
- Rehash with a larger M and new hash functions if Insert failed

Runtime

- *Search* and *Delete*: $O(1)$
- *Insert* may be slow, but expected constant time if the load factor is small
- Works well in practice

Notes

- The two hash tables do not have to be of the same size
- Load factor

$$\alpha = \frac{n}{T_0 + T_1}$$

Hashing vs. BSTs

Advantages of BSTs

- $O(\log n)$ worst-case operation cost
- Predictable space usage
- No need to rebuild the entire structure
- Ordered dictionary operations (rank, select, etc.)

Advantages of Hash Tables

- $O(1)$ operations (if hashes well-spread and load factor small)
- Choose space-time trade off via load factor
- Cuckoo hashing achieves worst-case $O(1)$ for *Search* and *Delete*.

CS 240 Module 8 Summary

Angel Zhang

Winter 2022

Quadtrees

Overview

- A set S of n points in the plane
- Bounding box R:
  - Smallest square $[0, 2^k] \times [0, 2^k]$ containing all points
  - Find R by compute the maximum x and y values in S
  - The side length is a power of 2

Range Search

- Query rectangle A
- R is the region associated with current node, 3 cases
  - R does not intersect $A \Rightarrow$ red node, do not search its children
  - R is fully contained in $A \Rightarrow$ green node, report all points inside R
  - R intersects A but is not contained in $A \Rightarrow$ blue node, recursively search its children
  - If R is a leaf and stores a point inside A, then report it

Runtime

- The **spread factor** is defined as
$$\beta(S) = \frac{L}{d_{min}}$$
where L is the side length of R and d_{min} is the smallest distance between two points in S.
- In the worst case, height $h \in \Omega(\log \beta(S))$
- In all cases, height $h \in O(\log \beta(S))$
- Construction: $\Theta(nh)$ worst case
- *RangeSearch*: $\Theta(nh)$ worst case even if the search result is empty
  - Worse than exhaustive search
  - In practice usually much faster

Summary

- Easy to generalize to higher dimensions (octrees, etc)
- Easy to compute
- Simple arithmetic: division by 2 each time (bit shift)
- Space potentially wasteful, but good if points are well-distributed

kd-Tree

Range Search

- Similar to quadtree range search
- Query rectangle A
- R is the region associated with current node, 3 cases
  - R does not intersect $A \Rightarrow$ red node, do not search its children
  - R is fully contained in $A \Rightarrow$ green node, report all points inside R
  - R intersects A but is not contained in $A \Rightarrow$ blue node, recursively search its children
  - If R is a leaf and stores a point inside A, then report it

Runtime

- Construction: $\Theta(n \log n)$
- Height: $O(\log n)$
- *RangeSearch:* $O(s + \sqrt{n})$

Problems

- Do not handle insertion and deletion well
- After insert or delete, split might no longer be at exact median
- Height is no longer guaranteed to be $O(\log n)$

Range Tree

Modified BST Range Search

- Search for k_1 gives left boundary path P_1
- Search for k_2 gives right boundary path P_2
- Find **topmost inside nodes**
  - not in P_1 or P_2
  - left children of nodes in P_2

- right children of nodes in P_1

- Output nodes in the search range
  - Check each node in P_1 and P_2 and report if in range
  - Report all topmost inside nodes and nodes in their subtree

Modified BST Range Search Runtime

- Searching for P_1 and P_2 takes $O(\log n)$
- Checking boundary nodes takes $O(\log n)$
- $O(\log n)$ topmost inside nodes
- Total number of nodes in the subtrees of topmost inside nodes $\leq s$
- Total time $O(s + \log n)$

Range Search in 2D Range Tree

- Perform modified BST Range Search on the primary BST using x-coordinates
- Find boundary and topmost inside nodes
- Check if boundary nodes have valid x-coordinate **and** y-coordinate
- For each topmost inside node, perform range search in associated y-BST using y-coordinates
  - Find boundary and topmost inside nodes
  - Check boundary nodes have valid x-coordinate **and** y-coordinate
  - Report all nodes below topmost inside nodes

Analysis

- Suppose d is the dimension of the range tree
- *RangeSearch*: $O(s + \log^d n)$
- Space: $O(n(\log n)^{d-1})$
- Construction: $O(n(\log n)^{d-1})$

Summary

Quadtree

- Height is $\Theta(\log \beta(S))$
- Range search is $\Theta(nh)$ worst case
- Simple, easy to implement insert and delete
- Works well only if points evenly distributed
- Wastes space if points not evenly distributed

kd-Tree

- Construction is $\Theta(n \log n)$
- Height is $\Theta(\log n)$
- Range search is $O(s + \sqrt{n})$
- Space is $O(n)$
- Insert and delete cause imbalance and affects range search time and there is no simple fix

2D Range Tree

- Construction is $O(n \log n)$
- Range search is $O(s + \log^2 n)$, fastest among three data structures
- Space is $O(n \log n)$
- Waste some space
- Insert and delete cause imbalance but this can be fixed with occasional rebuilt

CS 240 Module 9 Summary

Angel Zhang

Winter 2022

Karp-Rabin

Main Idea

- Use hashing to eliminate guesses faster
- Compute hash function for each guess, compare with pattern hash
  - If values are unequal, then the guess cannot be an occurrence
  - If values are equal, **verify** that pattern actually matches text

Runtime

- Expected runtime is $O(m+n)$, assuming a good hash function - few collisions
- Worst case is $\Theta(mn)$ but is extremely unlikely.

Knuth-Morris-Pratt

Details

- Failure Array
  - $F[j]$ stores the length of the longest prefix of $P[0 \dots j]$ that is a proper suffix of P (or in other words, a suffix of $P[1 \dots j]$)
  - Construction takes $\Theta(m)$
- Starting from the left, match P with T,
- Let i be the index within T and j be the index within P.
  - If match is successful, increment i and j (move forward)
  - If match is unsuccessful at m, if $j > 0$, update j to be $F[j-1]$. Otherwise update i to be $i+1$

Runtime

- Construction of failure array: $\Theta(m)$
- Main KMP function: $\Theta(n)$
- Total runtime: $\Theta(m+n)$

Boyer-Moore

Main Idea

- Fastest matching on English Text
- Reverse-order searching
- When a mismatch occurs choose, the better option among
  - Bad character heuristic
  - Good suffix heuristic

Details

- Last occurrence array
  - Last occurrence of character c in P
  - Define $L(c) = -1$ if character c does not occur in P
  - Bad character heuristic can only be used if $L(c) < j$
  - Computation takes $O(m + |\Sigma|)$
- Upon mismatch, update $j \leftarrow m - 1$
  - If $L(c) < j$,
    $$i \rightarrow i + (m-1) - L(c)$$
    (align the last occurrence of character)
  - If c does not occur in P,
    $$i \rightarrow i + m$$
    (shift past $T[i]$)
  - If $L(c) > j$,
    $$i \rightarrow i + 1 + (m-1) - j$$
    (shift P by 1)
- Formula for i that works in all cases

$$i \leftarrow i + m - 1 - \min\{L(c), j - 1\}$$

Summary

- Performs very well, even when using only bad character heuristic
- Worst case runtime is $O(nm)$ with bad character heuristic only, but in practice much faster
- On typical English text, Boyer-Moore looks only at $\approx 25\%$ of text T

Suffix Tree

Main Idea

- Search for many patterns P within the same fixed text T
- Preprocess the text T rather than pattern P
- Store all suffixes of T in a trie
  - Compressed trie $\Rightarrow O(n)$ space
  - Store suffixes implicitly via indices into T
- P is a substring of T iff P is a prefix of some suffix of T
- When pattern searching, search for P in compressed trie

Runtime

- Construction: $\Theta(n^2|\Sigma|)$
- Pattern matching: $\Theta(m|\Sigma|)$

Summary

- Theoretically good
- Construction is slow or complicated
- Wastes space
- Rarely used in practice

Suffix Arrays

Main Idea

- Store suffixes implicitly (use start indices)
- Store sorting permutation of the suffixes in T
- For pattern matching, apply binary search

Details

- Construction
  - Write out the suffixes of T
  - Sort lexicographically
  - $A^S[j] = i$ such that $T[i \dots n]$ is the suffix in this slot
  - Easy to construct using MSD-Radix-Sort (pad with any character to get the same length)
- Pattern matching
  - Apply binary search

Runtime

- Construction:
  - Worst case $\Theta(n^2)$
  - Can achieve $\Theta(n \log n)$
- Pattern matching: $\Theta(m \log n)$

Summary

- Sacrifice some performance for simplicity
- Slightly slower (by a log-factor) than suffix trees
- Easier to build
- Less space

CS 240 Module 10 Summary

Angel Zhang

Winter 2022

Type of Data Compression

Logical vs. Physical

- **Logical**:
  - Uses the meaning of the data
  - Only applies to certain domains
- **Physical**:
  - Only know physical bits in data
  - Does not know their meaning

Lossy vs. Lossless

- **Lossy Compression**
  - Achieves better compression ratios
  - Decoding is approximate
  - Exact source text S is not recoverable
- **Lossless Compression**
  - Always decodes S exactly

Character Encoding

- **Fixed-length code**: All codewords have the same length
- **Variable-length code**: Codewords can have different lengths

Huffman Encoding

Algorithm

- Determine frequency of each character
- For each character c, create a trie containing c
- Find two tries with minimum weight

- Merge them with new internal node and new weight is the sum
- Repeat last two steps until there is only one trie

Properties

- The constructed trie is not unique
- Two passes (compute frequency and encode)
- Constructed trie is optimal
- Coded text is shortest among all prefix-free character encodings with $\Sigma_C = \{0, 1\}$
- If the frequencies are almost equal, then the compression ratio would not be good.

Run-Length Encoding

Overview

- Variable-length
- Multi-character encoding
- Source and coded alphabet are both binary
- Decoding dictionary is unique

Elias Gamma Coding

- $\lfloor \log k \rfloor$ copies of 0, followed by
- binary representation of k (no leading 0)

Properties

- An all-0 string of length n would be compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits
- No compression until run-length ≥ 6
- Expansion when run-length equals to 2 or 4

Lempel-Ziv-Welch

Overview

- Start with dictionary D_0, usually ASCII, which uses code numbers $0, \ldots, 127$
- Every step adds a multi-character string to the dictionary, using code numbers $128, 129, \ldots$
- Encoding:
  - Current dictionary as a trie
  - Parse trie to find longest prefix w already in dictionary

  - Encode w with one number
  - Add wK where K is the character that follows w already in dictionary
  - Creates one child in the trie at the leaf where we stopped

- Encoded output:
  - A list of numbers
  - Usually converted to bit-string with fixed-width encoding using 12 bits

- Decoding:
  - Build dictionary while reading the coded string
  - One step behind

Summary

- Go through the string only once and do not need to see the whole string
- Compresses quite well on English text

bzip2

Overview

- Uses text transform
- Change input into a different text that is not necessarily shorter, but has other desirable qualities
- Steps:
  1. **Burrows-Wheeler Transform**: Repeated subtrings transform into long runs of characters
  2. **Move-to-Front Transform**: Long runs of characters transform into long runs of zeros and skewed frequencies
  3. **Modified RLE**: Long runs of zeroes mean shorter encoded text. Skewed frequencies remain
  4. **Huffman Encoding**: Compresses well since frequencies are skewed

Move-to-Front Transform

Overview

- Dictionary L is stored as an unsorted array or linked list
- After an element is accessed, move it to the front of the L
- Encode each character of S by its index in L
- After each encoding, update the L with MTF

Properties

- A character in S repeats k times $\iff$ C has a run of $k-1$ zeroes
- C contains many small numbers and a few large ones.
- C has the same length as S, but better properties

Burrows-Wheeler Transform

Overview

- Permute the source text
- The coded text has the exact same letters, but in a different order
- Source and coded alphabets are the same
- If original text had frequently occurring substrings, then transformed text should have many runs of the same character - more suitable for MTF

Details

- Assume that S ends with \$
- A **cyclic shift** of S is the concatenation of $S[i+1 \ldots n-1]$ and $S[0 \ldots i]$, for $0 \leq i < n$.
- The encoded text C consists of the last characters of the cyclic shifts of S after sorting them

Encoding

- Write all consecutive cyclic shifts
- Sort cyclic shifts lexicographically
  - \$ is lexicographically smaller than other characters
- Extract last characters from sorted shifts, i.e. last column
- Runtime
  - For sorting, letters after \$ do not matter
  - Same as sorting suffixes of the soure text, use MSD-Radix-Sort
  - $O(n \log n)$ for sorting
  - Read coded text from suffix array in $O(n)$ time

Decoding

- Given C, the last column of sorted shifts array
- Number the rows for the last column (from 0 to $m-1$)
- Reconstruct the first column by sorting C
- Number the rows for the first column (equal characters stay in the same order)
- Recover source text, starting from the \$ in the last column to recover $S[0]$
- Shift $S[0]$ back and repeat the process

Summary

- Encoding
  - $O(n \log n)$ with special sorting algorithm
  - MSD is good in practice but worst case is $\Theta(n^2)$
  - Read encoding from the suffix array
- Decoding
  - $O(n + |\Sigma_S|)$
  - Faster than encoding
- Encoding and decoding both use $O(n)$ space
- Tends to be slower than other methods
- Combined with MTF, modified RLE and Huffman leads to better compression

CS 240 Module 11 Summary

Angel Zhang

Winter 2022

Sorting in External Memory

- HeapSort
  - Poor memory locality
  - Access indices that are far apart (children's indices doubled from parent's indices's)
  - Typically one block transfer per array access
  - Does not adapt well to external memory model
- MergeSort
  - Access consecutive locations
  - Can read in blocks
  - Ideal for external memory model

MergeSort in External Memory

- $\frac{n}{B}$ block transfers for input streams and $\frac{n}{B}$ for output stream, total $\frac{2n}{B}$
- $\log_2 n$ rounds
- Total number of **block transfers** is

$$\frac{2n}{B} \cdot \log_2 n \in \Theta\left(\frac{n}{B} \log n\right)$$

d-way Merge Runtime

- Internal:
  - Priority queue P with size d, implemented with a min-heap
  - One $insert()$ and $deleteMin()$ for each item
  - Total cost is $\Theta(n \log d)$
- External:
  - Assuming $d + 1$ blocks and P fits into memory
  - $\frac{2n}{B} \in \Theta(\frac{n}{B})$ block transfers

d-way MergeSort Runtime

- Internal:
  - $\log_d$ rounds of merging
  - Each round takes $\Theta(n \log_2 d)$
  - Total cost is

$$\Theta(\log_d n \cdot n \log_2 d) = \Theta\left(\frac{\log_2 n}{\log_2 d} \cdot n \log_2 d\right) = \Theta(n \log n)$$

  - In internal memory, d-way MergeSort has the same running time theoretically
  - In practice d-way is slower for maintaining a heap
- External with no improvement:
  - $\log_d n$ rounds of merging
  - Each round takes $\Theta(\frac{n}{B})$ block transfers
  - Total cost is $\Theta(\frac{n}{B} \cdot \log_d n)$
- External with improvement:
  - Sorting $\frac{n}{M}$ sorted runs
  - $\log_d \frac{n}{M}$ rounds of merging
  - Each round takes $\Theta(\frac{n}{B})$ block transfers
  - Total number of **block transfers** is

$$\Theta\left(\frac{n}{B} \cdot \log_d \frac{n}{M}\right) = \Theta\left(\frac{n}{B} \cdot \log_{M/B} \frac{n}{M}\right)$$

    since $d \approx \frac{M}{B} - 1$
  - This is also a **lower bound** in external memory model for comparison-based sorting

2-4 Trees

Properties

- Every node is either
  - **1-node**: one KVP and two subtrees
  - **2-node**: two KVP and three subtrees
  - **3-node**: three KVP and four subtrees
- The keys at a node are between the keys in the subtrees.
- All empty subtrees are at the same level (important to ensure small height)
- Height is $O(\log n)$

Operations

- Insert
  - Overflow resolved by **node splitting**
  - Take the middle key (the right key if there are two middle keys) and move it up a level, then split the keys on two sides
  - Might need to split until reaching the root
- Delete
  - Underflow resolved by **rotate/transfer** or **merge**
  - If a rich immediate sibling exists, then rotate/transfer (similar to an AVL tree)
  - Otherwise merge (bring parent node down)

a-b Trees

Properties

- Each node has at least a subtrees, unless it is the root
- Each node has at most b subtrees
- The root has at least 2 subtrees
- A node has d subtrees $\iff$ it stores $d-1$ KVPs
- Empty subtrees are at the same level
- The keys in the node are between the keys in the corresponding subtrees
- Requirement: $a \leq \left\lceil \frac{b}{2} \right\rceil$
- Smallest number of KVP: $1 + 2a^h$
- Height: $O(\log_a n) = O(\log n / \log a)$

Operations

- Insert
  - Overflow resolved by **node splitting**
  - Take the middle key (the right key if there are two middle keys) and move it up a level, then split the keys on two sides
- Delete
  - Underflow resolved by **rotate/transfer** or **merge**
  - If a rich immediate sibling exists, then rotate/transfer (similar to an AVL tree)
  - Otherwise merge (bring parent node down)

Runtime

- Height is $O(\log n / \log a)$
- Choose $a = \lceil b/2 \rceil$ to minimize the height
- Work at node can be done in $O(\log b)$ time
- Total cost: $O(\log n)$

B-Tree

Properties

- An a-b tree tailored to the external memory model
- Every node is one block of memory (of size B)
- b is chosen maximally such that a node with $b - 1$ KVPs fits into a block of memory
- b is called the **order** of the B tree. Typically $b \in \Theta(B)$
- a is set to $\lceil b/2 \rceil$ to minimize height

Runtime

- All operations require visiting $\Theta(height)$ nodes
- Work within a node is done in the internal memory so no block transfer
- The height is $\Theta(\log_a n) = \Theta(\log_B n)$
- Therefore all operations require $\Theta(\log_B n)$ block transfers