Sliding Block Hashing (Slick)
(Closed) Hash Tables

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

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

$$h: \begin{array}{cccccccccc}
a & b & c & d & e & f & g & h & i & j \\
m & n & o & p & q & r & s & t & u & v \\
\text{am} & o & \bot & \bot & r & g & t & i & h & \bot \\
\end{array}$$
Partition $T$ into blocks of size $B$. 
$h$ hashes each element $x$ to a block $h(x)$.
Invariant: $x \in T[iB + o_i..(i+1)B + o_{i+1} - g_i - 1]$ or $x$ is bumped

Store bumped elements in **backyard $T'$**
Simple and Compact Bumping

\[ \delta(x) < t_i \Rightarrow \text{bump } x \text{ to backyard } T' \]

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

\[
\begin{array}{c|c|c|c|c}
\text{Offset } o_i & 0 & 1 & 2 & 0 \\
\text{Gap } g_i & 0 & 0 & 1 & 1 \\
\text{Threshold } t_i & 0 & 1 & 0 & 0 \\
\end{array}
\]

Backyard \( T' \)
Search

\[ i := h(x) \]

If \( \delta(x) < t_i \) then return \( T'.\text{search}(x) \)

search \( x \) in \( T[\text{blockRange}(i)] \)

\[
\begin{array}{c|c|c|c|c}
\text{Offset } o_i & 0 & 1 & 2 & 0 \\
\text{Gap } g_i & 0 & 0 & 1 & 1 \\
\text{Threshold } t_i & 0 & 1 & 0 & 0 \\
\end{array}
\]

**Time:** \( O(B) \) if \( \hat{B} = O(B) \)
**Insert**

Move just one element per block.  
May be impossible or too expensive  
\[ \rightsquigarrow \text{bump sth near block } h(x) \].

**Time:** \( O(B) + T_{\text{bumpedCase}} \) if \( O(B) \) blocks allowed to slide
**Delete**

**Procedure** delete$(k: K)$

\[
i := h(k) \\
\text{if } \delta(k) < t_i \text{ then } T'.\text{delete}(k); \quad \text{return} \\
\text{if } \exists j \in \text{blockRange}(i) : \text{key}(T[j]) = k \text{ then} \\
\quad T[j] := T[\text{blockEnd}(i)] \quad \quad \quad \quad \quad \quad \text{-- overwrite deleted element} \\
\quad g_i := g_i +1 \quad \quad \quad \quad \quad \quad \text{-- extend gap}
\]

\[
T : \quad \text{agl} \quad \text{xi} \quad \text{rtr}
\]
Build($S$)

Simplification: no bumping, unbounded $o_i$ and overflow area

sort $S$ lexicographically 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:= $\langle \rangle$

sort $S$ lexicographycally by $(h(e), \delta(e))$

$o$:= 0

for $i := 0$ to $m/B - 1$ do

$b$:= $\langle e \in S : h(\text{key}(e)) = i \rangle$

t:= 0

if excess > 0 then

for $j := 1$ to excess do bumped.pushBack($b$.popFront)

t:= $\delta$($b$.last) + 1

while $|b| > 0 \land \delta(b$.front) < t do

bumped.pushBack($b$.popFront)

$M[i]$:= ($o$, max($0, B - o - |b|$), t)

for $j := 0$ to $|b| - 1$ do $T[iB + o + j]$:= $b[j]$ (write $b_i$ to $T$

$o$:= max($0, o + |b| - B$)

$M[m/B]$:= ($0, 0, 0$) (next offset)

$T'$.build(bumped) (sentinel metadata)

$\text{excess}$:= max($|b| - \hat{B}, o + iB + |b| - m, o + |b| - B - \hat{o}$)

$\text{offset}$ for$i$:= 0 to $m/B - 1$ do

$\text{extract block } b_i \text{ from } S$

$\text{threshold for } b_i$

$\langle \rangle$

sort $S$ lexicographycally by $(h(e), \delta(e))$

$o$:= 0

for $i := 0$ to $m/B - 1$ do

$b$:= $\langle e \in S : h(\text{key}(e)) = i \rangle$

t:= 0

if excess > 0 then

for $j := 1$ to excess do bumped.pushBack($b$.popFront)

t:= $\delta$($b$.last) + 1

while $|b| > 0 \land \delta(b$.front) < t do

bumped.pushBack($b$.popFront)

$M[i]$:= ($o$, max($0, B - o - |b|$), t)

for $j := 0$ to $|b| - 1$ do $T[iB + o + j]$:= $b[j]$ (write $b_i$ to $T$

$o$:= max($0, o + |b| - B$)

$M[m/B]$:= ($0, 0, 0$) (next offset)

$T'$.build(bumped) (sentinel metadata)
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

<table>
<thead>
<tr>
<th>Offset $o_i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gap $g_i$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Threshold $t_i$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Backyard $T'$
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.
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

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

Store $O(\log B)$ most significant bits of quotient separately.
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 unsuccessful 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

- **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?

- **Linear Cuckoo (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 $O(\log \log n)$ bits from the keys of each element. Allows bit parallel search in constant time for $B = O\left(\frac{\log n}{\log \log n}\right)$

- Cleary’s trick:
  Extract $\log \frac{m}{B}$ key bits from $h(x)$.

$\leadsto$ succinct variant with $\log \left(\frac{|U|}{n}\right) + O(n \log B)$ 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 (⇝ less full table),  
less explicit bumping (⇝ slower search)  

Robin Hood: non-bumping Slick is similar but faster  

Hopscotch: More but less effective metadata  

Cuckoo with overlapping Windows: sliding, bumping→> 1 choices  

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

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

\[
\text{offset} \quad \text{gap} \quad \text{threshold}
\]

**Class** MetaData = \(\hat{o} : 0..\hat{o} \times \hat{g} : 0..\hat{B} \times \hat{t} : 0..\hat{t}\)

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

**Function** blockStart\((i : \mathbb{N}) \text{ return } Bi + o_i\)
**Function** blockEnd\((i : \mathbb{N}) \text{ return } Bi + B + o_{i+1} - g_i - 1\)
**Function** blockRange\((i : \mathbb{N}) \text{ return } \text{blockStart}(i) .. \text{blockEnd}(i)\)
Procedure \texttt{insert}(e: E)
\begin{align*}
k &:= \text{key}(e); \quad i := h(k) \\
\text{if } \delta(k) < t_i \text{ then } T'.\text{insert}(e); \quad \text{return} \quad \text{--- } e \text{ is already bumped} \\
\text{if } \exists j \in \text{blockRange}(h(k)) : \text{key}(T[j]) = k \text{ then return} \\
\text{block too large} \\
\text{if } |\text{blockRange}(i)| = \hat{B} \text{ or not } (g_i > 0 \land \text{slideGapFromRight}(i)) \text{ then} \\
\text{--- no empty slot usable} \\
\text{(* bump } e \text{ 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 &:= \text{blockStart}(i) \\
\text{while } j \leq \text{blockEnd}(i) \text{ do} \quad \text{--- Scan existing elements. Bump them as necessary} \\
\text{if } \delta(\text{key}(T[j])) < t' \text{ then} \\
T'.\text{insert}(T[j]) \\
T[j] := T[\text{blockEnd}(i)]; \quad g_i++; \quad \text{--- move to backyard} \\
\text{else } j++ \\
\text{if } \delta(k) < t' \text{ then } T'.\text{insert}(e); \quad \text{return} \\
g_i--; \quad T[\text{blockEnd}(i)] := e \quad \text{--- insert } e \text{ into an unused slot} \\
\text{return} 
\end{align*}
(* 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**  

\[
\text{if } i \geq m/B \lor o_i = \hat{o} \text{ then return false}
\]

\[
i++
\]

\[
g_i--
\]

**while** \( i > i_0 \) **do**  

(* Slide \( b_i \) to the right *)

\[
T[\text{blockEnd}(i) + 1] := T[\text{blockStart}(i)]
\]

\[
o_i++
\]

\[
i--
\]

\[
g_i++
\]

return true