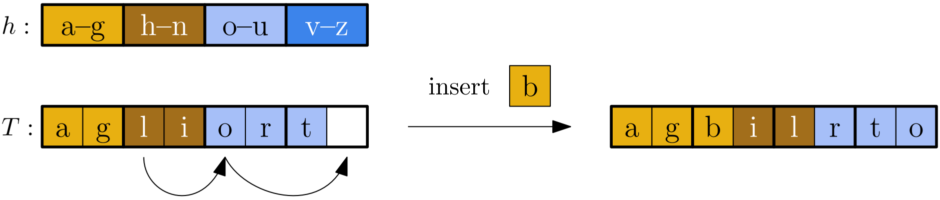
Karlsruhe Institute of Technology

Sliding Block Hashing (Slick)

ITI AG Sanders



(Closed) Hash Tables

Map set S of n elements to m cells of a table $T[0..m-1]$.

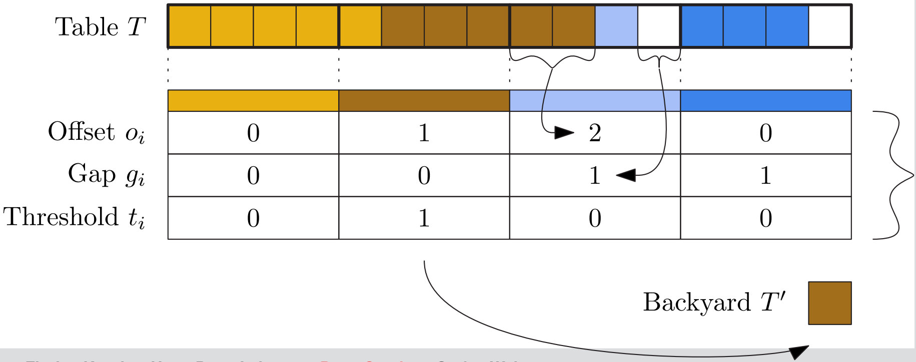
Example: Linear Probing, $S = \{a, l, g, o, r, i, t, h, m\}$

h:												
	a	b	c	d	e	f	g	h	i	j	k	l
	m	n	o	p	q	r	s	t	u	v	w	x
	a	m	o	⊥	⊥	r	g	t	i	h	⊥	l

Sliding Block Hashing with Bumping

Partition T into blocks of size B.
h hashes each element x to a block $h(x)$.
Invariant: $x \in T[\underbrace{iB + o_i..(i+1)B + o_{i+1} - g_i - 1}_{\text{blockRange}(i)}]$ or x is bumped

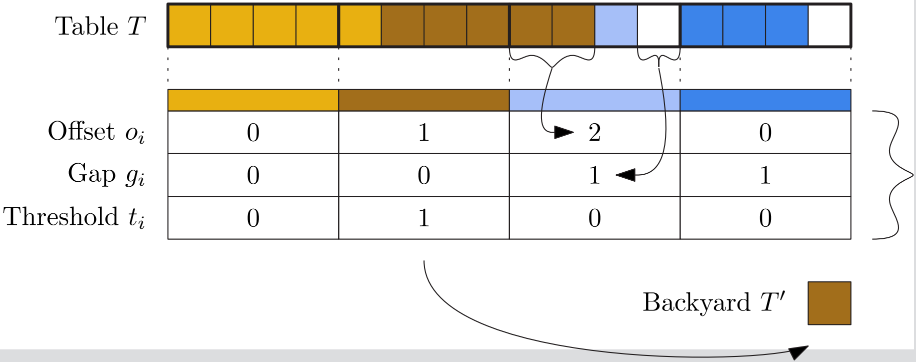
Store bumped elements in backyard T'



Simple and Compact Bumping

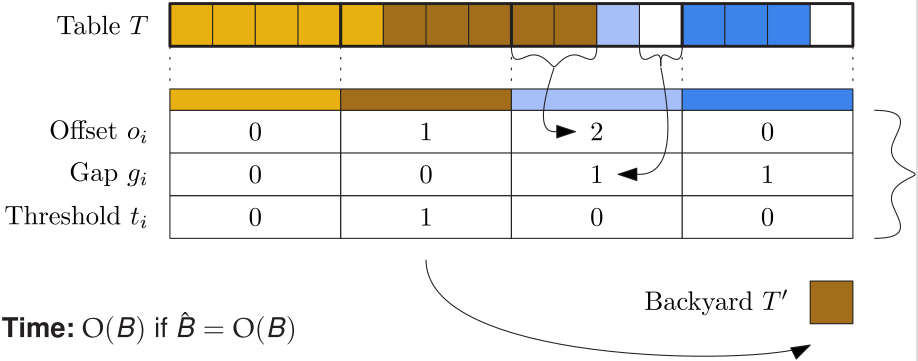
$\delta(x) < t_i \Rightarrow$ bump x to backyard T'
Set $t_i \in 0..\hat{t}$ to ensure:

- $o_i \in 0..\hat{o}$
- at most $\hat{B}$ elements per block
- no table overflow to the right



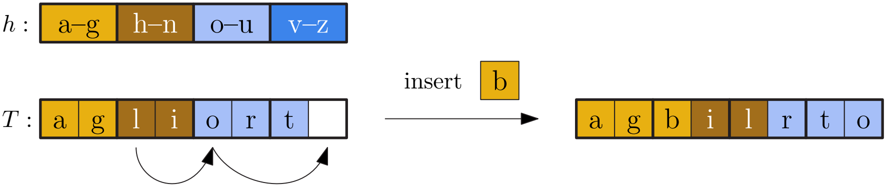
Search

$i := h(x)$
If $\delta(x) < t_i$ **then return** T'.search(x)
search x in T[blockRange(i)]



Time: $\mathrm{O}(B)$ if $\hat{B} = \mathrm{O}(B)$

Insert



Move just one element per block.
May be impossible or too expensive
$\leadsto$ bump sth near block $h(x)$.
Time: $\mathrm{O}(B) + T_{\text{bumpedCase}}$ if $\mathrm{O}(B)$ blocks allowed to slide

 Florian Kurpicz, Hans-Peter Lehmann, Peter Sanders, Stefan Walzer

Delete

Procedure delete(k: K)
    $i := h(k)$
    if $\delta(k) < t_i$ **then** T'.delete(k);   **return** -- bumped
    if $\exists j \in \text{blockRange}(i) : \text{key}(T[j]) = k$ **then** -- found
        $T[j] := T[\text{blockEnd}(i)]$ -- overwrite deleted element
        g_i++ -- extend gap

Build(S)

Simplification: no bumping, unbounded o_i and overflow area

sort S lexicographycally by $h(e)$
$o{:=}\,0$
foreach block i with elements $b = \{b_1, \ldots, b_k\} \subseteq S$ **do**
    $o_i{:=}\,o$
    store b in $t[iB + o..iB + o + k - 1]$
    $o{:=}\,o + k - B$
    if $o < 0$ **then** $g_i{:=}\,-o$; $o{:=}\,0$ **else** $g_i{:=}\,0$

Build(S)

Outline of general case.
When $o > \hat{o}$: bump something (set thresholds appropriately)
Similarly bump at end of table or when a block is too large.
Recurse on bumped elements.

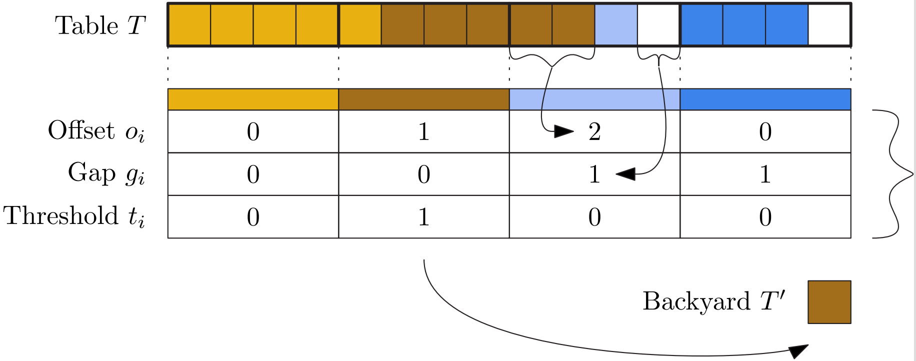
```
Procedure greedyBuild(S: Sequence of E)
    bumped:= ⟨⟩
    sort S lexicographycally by (h(e), δ(e))
    o:= 0                                                      -- offset
    for i := 0 to m/B − 1 do                                   -- for each block
        b:= ⟨e ∈ S : h(key(e)) = i⟩                            -- extract block b_i from S
        t:= 0                                                  -- threshold for b_i
        excess:= max(|b| − B̂, o + iB + |b| − m, o + |b| − B − ô)
                     [|b|≤B̂]    [|T|=m]          [o_{i+1}≤ô]
        if excess > 0 then
            for j := 1 to excess do  bumped.pushBack(b.popFront)
            t:= δ(bumped.last) + 1                              -- adapt threshold
        while |b| > 0 ∧ δ(b.front) < t do
            bumped.pushBack(b.popFront)
        M[i]:= (o, max(0, B − o − |b|), t)                      -- write metadata for b_i
        for j := 0 to |b| − 1 do  T[iB + o + j]:= b[j]          -- write b_i to T
        o:= max(0, o + |b| − B)                                 -- next offset
    M[m/B]:= (0, 0, 0)                                         -- sentinel metadata
    T'.build(bumped)
```

Space Consumption

Proposition:

Only $me^{-\Omega(B)}$ empty cells achievable with appropriate overload $\alpha = \frac{n}{m} > 1$.

Just $O\left(\frac{m}{B} \log B\right)$ bits of metadata



Representing Metadata

Tradeoff: Space versus Time.

Space efficient representation:

- Encode triple $M_i = (o_i, g_i, t_i)$ in a single K-bit code word.
- Only one code word for case $g_i > 0$. In that case encode actual M_i in an empty cell.
  All other code words imply $g_i = 0$.
- Case of k-bit thresholds: $2^k + 1$ values for t_i.
  Choose $\hat{o} = 2^k - 2$, i.e., $2^k - 1$ values for t_i.
  $\Rightarrow (2^k + 1) \times (2^k - 1) + 1 = 2^{2k}$ code words needed – $2k$ bits.
  For example 4 bit thresholds and $\hat{o} = 14$ implies 8 bits of metadata per block.

Deletion and Backyard Cleaning

Karlsruhe Institute of Technology

Suppose we have a hash table with a stable number of elements but a lot of insertions and deletions.

Problem:

So far we never unbump anything. Thus the backyard T' grows while there is more and more free space in the main table T.

Backyard cleaning:

When there is “enough” room in T to accommodate T',
Reset all thresholds to 0 and “merge” T' into T.
Various optimizations possible.

Succinct Slick

Karlsruhe Institute of Technology

- Map keys x via a pseudorandom permutation $\pi(x)$.
- Use $h(x) = \pi(x) \bmod m/B$ as block index.
- Store only quotient x div m/B. (And associated information)

Succinct Slick with Fingerprints

Store $\mathrm{O}(\log B)$ most significant bits of quotient separately.

Slick allowing Adaptive Growing

Karlsruhe Institute of Technology

Work in progress.
The vanilla way to grow a table is to reallocate with more space when the table gets too large (In Slick, the backyard would get too big).

Comparison: Slick vs. Linear Probing

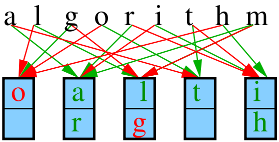
- \+ Less space achievable
- \+ No special empty element needed
- \+ Faster insertions and unsucessful search in space efficient configurations
- \+ Deterministic search time guarantees
- − (Somewhat) more complicated
- − Full concurrent implementation would be slow (locking issues)

h:												
	a	b	c	d	e	f	g	h	i	j	k	l
	m	n	o	p	q	r	s	t	u	v	w	x

a	m	o	⊥	⊥	r	g	t	i	h	⊥	l

Comparison: Slick vs Cuckoo

+ Faster Search at similar space?
+ No special empty element needed
+ Good provable insertion time bounds as a function of number of empty cells.
+ No rehashing with “unlucky” hash functions needed
+ May work with weaker families of hash functions ?



Interesting Special Cases / Variants

Karlsruhe Institute of Technology

- **Bumped Robin Hood Hashing:** Impose maximum search distance $\hat{o}$. Only bumping metadata. Possibly $B = 1$, $\hat{t} = 1$ (one bumping bit per table entry). Different notation in arxiv paper ($B \leftrightarrow \hat{o}$)
- **No bumping:** Blocked Robin Hood Hashing Faster insertions, search than classical Robin Hood?
- **No sliding:** Similar to iceberg/backyard cuckoo hashing. But more compact and concrete bumping information?
- **L**inear **C**uckoo (Luckoo) Hashing: x is in block $h(x)$ or $h(x) + 1$. Embed metadata into cache lines.

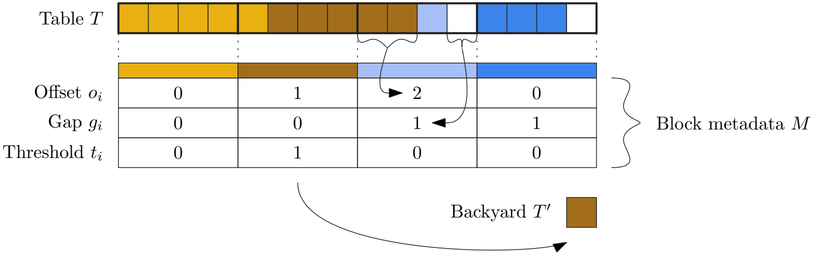
Succinct Slick

Store random permutations of keys

- Separate out $\mathrm{O}(\log\log n)$ bits from the keys of each element. Allows bit parallel search in constant time for $B = \mathrm{O}\left(\frac{\log n}{\log\log n}\right)$
- Cleary's trick:
  Extract $\log\frac{m}{B}$ key bits from $h(x)$.
  $\leadsto$ succinct variant with $\log\binom{|U|}{n} + \mathrm{O}(n\log B)$ bits of space

Future Work

- Efficient implementation. (SIMD? data dependencies? parameter tuning, compact metadata encoding, Luckoo?)
- Implement succinct variant
- Growing variant?
- More analysis (also for simple families of hash functions?)
- Variant for dynamic AMQ/Bloom Filter replacements?



More Comparison with Related Work

Iceberg, Backyard Cuckoo:
no sliding ($\leadsto$ less full table),
less explicit bumping ($\leadsto$ slower search)

Robin Hood: non-bumping Slick is similar but faster

Hopscotch: More but less effective metadata

Cuckoo with overlapping Windows: sliding, bumping$\rightarrow > 1$ choices

Bumped Ribbon Retrieval: Similar blocking, bumping and overloading;
Static, “smeared-out” information; construction using linear algebra

SlickHash Class

Class SlickHash$(m, B, \hat{B}, \hat{o}, \hat{t} : \mathbb{N}_+, h : E \to 0..m/B - 1)$

Class MetaData = $\overbrace{o : 0..\hat{o}}^{\text{offset}} \times \overbrace{g : 0..\hat{B}}^{\text{gap}} \times \overbrace{t : 0..\hat{t}}^{\text{threshold}}$

T : Array $[0..m-1]$ **of** E -- main table
$M = (0, B, 0)^{m/B} \circ (0, 0, 0)$: Array $[0..m/B]$ **of** MetaData
T': HashTable -- backyard

Function blockStart(i: $\mathbb{N}$) **return** $Bi + o_i$
Function blockEnd(i: $\mathbb{N}$) **return** $Bi + B + o_{i+1} - g_i - 1$
Function blockRange(i: $\mathbb{N}$) **return** blockStart(i)..blockEnd(i)

Procedure insert(e: E)

k:= key(e); i:= $h(k)$

if $\delta(k) < t_i$ **then** T'.insert(e); **return** —— e is already bumped

if $\exists j \in \text{blockRange}(h(k)) : \text{key}(T[j]) = k$ **then return**

if $\overbrace{|\text{blockRange}(i)| = \hat{B}}^{\text{block too large}}$ **or** $\overbrace{\textbf{not } (g_i > 0 \,||\, \text{slideGapFromRight}(i))}^{\text{no empty slot usable}}$ **then**

(* bump e or some element from block b_i *)

$t' := 1 + \min\{\delta(x) : x \in \{k\} \cup \{\text{key}(T[j]) : j \in \text{blockRange}(i)\}\}$

$t_i := t'$

$j :=$ blockStart(i)

while $j \leq$ blockEnd(i) **do** —— Scan existing elements. Bump them as

if $\delta(\text{key}(T[j])) < t'$ **then**

T'.insert($T[j]$) —— move to backyard

$T[j] := T[\text{blockEnd}(i)]$; g_i++ —— remove from T

else j++

if $\delta(k) < t'$ **then** T'.insert(e); **return**

g_i−−; $T[\text{blockEnd}(i)] := e$ —— insert e into an unused slot

return

(* Look for a free slot to the right and move it to block b_i if successful *)
Function slideGapFromRight(i_0: $\mathbb{N}$) : **boolean**

```
    i := i_0
    while g_i = 0 do                                    -- look for a free slot
        if i ≥ m/B ∨ o_i = ô then return false
        i++
    g_i--
    while i > i_0 do                  -- shift free slot towards block b_{i_0}
        (* Slide b_i to the right *)
        T[blockEnd(i) + 1] := T[blockStart(i)]
        o_i++
        i--
    g_i++
    return true
```