Recitation 13

B+trees, Linear Hashing, and Extendible Hashing

1

B+-tree Index

- A node is of the form:

  $[p_0, k_1, p_1, k_2, p_2, .., k_i, p_i, k_{i+1} .., k_n, p_n]$

- p_i's are pointers and k_i's are field values (keys)
- ***Tree Order*** is the number of pointers, e.g., ***n***
- For every field value ***k*** in a node pointed to by p_i

  $k_i < k \leq k_{i+1}$ (alternative $k_i \leq k < k_{i+1}$)

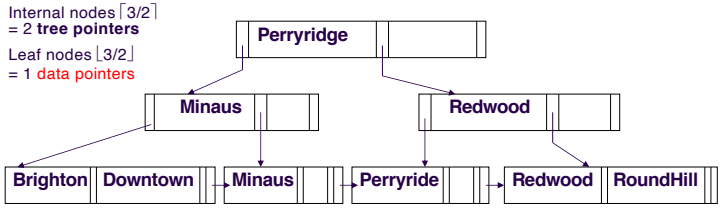
- Every node, except for the root, has between n/2 and n children or pointers
  - internal: ⌈n/2⌉ **tree pointers**; leaf: ⌊n/2⌋ data pointers
- Leaf nodes are chain to form a link list (***fast sequential access***)

2

Examples of B+ trees

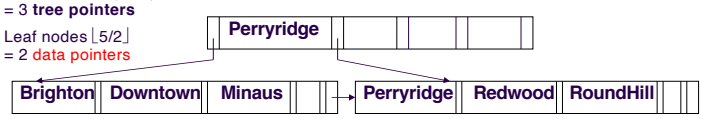
B+-tree with n=3

- Internal nodes ⌈3/2⌉ = 2 **tree pointers**
- Leaf nodes ⌊3/2⌋ = 1 data pointers



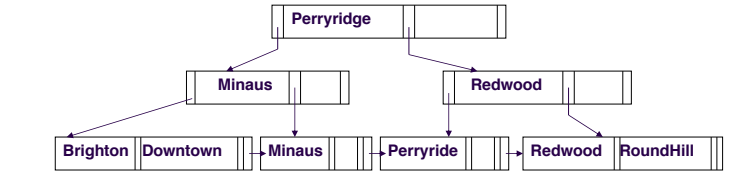
B+-tree with n=5

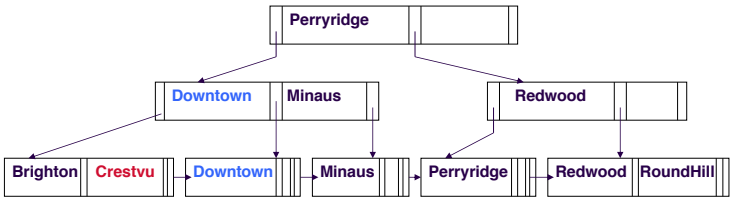
- Internal nodes ⌈5/2⌉ = 3 **tree pointers**
- Leaf nodes ⌊5/2⌋ = 2 data pointers



3

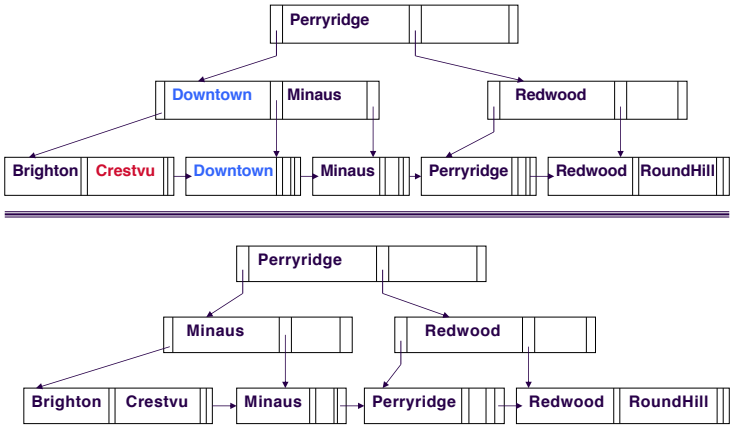
Insertion of "Crestvu"



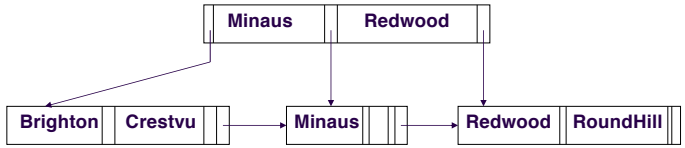


4

Deletion of "Downtown"



Deletion of "Perryridge"

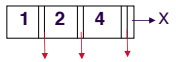


Insert Examples of B+ trees

B+-tree with n=4, i.e.,

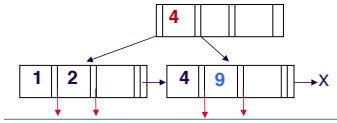
- internal nodes should have ⌈4/2⌉ =2 **tree pointers**;
- Leaf nodes should have ⌊4/2⌋ = 2 data pointers

Step 1: insert 1, 2, 4



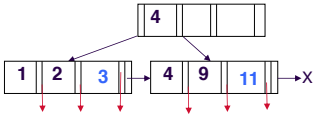
Step 2: insert 9

Although you push 9 on the stack & inserted afterwards when allocating a new block, it is considered when deciding the split: 1 2 4. So the split is at 1 2 | 4 9 and copy up 4



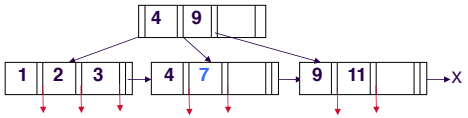
Insert Examples of B+ trees

Step 3: insert 3, 11



Step 4: insert 7

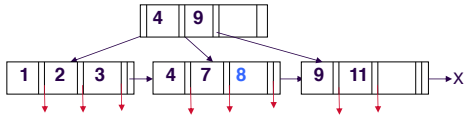
Although you push 7 on the stack when allocating a new block, it is considered when deciding the split: 4 9 11. So the split is at 4 7 | 9 11



9

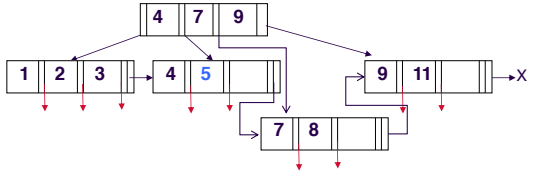
Insert Examples of B+ trees

Step 5: insert 8



Step 6: insert 5

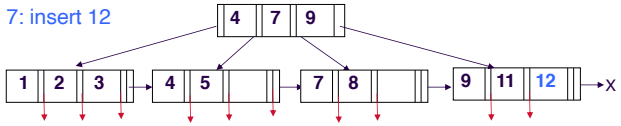
Although you push 5 on the stack when allocating a new block, it is considered when deciding the split: 4 7 8. So the split is at 4 5 | 7 8



10

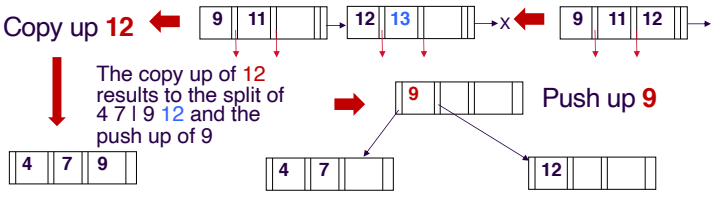
Insert Examples of B+ trees

Step 7: insert 12



Step 8: insert 13

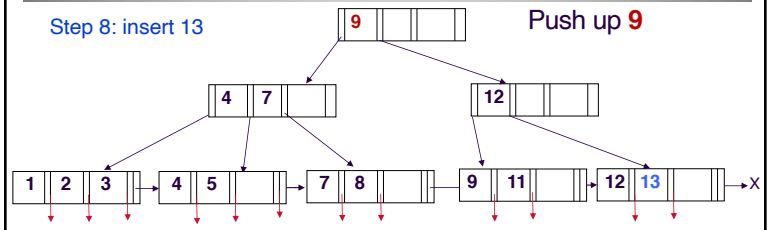
Push 13 on the stack when allocating a new block, it is considered when deciding the split: 9 11 12. So the split is at 9 11 | 12 13. then copy up 12.



11

Insert Examples of B+ trees

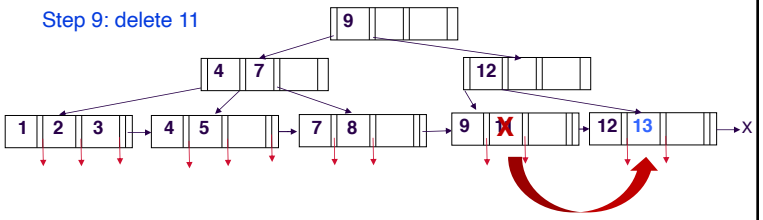
Step 8: insert 13



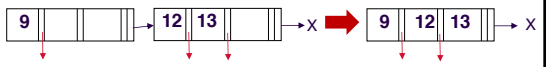
12

Delete Examples of B+ trees

Step 9: delete 11



Merge

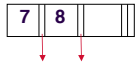


13

Examples of B+ trees

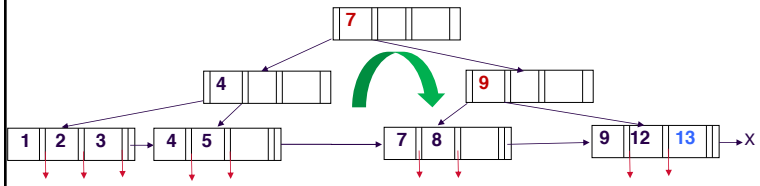
Step 9: delete 11 (cont)

Borrow from the left subtree



Push up 7

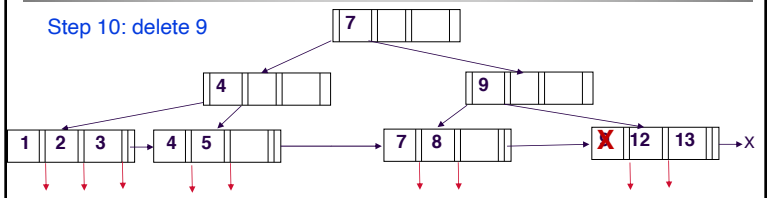
Push down 9



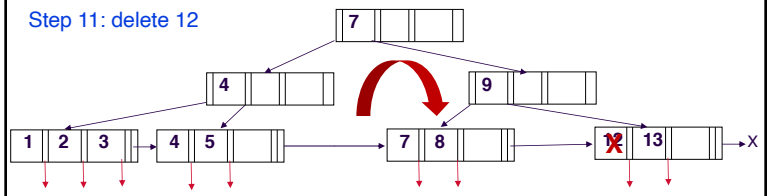
14

Examples of B+ trees

Step 10: delete 9



Step 11: delete 12

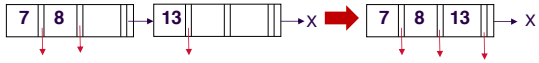


15

Examples of B+ trees

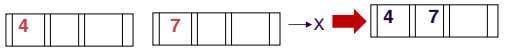
Step 11: delete 12 (cont)

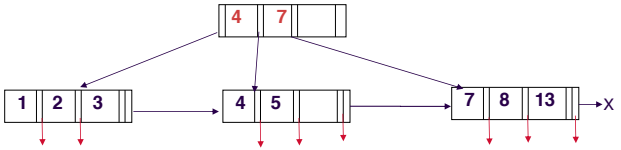
Merge



We can't borrow a node from the left subtree

Merge the root





16

17

Linear Hashing

- The file size grows linearly, bucket by bucket, using a family of hash functions, each has double the range of its predecessor
  - The file starts with ***s*** main buckets, where ***s*** is a power of 2, and ***k*** overflow buckets.
  - Buckets are numbered from 0 to ***s-1***
    => initial hashing function $h_0(key) = key \bmod s$.
  - Collisions are handled thru ***chaining***.
- We keep the following info:
  - B_{last}: A pointer to the ***current*** last bucket.
    Initially, $B_{last} = s-1$.
  - B_{split}: A pointer to the bucket that should be split next.
    Initially, $B_{split} = 0$

18

Insertion of a Record (Method 1)

- if there is a collision, push tuple to an overflow bucket.
- if there are no overflow buckets available, proceed as follows until one becomes available:
  - A new bucket is appended at the end of the hash table and $B_{last} = B_{last} + 1$
  - Records in B_{split} bucket are hashed again using
    $h_1(key) = key \bmod (2s)$,
    -- these will either remain in B_{split} or stored in B_{last}

  $\Rightarrow$ *this may free an overflow bucket.*
  - B_{split} becomes $B_{split} + 1$.
- if $B_{last} = 2s-1$
  set $s = 2s$, $B_{last} = s-1$, $B_{split} = 0$, $h_0(key) = h_1(key)$
- proceed as above.

19

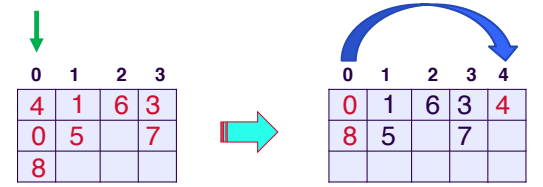
Insertion of a Record (Method 2)

- if there is a collision, push tuple to an overflow bucket.
- if there are no overflow buckets available, proceed as follows until one becomes available:
  - A new bucket is appended at the end of the hash table (but B_{last} *does not change as in Method 1)*
  - Records in B_{split} bucket are hashed again using
    $h_1(key) = key \bmod (2s)$,
    -- these will either remain in B_{split} or stored in B_{last}

  $\Rightarrow$ *this may free an overflow bucket.*
  - B_{split} becomes $B_{split} + 1$.
- if $B_{last} < B_{split}$
  set $s = 2s$, $B_{last} = s-1$, $B_{split} = 0$, $h_0(key) = h_1(key)$
- proceed as above.

20

Example



B_{split}= 0 B_{last}=3

$h_0(k)=k \bmod 2^2$

1, 4, 3, 0, 6, 8, 5, 7, 16 ?

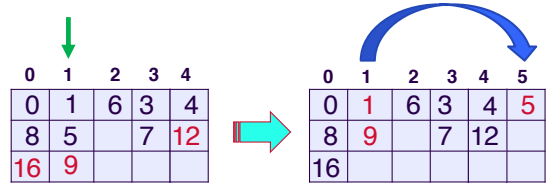
B_{split}= 1 B_{last}=3

$h_0(k)=k \bmod 2^2$

$h_1(k)=k \bmod 2^3$

21

Example



B_{split}= 1 B_{last}=3

B_{split}= 2 B_{last}=3

$h_0(k)=k \bmod 2^2$

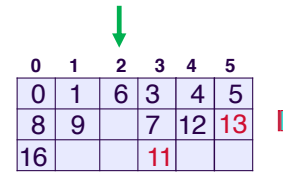
$h_1(k)=k \bmod 2^3$

1, 4, 3, 0, 6, 8, 5, 7, 16, 12, 9, 13?

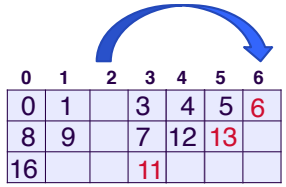
Is $B_{split} > B_{last}$?

22

Example



B_{split}= 2 B_{last}=3



B_{split}= 3 B_{last}=3

$h_0(k)=k \bmod 2^2$

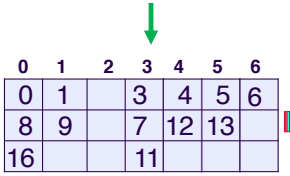
$h_1(k)=k \bmod 2^3$

1, 4, 3, 0, 6, 8, 5, 7, 16, 12, 9, 13, 11, 15 ?

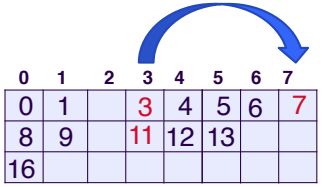
Is $B_{split} > B_{last}$?

23

Example



B_{split}= 3 B_{last}=3



B_{split}= 4 > B_{last}=3

$h_0(k)=k \bmod 2^2$

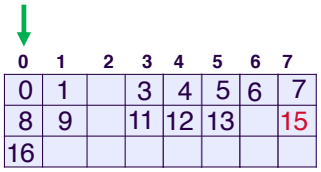
$h_1(k)=k \bmod 2^3$

1, 4, 3, 0, 6, 8, 5, 7, 16, 12, 9, 13, 11, 15 ?

Is $B_{split} > B_{last}$?

24

Example



B_{split}= 0 B_{last}=7

~~$h_0(k)=k \bmod 2^2$~~ $h_0(k)=k \bmod 2^3$

$h_1(k)=k \bmod 2^3$ $h_1(k)=k \bmod 2^4$

1, 4, 3, 0, 6, 8, 5, 7, 16, 12, 9, 13, 11, 15

Search for a record

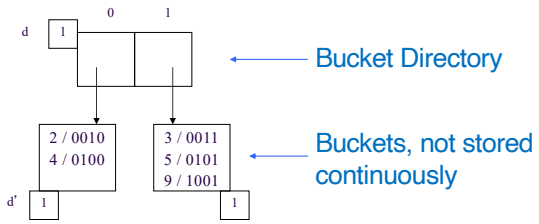
- Search for a Record
  - ***l* = h_0(key)**
  - if ***l* < B_{split}** then ***l* = h_1(key)**
  - search the bucket whose hash value is ***l*** (and its overflow, if any).

Dynamic Hashing Methods

- Allow the file size to change as records are added or deleted.
  - Linear Hashing
    - No additional structure
  - **Extendible Hashing**
  - Binary Hashing

Extendible Hashing

- The file is structured into two levels: ***directory*** and buckets.



Extendible Hashing

- Directory contains only pointers to buckets (no keys)
- Use some hash function to generate a ***pseudokey*** of b-bits (typically b=32)
- Use the first (or last) ***d*** bits to find the offset of the bucket pointer in the directory.
- ***d*** is called the (global) ***directory depth***. It may be stored in the directory
- In each bucket a local depth ***d'*** is stored indicating the most (or least) significant bits common to all keys in that bucket (***d'≤ d***)

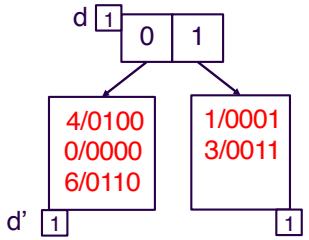
Extendible Hashing – File Growth

- No overflow buckets. New buckets are added one by one.
- When a bucket B with local depth ***d'*** overflows, B is split into two buckets and all keys are rehash using ***d'+1*** bits
- If after a split, ***d' > d***, double the size of the directory (***d=d+1***) to accommodate the growth

Example

- Use the least significant bits as bucket keys
- Each bucket can hold three records
- The keys to insert are 1,4,3,0,6,8,5,7,16,12,9,13

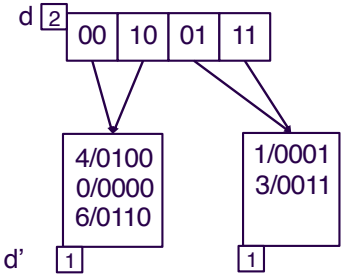
Example



1, 4, 3, 0, 6, 8/1000

33

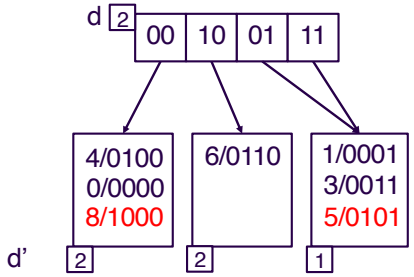
Example



1, 4, 3, 0, 6, 8/1000

34

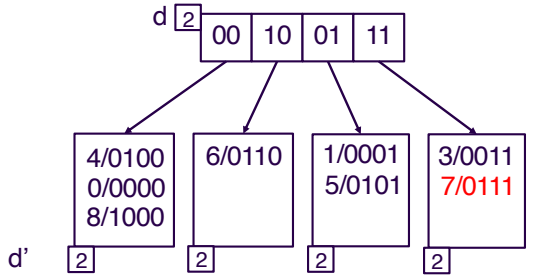
Example



1, 4, 3, 0, 6, 8, 5, 7/0111

36

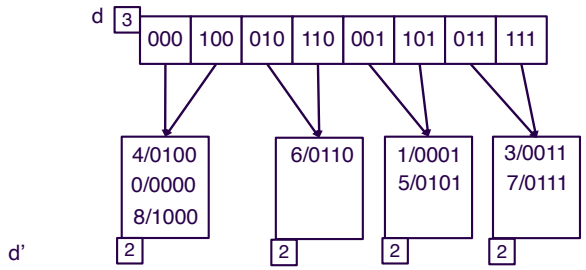
Example



1, 4, 3, 0, 6, 8, 5, 7, 16/10000

37

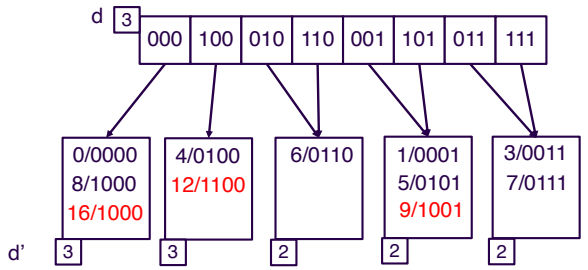
Example



1, 4, 3, 0, 6, 8, 5, 7, 16/10000

38

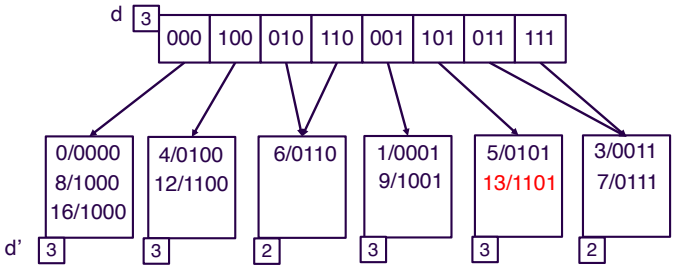
Example



1, 4, 3, 0, 6, 8, 5, 7, 16, 12, 9, 13/1101

39

Example



1, 4, 3, 0, 6, 8, 5, 7, 16, 12, 9, 13

40