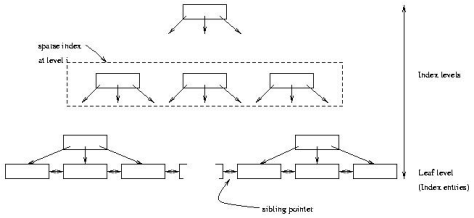
## B+ Tree

- Supports equality and range searches, multiple attribute keys and partial key searches
- Either a secondary index (in a separate file) or the basis for an integrated storage structure
- Responds to dynamic changes in the table

44

## B+ Tree Structure




- Leaf level is a (sorted) linked list of index entries
- Sibling pointers support range searches in spite of allocation and deallocation of leaf pages (but leaf pages might not be physically contiguous)

45

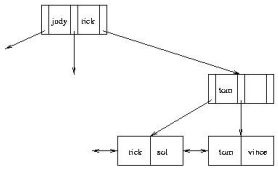
## Insertion and Deletion in B+ Tree

- Structure of tree changes to handle row insertion and deletion - *no* overflow chains
- Tree remains balanced: all paths from root to index entries have same length
- Algorithm guarantees that the number of separator entries in an index page is between $\Phi/2$ and $\Phi$
  - Hence the maximum search cost is $log_{\Phi/2}Q + 1$ (with ISAM search cost depends on length of overflow chain)

46

## Handling Insertions - Example

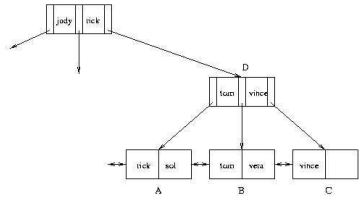
- Insert "vince".




47

## Handling Insertions (con't)

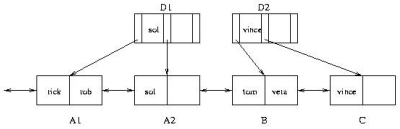
- Insert "vera": Since there is no room in leaf page:
  1. Create new leaf page, C
  2. Split index entries between B and C (but maintain sorted order)
  3. Add separator entry at parent level




48

## Handling Insertions (con't)

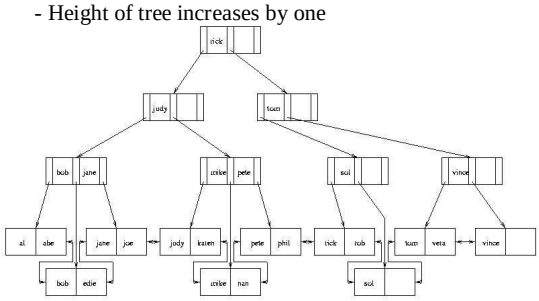
- Insert "rob". Since there is no room in leaf page A:
  1. Split A into A1 and A2 and divide index entries between the two (but maintain sorted order)
  2. Split D into D1 and D2 to make room for additional pointer
  3. Three separators are needed: "sol", "tom" and "vince"




49

## Handling Insertions (con't)

- When splitting a separator page, push a separator up
- Repeat process at next level
- Height of tree increases by one




## Handling Deletions

- Deletion can cause page to have fewer than $\Phi/2$ entries
  - Entries can be redistributed over adjacent pages to maintain minimum occupancy requirement
  - Ultimately, adjacent pages must be merged, and if merge propagates up the tree, height might be reduced
- In practice, tables generally grow, and merge algorithm is often not implemented
  - Reconstruct tree to compact it

51

## Hash Index

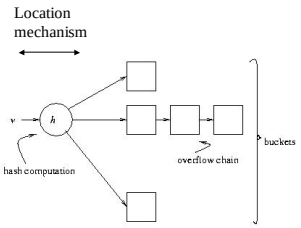
- Index entries partitioned into *buckets* in accordance with a *hash function*, $h(v)$, where $v$ ranges over search key values
  - Each bucket is identified by an address, $a$
  - Bucket at address $a$ contains all index entries with search key $v$ such that $h(v) = a$
- Each bucket is stored in a page (with possible overflow chain)
- If index entries contain rows, set of buckets forms an integrated storage structure; else set of buckets forms an (unclustered) secondary index

52

## Equality Search with Hash Index

Given $v$:
1. Evaluate $h(v)$
2. Fetch bucket at $h(v)$
3. Search bucket

Cost is number of pages in bucket (cheaper than $B^+$ tree if no overflow chains)




53

## Choosing a Hash Function

- Goal of $h$: map search key values randomly
  - Occupancy of each bucket roughly same for an average instance of indexed table
- Example: $h(v) = (c_1 * v + c_2) \bmod M$
  - $M$ must be large enough to minimize the occurrence of overflow chains
  - $M$ must not be so large that bucket occupance is small and too much space is wasted

54

## Hash Indices - Problems

- Does not support range search
  - Since adjacent elements in range might hash to different buckets, there is no way to scan buckets to locate all search key values $v$ between $v_1$ and $v_2$
- Although it supports multiple attribute keys, it does not support partial key search
  - Entire value of $v$ must be provided to $h$
- Dynamically growing files produce overflow chains, which negate the efficiency of the algorithm

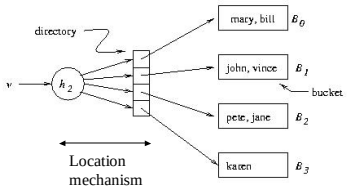
55

# Extendable Hashing

- Eliminates overflow chains by splitting a bucket when it overflows
- Range of hash function has to be extended to accommodate additional buckets
- **Example:** family of hash functions based on *h*:
  - $h_k(v) = h(v) \mod 2^k$ (use the last *k* bits of *h(v)*)
  - At any given time a unique hash, $h_k$, is used depending on the number of times buckets have been split



# Extendable Hashing - Example




| *v* | *h(v)* |
|---|---|
| pete | 11010 |
| mary | 00000 |
| jane | 11110 |
| bill | 00000 |
| john | 01001 |
| vince | 10101 |
| karen | 10111 |

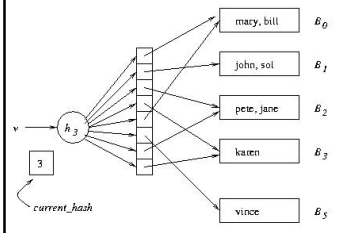
Extendable hashing uses a directory (level of indirection) to accommodate family of hash functions

Suppose next action is to insert sol, where *h(sol) = 10001.*

**Problem**: This causes overflow in $B_1$



# Example (con't)




**Solution**:

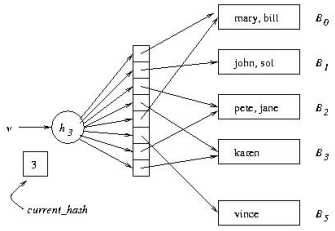
1. Switch to $h_3$
2. Concatenate copy of old directory to new directory
3. Split overflowed bucket, *B*, into *B* and *B´*, dividing entries in *B* between the two using $h_3$
4. Pointer to *B* in directory copy replaced by pointer to *B´*

Note: Except for *B´*, pointers in directory copy refer to original buckets.

*current_hash* identifies current hash function.



# Example (con't)




Next action: Insert judy, where *h(judy) = 00110*

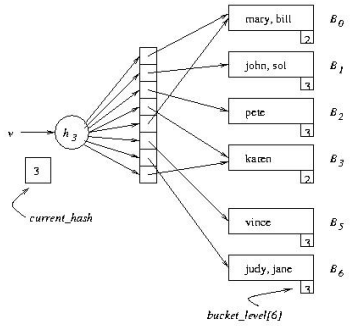
$B_2$ overflows, but directory need not be extended

**Problem**: When $B_i$ overflows, we need a mechanism for deciding whether the directory has to be doubled

**Solution**: *bucket_level[i]* records the number of times $B_i$ has been split. If *current_hash* > *bucket_level[i]*, do not enlarge directory



# Example (con't)






# Extendable Hashing

- Deficiencies:
  - Extra space for directory
  - Cost of added level of indirection:
    - If directory cannot be accommodated in main memory, an additional page transfer is necessary.

## Choosing An Index

- An index should support a query of the application that has a significant impact on performance
  - Choice based on frequency of invocation, execution time, acquired locks, table size

Ex 1
```
SELECT E. Id
FROM   Employee E
WHERE  E.Salary < :upper  AND  E.Salary > :lower
```

- This is a range search on Salary.
- Since the primary key is Id, it is likely that there is a clustered, main index on that attribute that is of no use for this query.
- Choose a secondary, $B^+$ tree index with search key Salary

## Choosing An Index

Ex 2
```
SELECT T.StudId
FROM   Transcript T
WHERE  T.Grade = :grade
```

- This is an equality search on Grade.
- Since the primary key is (StudId, Semester, CrsCode) it is likely that there is a main, clustered index on these attributes that is of no use for this query.
- Choose a secondary, B+ tree or hash index with search key Grade

63

## Choosing an Index

Ex 3
```
SELECT  T.CrsCode, T.Grade
FROM  Transcript T
WHERE  T.StudId = :id  AND  T.Semester = 'F2000'
```

- Equality search on StudId and Semester.
- If the primary key is (StudId, Semester, CrsCode) it is likely that there is a main, clustered index on this *sequence* of attributes.
- If the main index is a $B^+$ tree it can be used for this search.
- If the main index is a hash it cannot be used for this search. Choose $B^+$ tree or hash with search key StudId (since Semester is not as selective as StudId) or (StudId, Semester)

64

## Choosing An Index

Ex 3 (con't)
```
SELECT  T.CrsCode, T.Grade
FROM  Transcript T
WHERE  T.StudId = :id  AND  T.Semester = 'F2000'
```

- Suppose Transcript has primary key (CrsCode, StudId, Semester). Then the main index is of no use (independent of whether it is a hash or $B^+$ tree).

65