B-trees

What they are

How they are implemented

B-trees

Self-balancing search trees

Fast search, insertion, deletion

Widely used for databases and file systems

Introduced by Bayer & McCreight (1970)  

What does B stand for? Balanced? Bushy? Boeing?

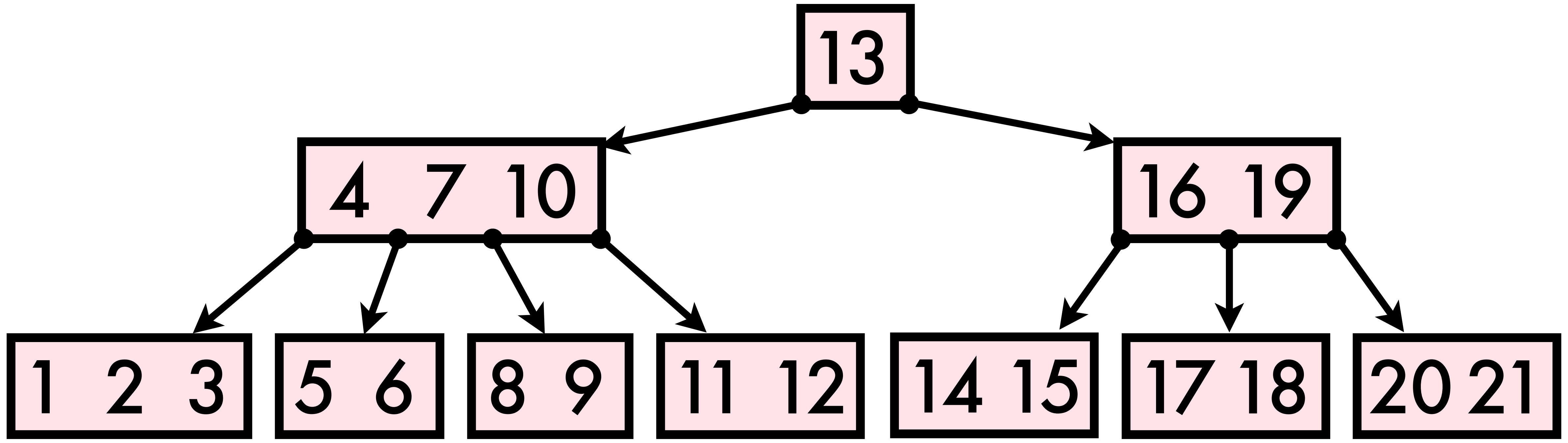
"The more you think about what the B could mean, the more you learn about B-trees" (Bayer)

B-tree complexity (for n data items)

Worst case: search, insert, delete $\rightarrow O(\log n)$

Storage of B-trees: $\Theta(n)$

B-trees: Balanced search trees where nodes can have many children (e.g. thousands)



Higher branching factor $\Longrightarrow$ Reduced tree height

$\Longrightarrow$ Fewer disk accesses

Example applications: MySQL ApFS btrfs

References/Notes/Image credits:
R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", ACM SIGFIDET (1970)
(R. Bayer) https://www.computerhope.com/people/rudolf_bayer.htm
(E. McCreight photo and discussion of naming) https://www.mccreight.com/people/ed_mcc/index.htm
(B-tree use in MySQL) https://www.vertabelo.com/blog/all-about-indexes-part-2-mysql-index-structure-and-performance/
(B-tree use in ApFS) https://www.ntfs.com/apfs-structure.htm
(B-tree use in btfs) https://en.wikipedia.org/wiki/Btrfs

Precursor: Memory Hierarchy/External Memory

Computer memory is arranged as a hierarchy

CPU

Registers

Cache

Main memory

External memory

Faster ↑ | Larger ↓

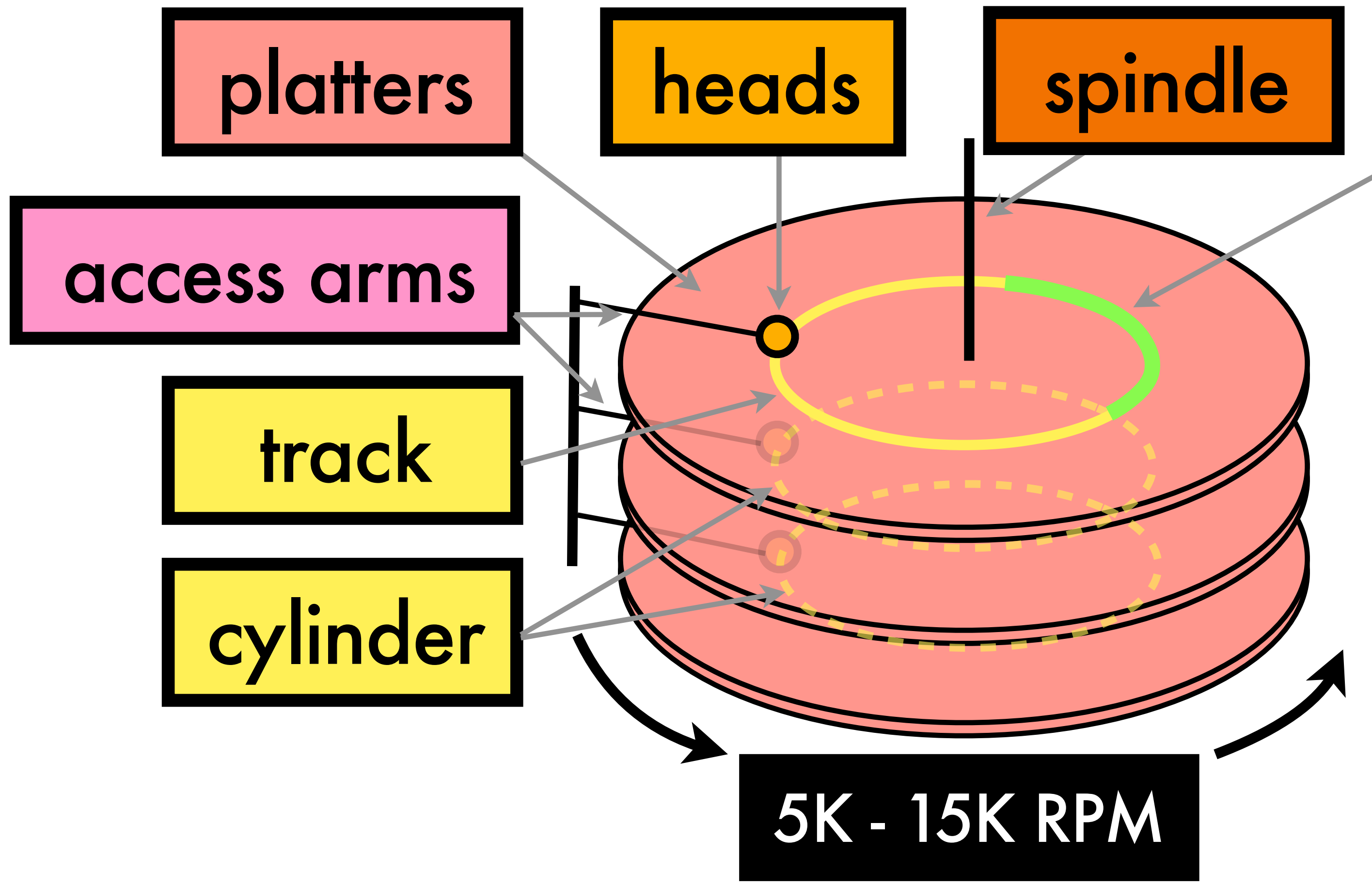
For many problems, we care about **two levels**:

- The level that can store all items in the problem
- The level above this — Transfer is often the bottleneck

B-trees are well-suited to addressing this challenge

Focus on External memory↔Main memory

Hard Disk Drive (introduced in 1956)



seeking is slow due to arm moves (high latency)

reading/writing neighbours in block is (relatively) fast

SSDs: lower latency than HDDs, but still higher than main memory (both SATA SSD & NVMe SSD)

SSDs also use blocks for data access

References:
(Memory hierarchy) M. T. Goodrich et al., "Algorithm design and applications", Chap. 20 (2015)
(Introduction of HDD in 1956) https://www.ibm.com/ibm/history/exhibits/storage/storage_350.html
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap. 5.4.9 (1998)
T. Cormen et al., "Introduction to algorithms", Chap. 18, MIT press (2022)

B-trees and Counting Disk Accesses

A key idea for B-trees:

Make number of children as large as possible while ensuring each node fits in a single block

Navigating down a shallow-but-wide B-tree then involves very few disk accesses

Example: Suppose we have 200 children (199 keys) at each internal node

A (full) B-tree with depth 3 will contain $1 + 200 + 200^2 + 200^3 = 8040201$ nodes

If we keep the root node in memory, we can access $\approx 1.6\text{B}$ keys with just three disk accesses!

Counting disk accesses

Reading/writing blocks from disk is expensive, so we track both: CPU time | Disk block read/writes

To access an object u that is not in memory, we must read the block that contains it `read_block(u)`

To store changes to u, we need to write its block to disk `write_block(u)`

References:
M. T. Goodrich et al., "Algorithm design and applications", Chap. 20 (2015)
T. Cormen et al., "Introduction to algorithms", Chap. 18, MIT press (2022)
Note: A similar cost model for counting disk accesses (based on page accesses) is described in detail by R. Sedgewick et al., "Algorithms", 4th Ed. (2011)

B-tree Definition

B-tree properties (based on CLRS)

A B-tree is a tree with minimum degree, t:

- Node u has attributes:

u.keys | list | (ascending order) | u.is_leaf

- Internal node u has len(keys) $+1$ children

u.children | list of length len(keys) + 1

- The keys of node u separate its children's keys

u.keys[i] $\leq$ u.children[i+1].keys[j] $\leq$ u.keys[i+1] | $\forall$ valid j

u.children[0].keys[j] $\leq$ u.keys[0] | u.keys[-1] $\leq$ u.children[-1].keys[j]

- All leaves have the same depth
- All nodes (except root) have $\geq t-1$ keys
- All nodes have $\leq 2t-1$ keys

When $t=2$, the B-tree is called a 2-4 tree or 2-3-4 tree

Warning: there are many different notation/ definition conventions for B-trees!

Bayer & McCreight | Knuth (TAOCP) | CLRS

The height of an n-key B-tree grows $\Theta(\log n)$

Num. nodes in a max height ("skinny") tree

$$= \underbrace{1}_{\text{root}} + 2 + 2t + 2t^2 + \ldots \;= 1 + 2\left(\frac{t^h - 1}{t-1}\right)$$

Num. keys $n = 1 + \cancel{(t-1)} \cdot 2\left(\frac{t^h-1}{\cancel{t-1}}\right) = 2t^h - 1$

$$\implies h_{\max} = \left\lfloor \log_t\left(\frac{n+1}{2}\right)\right\rfloor$$

Floor $\lfloor \cdot \rfloor$ for other n values

Note: base t in the log makes B-trees short!

If t fixed, use $\Theta(\log n)$ not $\Theta(\log_t n)$ (base change is constant factor)

References:
R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", ACM SIGFIDET (1970)
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap. 6.2.4 (1998)
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.1, MIT press (2022)
(2-3-4 trees) https://en.wikipedia.org/wiki/2-3-4_tree
M. T. Goodrich et al., "Algorithm design and applications", Chap. 20.2 (2015)

B-tree Search

Python B-tree search procedure (recursive):

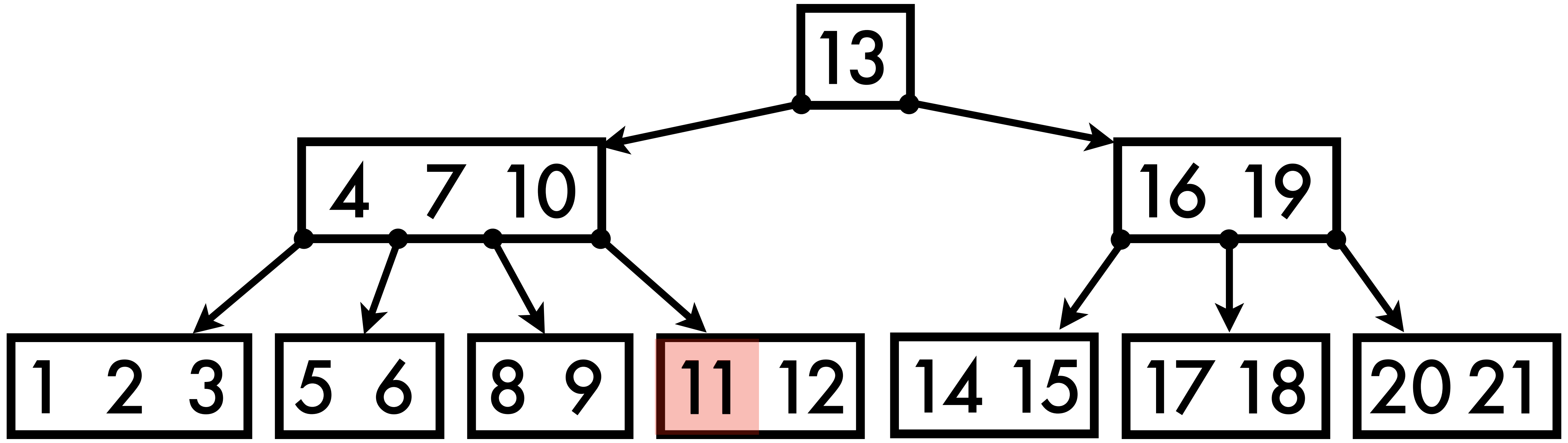
```python
def search(self, u, key): # u is a node
    # linear scan to find index of key
    i = 0
    while i < len(u.keys) and key > u.keys[i]:
        i += 1
    if i < len(u.keys) and key == u.keys[i]:
        return (u, i)
    if u.is_leaf:
        return None
    read_block(u.children[i])
    return self.search(u.children[i], key)
```

Arguments: (root, 11)

Returns (node, 0)

Could replace linear scan with binary search

(not always useful due to caching effects)



Example B-tree

Search complexity

Consider costs with min. degree, t and num. keys, n

We've seen that tree height is $O(\log_t n)$ for n keys

CPU Linear scan $O(t)$ per node, $O(t \log_t n)$ total

(or with binary search, $O(\log_2 t \log_t n) = O(\log_2 n)$)

Disk block reads $O(\log_t n)$

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.2, MIT press (2022)
(Current Linux B-+-tree - uses linear scans) https://github.com/torvalds/linux/blob/7f317d34906c1033f0752fc137dda04e43979bb8/include/linux/btree.h

B-tree Insertion

Overview of strategy

Idea: search for leaf node and insert key

What if that node is already full?

Split full node into two nodes at median key:

Keys to left of median key go to the first

Keys to right of median key go to the second

Move median key up into parent

What if the parent is already full...?

Two strategies for B-tree insertion:

1. "**Insert-then-fix**" (Bayer & McCreight)

Insert at leaf, then reverse up tree and fix

2. "**Fix-then-insert**" (CLRS) Split full nodes on the way down, then insert at leaf

Benefit: "1 pass"

Insert complexity: CPU $O(t \log_t n)$ Disk $O(\log_t n)$

References:

R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", ACM SIGFIDET (1970)

(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.2, MIT press (2022)

L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

B-tree Deletion

Overview of strategy

Idea: search for node and delete key

What if that node becomes too small?

"Fix-then-delete" - only (recursively) call delete on nodes with $\geq t$ keys (safe to delete 1)

1 pass

This means we may need to transfer a key down into a child before calling delete

OK since we ensure current node has t keys!

There are 3 cases to handle - when search:

1. Reaches leaf node
2. Reaches internal node containing target key
3. Reaches internal node without target key

Implement B-tree deletion in Python

2/2 Here's an implementation of B-tree deletion in Python:

python Copy code

```python
class BTreeNode:
    def __init__(self, keys=[], children=[], is_leaf=True, max_num_keys=5):
        self.keys = keys
        self.children = children
        self.is_leaf = is_leaf
        if max_num_keys < 3:  # max_num_keys must be odd and greater or equa
            max_num_keys = 3
        if max_num_keys % 2 == 0:  # max_num_keys must be odd
            max_num_keys += 1
        self.max_num_keys = max_num_keys

    def is_full(self):
        return len(self.keys) >= self.max_num_keys

class BTree:
    def __init__(self, max_num_keys=5):
        self.max_num_keys = max_num_keys
        self.root = BTreeNode(max_num_keys=max_num_keys)

    def find_leaf(self, key, node=None):
        if node is None:
            node = self.root
        if node.is_leaf:
            return node
        for i in range(len(node.keys)):
            if key < node.keys[i]:
                return self.find_leaf(key, node.children[i])
        return self.find_leaf(key, node.children[len(node.keys)])

    def remove(self, key):
        leaf = self.find_leaf(key)
        if key in leaf.keys:
            self.remove_from_leaf(leaf, key)

    def remove_from_leaf(self, leaf, key):
        index = leaf.keys.index(key)
        del leaf.keys[index]
        if len(leaf.keys) < self.max_num_keys // 2:
            self.repair_leaf(leaf)

    def repair_leaf(self, leaf):
        if self.root == leaf:
            if not leaf.keys:
                self.root = None
            return
        parent = self.find_parent(leaf)
        index = parent.children.index(leaf)
        if index > 0 and len(parent.children[index - 1].keys) > self.max_num
            self.borrow_from_prev_leaf(leaf, parent, index)
        elif index < len(parent.keys) and len(parent.children[index + 1].key
            self.borrow_from_next_leaf(leaf, parent, index)
        else:
            if index > 0:
                self.merge_with_prev_leaf(leaf, parent, index)
            else:
                self.merge_with_next_leaf(leaf, parent, index)

    def borrow_from_prev_leaf(self, leaf, parent, index):
        prev_leaf = parent.children[index - 1]
        key = prev_leaf.keys[-1]
        prev_leaf.keys = prev_leaf.keys[:-1]
        leaf.keys = [key] + leaf.keys

    def borrow_from_next_leaf(
```

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)

L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)

B-tree Deletion - Complexity

Deletion complexity (one pass)

Successor/predecessor calls followed by function termination (still "one pass")

Tree height is $O(\log_t n)$ for n keys:

CPU Linear scan $O(t)$ per node, $O(t \log_t n)$ total

Disk block reads/writes $O(\log_t n)$

Note: in practice, most deleted keys are in the leaves (for large values of t)

Other B-tree variants we did not discuss:

B+-tree - all values stored in leaves (not internal nodes) which are linked

B*-tree - aims to keep non-root nodes "more full" (at least 2/3)

References:
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap. 18.3, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 7 (2022)
D. Comer, "The Ubiquitous B-tree", ACM Computing Surveys (1979)
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap. 6.2.4 (1998)

Hash tables

What they are | How they are implemented

Hash tables

Data structures for fast search, insertion & deletion

Introduced by H. P. Luhn at IBM (1953) 

Complexity (for n data items)

In typical conditions, hash tables ops are fast:

Avg. case: search, insert, delete $\rightarrow O(1)$ key benefit

Worst case: search, insert, delete $\rightarrow \Theta(n)$

Storage complexity of hash tables: $\Theta(n)$

Suited for **Abstract Data Types** | Set | Map

References/Notes/Image credits:
H. Stevens, "Hans Peter Luhn and the birth of the hashing algorithm", IEEE spectrum (2018)
(Luhn photo) https://researcher.watson.ibm.com/researcher/view_page.php?id=6990
"Associative Array"/"Dictionary" can replace "Map" https://en.wikipedia.org/wiki/Associative_array

Idea: search by index

Idea: replace search with array indexing $O(1)$

Direct-address table

Suppose the n objects $x_0, \ldots, x_{n-1}$ we'll store

have unique integer keys $k_0, \ldots, k_{n-1}$

$k_i \in \{0, \ldots, m-1\}$ Universe, U, is set of possible keys

$U = \{0, \ldots, m-1\}$

Build a big array with m slots:

				x_1									
0	1	2	3	4	$\cdots$								$m-1$

Unused slots have a value of None:

Example operations on x_0 $(k_0 = 2)$ x_1 $(k_1 = 4)$

Insert x_0, x_1 | delete x_0 | search $k = 4$ → x_1

We search, insert, delete using array in $O(1)$

What happens if $|U| \gg n$? Lots of wasted space!

Suppose we want to store 5 IPv6 addresses

Our universe size is $|U| = 2^{128}$

$>$ 1K trillion trillion 1TB hard drives! $> 28 \cdot 10^{27}$ GBP

A hash table uses a function, h, to compute slots

$h : U \rightarrow \{0, \ldots, m-1\}$ is a hash function

Goal: design h to shrink array size $(m = \Theta(n))$

Direct-address table: $x_i \rightarrow k_i \rightarrow$ slot k_i

big array $m = |U|$

						x_i		

Hash table: $x_i \rightarrow k_i \rightarrow$ hash function, h $\rightarrow$ slot $h(k_i)$

small array $m = \Theta(n)$

			x_i	

References/Notes:
(search by location lookup) D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 6.4, (1974)
(Direct-address table) T. Cormen et al., "Introduction to algorithms", Chap 11.1, MIT press, (2022)
IPv6 address example borrowed from M. Levin, "Data Structures", Coursera (2022)
Price: £28 estimate for1TB WD Seagate HGST HP 3.5" SATA Internal Hard Drive HDD PC CCTV, (ebay, Sep 2022)

Hash functions

"Input data is not random! So good hash functions must be random!" J. Erickson

Suppose $U \subset \mathbb{Z}$ and our hash table has m slots

A basic hash function: $h(k) = k \bmod m$

$m = 5$ | | | x_0 | x_2 | | collision!

0 1 2 3 4

x_0 $(k_0 = 2)$ x_1 $(k_1 = 8)$ x_2 $(k_2 = 23)$

Two key requirements for our hash function:

1. Fast to compute
2. Minimise collisions $h(k_i) = h(k_j)$ with $k_i \neq k_j$

Ideal $h(k)$ rolls a fair m-sided die for each k:

an *independent uniform random hash function*

How to get randomness from nonrandom data?

Division method — Static

$h(k) = k \bmod m$

helps (a bit) if m is prime

Multiplication method

Choose $A \in (0,1)$

$h(k) = \lfloor m \cdot (Ak \bmod 1) \rfloor$

vulnerable to unfavourable key distributions (many collisions)

Random

Universal family H:

$P_{h \in H}[h(k_i) = h(k_j)] \leq \frac{1}{m} \quad \forall i \neq j$

A universal family

Pick a prime number $p > |U|$

$h_{a,b}(x) \triangleq ((ax + b) \bmod p) \bmod m$

$H_{p,m} = \{h_{a,b} \,|\, a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$

(a, b are "salts")

less vulnerable

Cryptographic

Pre-image resistance

Collision resistance

(typically slower)

Applications

Hash tables | String search | Passwords

Signatures | Digests | Proof-of-work

References/Notes/Image credits:

(Requirements/randomness) D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 6.4 (1998)

(Hash functions) T. Cormen et al., "Introduction to algorithms", Chap 11.3, MIT press, (2022)

J. Erickson, "Algorithms" http://algorithms.wtf/ "Lecture 5: Hash Tables" (2019)

J. L. Carter et al., "Universal classes of hash functions", ACM STOC (1977)

Following T. Cormen et al., "Introduction to algorithms", Chap 11.3, MIT press, (2022), we use the notation that $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$

(Bitcoin logo) https://commons.wikimedia.org/wiki/File:Bitcoin_logo.svg#/media/File:Bitcoin.svg

Chaining

Chaining: a simple way to handle collisions

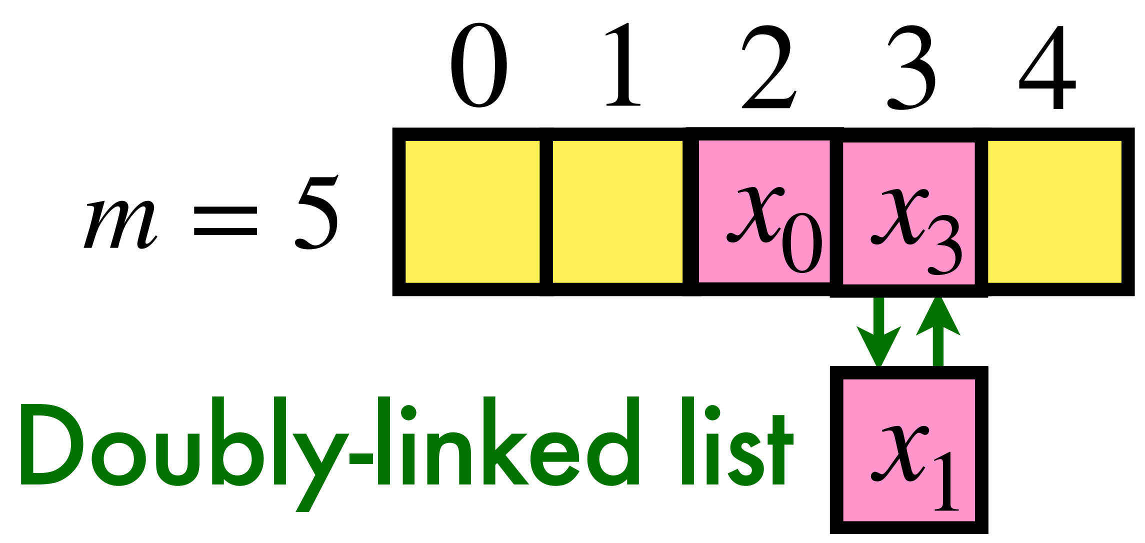
Insert

x_0 $(k_0 = 2)$

x_1 $(k_1 = 8)$

x_2 $(k_2 = 23)$

x_3 $(k_3 = 98)$



$h(k) = k \bmod m$

Search for $(k = 8)$

Delete x_2

<u>Worst case scenario</u> (for search)

All n keys collide $\implies$ all objects in same slot

Search is then $\Theta(n)$ with linked lists

($\Theta(\log n)$ if lists are ordered for binary search)

<u>Average scenario</u> (cost of unsuccessful search)

Define the *load factor* of table: $\alpha \triangleq \frac{n}{m}$ $\frac{\boxed{\text{items}}}{\boxed{\text{slots}}}$

Assume our hash function is universal

Collision probability $\leq 1/m$

$\mathbb{E}(\text{chain length}) = n/m = \alpha$

Average cost: $\Theta(1 + \alpha)$ (hashing + chain search)

<u>Average cost of successful search</u>

Similarly to unsuccessful search: $\Theta(1 + \alpha)$

References/Notes:
See J. Erickson, "Algorithms" http://algorithms.wtf/ "Lecture 5: Hash Tables" (2019) for a more detailed proof
or T. Cormen et al., "Introduction to algorithms" MIT press, Chap 11.2 (2022) for an extended analysis

Open addressing

Open addressing: chain-free collision handling

Coined by William W. Peterson in 1957 

The simplest variant is linear probing:

Insert

- x_0 $(k_0 = 2)$
- x_1 $(k_1 = 8)$
- x_2 $(k_2 = 23)$
- x_3 $(k_3 = 98)$

$m = 5$

0	1	2	3	4
x_3		x_0	x_1	D

primary clustering

$h(k) = k \bmod m$

Search for $(k = 98)$

Delete x_2

Open addressing schemes

Probe sequences

produce permutation of $(0,1,\ldots,m-1)$

position in probe sequence

Double hashing: $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$

For a permutation, $h_2(k)$ and m must be coprime

<u>Analysis:</u> number of probes in unsuccessful search $(\alpha < 1)$

Assume independent uniform permutation hashing

Max probes: $\dfrac{1}{1-\alpha} = \underbrace{1}_{\text{at least one}} + \overbrace{\alpha}^{\text{more than 1}} + \underbrace{\alpha^2}_{\text{more than 2}} + \overbrace{\alpha^3}^{\text{more than 3}} + \ldots$

References/Notes/Image credits:
D. E. Knuth, "The art of computer programming, vol. 3: sorting and searching", Chap 6.4, (1974)
W. W. Peterson, "Addressing for random-access storage." IBM journal of Research and Development (1957)
https://en.wikipedia.org/wiki/W._Wesley_Peterson#/media/File:W._Wesley_Peterson.jpg
(Open addressing) T. Cormen et al., "Introduction to algorithms", Chap 11.4, MIT press, (2022)