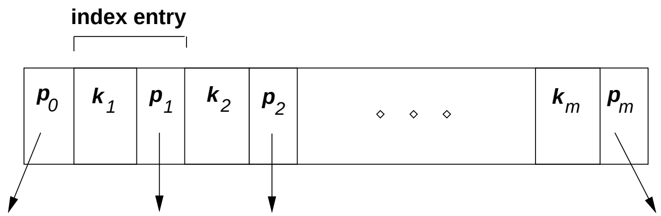
Algorithm: *search* (k)
Input: search key value k
Output: pointer to B^+ tree page containing potential hit(s)

`return` $\mathit{tree\_search}\ (\mathit{root}, k)$; // root denotes the root page of the B^+ tree

Algorithm: *tree_search* (p, k)
Input: current page p, search key value k
Output: pointer to B^+ tree page containing potential hit(s)

`if` $\mathit{leaf}(p)$ `then`
    `return` p;
`else`
    `if` $k < k_1$ `then`
        `return` $\mathit{tree\_search}\ (p_0, k)$;
    `else`
        `if` $k \geq k_m$ `then`
            `return` $\mathit{tree\_search}\ (p_m, k)$;
        `else`
            find i such that $k_i \leq k < k_{i+1}$;
            `return` $\mathit{tree\_search}\ (p_i, k)$;

// layout of p:



Algorithm: $insert\ (p, k*)$
Input: current page p, entry $k*$ to be inserted
Output: entry propagated upwards the tree (`NULL` if no further propagation)

$m \leftarrow \#entries(p)$;
`if` $\neg leaf(p)$ `then`
  on p, find i such that $k_i \leq k < k_{i+1}$;
  $n \leftarrow insert(p_i, k*)$;
  `if` $n =$ `NULL` `then`
    `return` `NULL`;
  `else`
    `if` $m < 2 \cdot d$ `then`
      insert n into p;
      `return` `NULL`;
    `else`
      // $m + 1 = d + \underbrace{1}_{k'} + d$
      split p into p and new page p',
      first d keys and $d + 1$ pointers stay on p,
      last d keys and $d + 1$ pointers go to p';
      $n \leftarrow \langle$middle key k', $addr(p')\rangle$;
      `if` $root(p)$ `then`
        $r \leftarrow$ new empty node;
        $root(r) \leftarrow true$;
        insert $addr(p)$ into r; // as p_0
        insert n into r;
        `return` `NULL`;
      `else`
        `return` n;
`else`
  // p is a leaf node
  `if` $m < 2 \cdot d$ `then`
    insert $k*$ into p;
    `return` `NULL`;
  `else`
    // $m + \underbrace{1}_{k*} = 2 \cdot d + 1$
    split p into p and new page p',
    first d entries stay on p, last $d + 1$ entries go to p';
    $n \leftarrow \langle$smallest value k' on p', $addr(p')\rangle$;
    `return` n;

Algorithm: $delete\ (p, k)$
Input: current page p, key value k to be deleted
Output: key value to be deleted in p's parent
(NULL if no further deletion in parent)

```
m ← #entries(p);
if ¬leaf(p) then
    // p is a non-leaf node
    find i such that k_i ≤ k < k_{i+1};
    n ← delete(p_i, k);
    if n = NULL then
        return NULL;
    else
        remove entry with key n from p;
        if root(p) ∧ m = 1 then
            // last entry in root deleted, determine new root
            root(addr(p_0)) ← true;
            delete p;
            return NULL;
        if m > d then
            return NULL;
        else
            // underflow in p
            p' ← sibling(p);        // wlog: p' right sibling of p
            if #entries(p') > d then
                // non-leaf node redistribution
                move smallest entry of p' (= ⟨k', p'_0⟩) into p;
                swap(separator(p, p'), key value k' in p);
                return NULL;
            else
                // merge non-leaf nodes p, p'
                insert separator(p, p') into p;
                move all entries from p' to p;
                delete p';
                return separator(p, p');
else
    ⋮
    // leaf node handling: see next slide
    ⋮
```

Algorithm: $delete\ (p, k)$
Input: current page p, key value k to be deleted
Output: key value to be deleted in p's parent
(NULL if no further deletion in parent)

```
m ← #entries(p);
if ¬leaf(p) then
    ⋮
    // non-leaf node handling: see previous slide
    ⋮
else
    // p is a leaf node
    if k* found on p then
        remove k* from p;
    else
        return NULL;
    if m > d then
        return NULL;
    else
        // underflow in p
        p' ← sibling(p);        // wlog: p' right sibling of p
        if #entries(p') > d then
            // leaf node redistribution
            move entry from p' to p;
            separator(p, p') ← smallest key value on p';
            return NULL;
        else
            // merge leaf nodes p, p'
            move all entries from p' to p;
            delete p';
            return separator(p, p');
```

- $separator(p, p')$ represents the separating key value k in the common parent node of siblings p and p'.
- $\#entries(p)$ computes the number of actually occupied entries in B^+ tree node p.
- $swap(x, y)$ exchanges the values of x and y.

Algorithm: *hsearch* (k)
Input: search for hashed record with key value k
Output: pointer to hash bucket containing potential hit(s)

```
n ← [n];            // global depth of hash directory
b ← h(k) & (2^n − 1);
return bucket[b];
```

Algorithm: *hinsert* $(k*)$
Input: entry $k*$ to be inserted
Output: new global depth of extendible hash directory

```
n ← [n];            // global depth of hash directory
b ← hsearch(k);
if b has capacity then
    place h(k)* in bucket b;
    return n;
else
    // bucket b overflows, we need to split
    d ← [d];            // local depth of bucket b
    create a new empty bucket b2;
    // redistribute entries of bucket b including h(k)*
    for each h(k')* in bucket b do
        if h(k') & 2^d ≠ 0 then
            move h(k')* to bucket b2;
    [d] ← d + 1;        // new local depth of buckets b and b2
    if n < d + 1 then
        // we need to double the directory
        allocate 2^n directory entries bucket[2^n ... 2^(n+1) − 1];
        copy bucket[0 ... 2^n − 1] into bucket[2^n ... 2^(n+1) − 1];
        n ← n + 1;
        [n] ← n;
    bucket[(h(k) & (2^(n−1) − 1)) | 2^(n−1)] ← addr(b2);
return n;
```

Remarks:

- & and | denote **bit-wise and** and **bit-wise or** (just like in C, C++)
- The directory entries are accessed via the array $bucket[0 \ldots 2^{\boxed{n}} - 1]$ whose entries point to the hash buckets.

Algorithm: $hsearch\ (k)$
Input: search for hashed record with key value k
Output: pointer to hash bucket containing potential hit(s)

```
b ← h_level(k);
if b < next then
    // bucket b has already been split,
    // the record for key k may be in bucket b or bucket 2^level · N + b,
    // rehash:
    b ← h_level+1(k);
return bucket[b];
```

Algorithm: $hinsert\ (k*)$
Input: entry $k*$ to be inserted
Output: none

```
b ← h_level(k);
if b < next then
    // bucket b has already been split, rehash:
    b ← h_level+1(k);
place h(k)* in bucket[b];
if full(bucket[b]) then
    // the last insertion triggered a split of bucket next
    allocate a new bucket b';
    bucket[2^level · N + next] ← b';
    // rehash the entries of bucket next
    for each entry with key k' in bucket[next] do
        place entry in bucket[h_level+1(k')];
    next ← next + 1;
    // did we split every bucket in the original hash table?
    if next > 2^level · N − 1 then
        // hash table size has doubled, start a new round now
        level ← level + 1;
        next ← 0;
return ;
```

Remarks:

- $bucket[b]$ denotes the bth bucket in the hash table.
- Function $full(\cdot)$ is a tunable parameter: whenever $full(bucket[b])$ evaluates to $true$ we trigger a split.

► $hdelete(k)$ for a linear hash table can essentially be implemented as the inverse of $hinsert(k)$:

Algorithm: $hdelete\ (k)$
Input: key k of entry to be deleted
Output: none

⋮

```
if empty(bucket[2^level · N + next]) then
    // the last bucket in the hash table is empty, remove it
    remove bucket[2^level · N + next] from hash table;
    next ← next − 1;
    if next < 0 then
        // round-robin scheme for deletion
        level ← level − 1;
        next ← 2^level · N − 1;
```

⋮