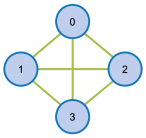
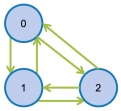
Complete Graph

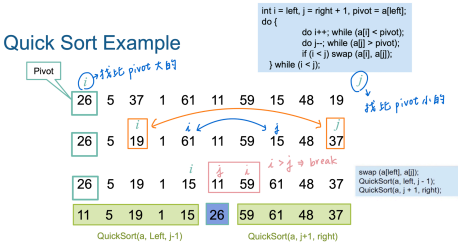
- Complete undirected graph
  - Graph with n vertices has exactly $n(n-1)/2$ edges



- Complete directed graph
  - Graph with n vertices has exactly $n(n-1)$ edges



Quick Sort Example



Properties

- Find a better splitting record:
  - Try to find the median one
  - Median{ first, middle, last}
- Not a stable sort

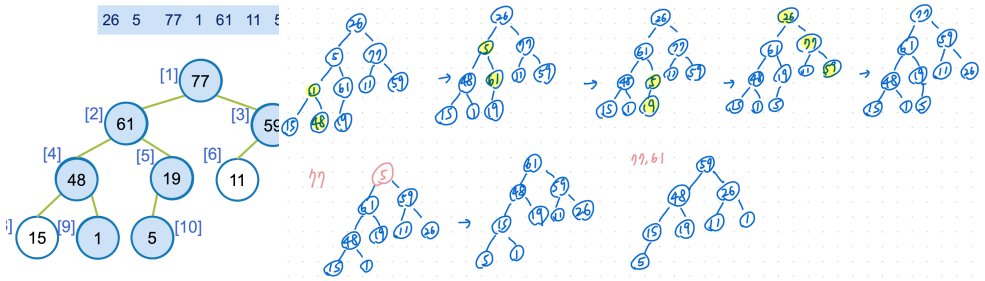
not able to be utilize in radix sort

Properties

- Time complexity
  - Number of merge pass: O(logn)
  - Time complexity of merge pass: O(n)
  - Time complexity = O(nlogn)
- Require additional storage to store merged results
- Stable sort

no swaping

Running Example for Heap Sort



Bin Sort (Bucket Sort)

- Assume the sorted records come from a set of size **m**, {1,2,...m}
- Create **m** buckets
- Scan the sequence a[1] ... a[n], and put a[i] element into the **a[i]**th bucket
- Concatenate all buckets to get the sorted list
  - Suitable for a set with small **m**

	Complexity
Worst	$O(n^2)$
Best	$O(n+k)$
Average	$O(n+k)$
Worst space	$O(n+k)$ auxiliary

k: 桶數, n: 資料數

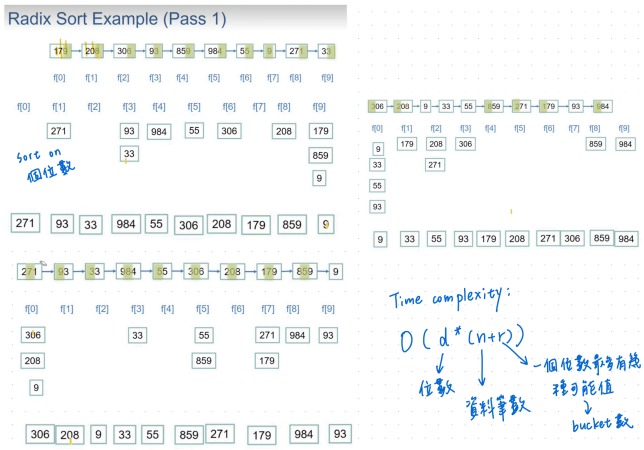
Example:
6, 28, 96, 14, 74, 37, 9, 71, 91, 36

bucket

排序 bucket 中元素 (insertion sort)

→ 6, 9, 14, 28, 36, 37, 71, 74, 91, 96

Radix Sort Example (Pass 1)



Time complexity:

O(d * (n+r))

d: 位數, n: 資料筆數, r: 一個位數有多少種可能值 → bucket 數

Design Guidelines

- Insertion sort is good for small n and when the list is partially sorted
- Merge sort is slightly faster than heap sort but it require additional storage
- Quick sort outperforms in average

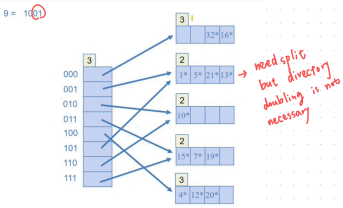
External Sorting Algorithm

- Insertion sort, Quick sort, Heap sort ⇒ not good for external sorting
- Merge sort ⇒ good for external sorting
  - ↳ cannot put unsorted/sorted data into disk while sorting
  - Segments (runs) of input lists sorted using an internal sort
  - The runs generated in phase one are merged together following the **merge-tree** pattern
- Why merge sort?
  - Sublists could be sorted independently and merged later
  - During the merging, only the leading records of the two runs needed to be loaded in memory

External Sort

- The lists are **too large** to be completely loaded
  - The list could reside on a disk
- The external sorting algorithm
  - Read partial records
  - Perform the sorting
  - Write the result back to disk
- "**Block**"
  - The unit of data that is read/written at one time

Dynamic Hashing: Insert 9*



Collisions

- Many keys might be mapped to the same home bucket
- Collision
  - When a key is mapped to a non-empty home bucket
- Overflow
  - When a key is mapped to a full home bucket
- Overflow and collision occur simultaneously when each bucket has 1 slot

Huffman Codes

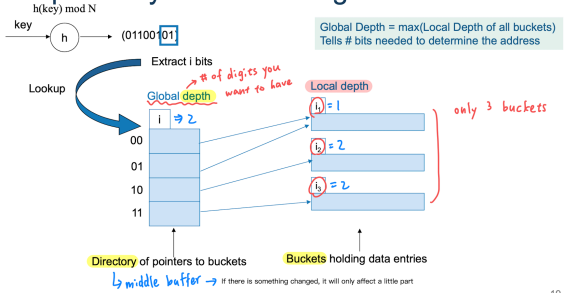
- Cost of decoding a code word is proportional to the number of bits in the code
- Assume the frequency of a message M_i been transmitted is q_i, the total cost is:

$$\sum_{i=1}^{n} q_i d_i$$

- How do we construct a decode tree such that the transmission cost is minimized?

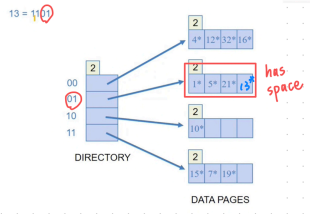
↗ can be used for not only uniform

Example of Dynamic Hashing

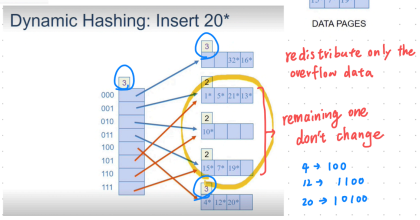


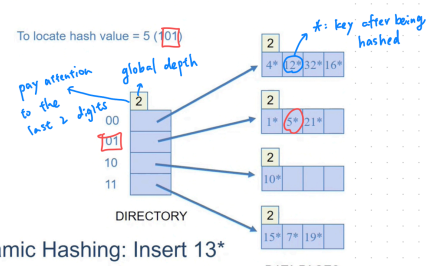
Global Depth = max(Local Depth of all buckets)
Tells # bits needed to determine the address

Dynamic Hashing: Insert 13*

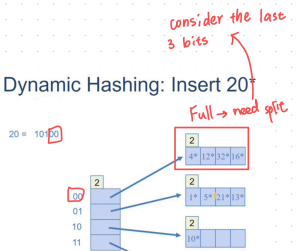


Dynamic Hashing: Insert 20*





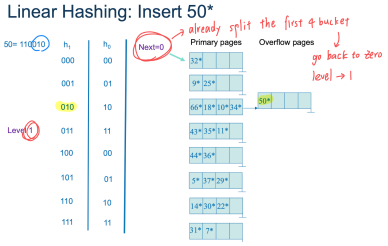
Dynamic Hashing: Insert 20*



Linear Hashing

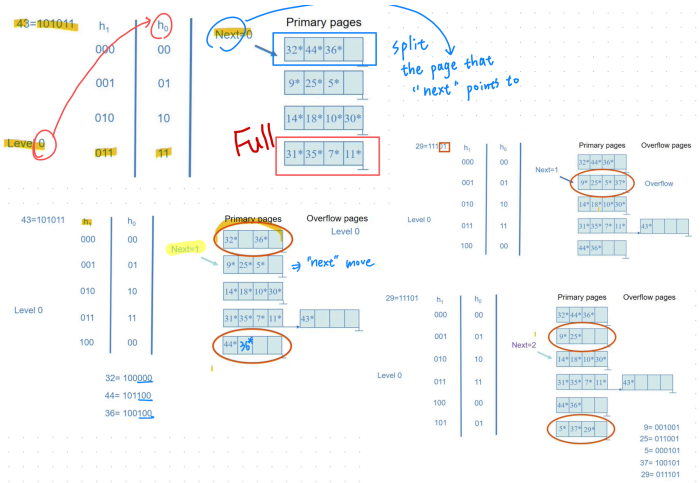
- Requires an overflow area
- Blocks are split in linear order
- Round round-robin fashion
  - eventually all buckets are split
- Idea: Use a family of hash functions h0, h1, h2, ...
  - $h_i(key) = h(key) \bmod (2^i N)$; N = initial # buckets
  - If $N = 2^{d_0}$, for some d_0, h_i checks the last d_i bits, where $d_i = d_0 + i$.
  - h_{i+1} doubles the range of h_i

Linear Hashing: Insert 50*



B-tree: Insert

- Search
- Insert the new key into a leaf
- If the leaf overflows
  - Split the leaf into two and push up the middle key to the leaf's parent
  - If the parent overflows
    - Split the parent into two and push up the middle key again
  - This strategy might have to be repeated all the way until arriving the root
  - If necessary, the root is split in two and the middle key is pushed up to a new root, making the tree one level higher



Local/Global Depth

- Initially, all local depths are equal to global depth
  - # of bits need to express the total # of buckets
- While the process of split, if a bucket whose local depth = global depth
  - The directory must be doubled
- Global depth + 1 when the directory doubles
  - Local depth + 1 when a bucket is split

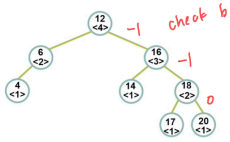
B-tree: Definition

bf = 0

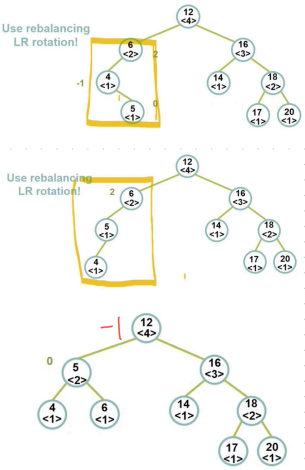
- A B-tree of order *m* is a height-balanced tree , where each node may have up to *m* children, and in which:
  1. All leaves are on the same level
  2. No node can contain more than *m* children
  3. All nodes except the root have at least keys $\lceil \frac{m}{2} \rceil - 1$
  4. The root is either a leaf node, or it has from 2 to m children

Example AVL Tree: Insert 17

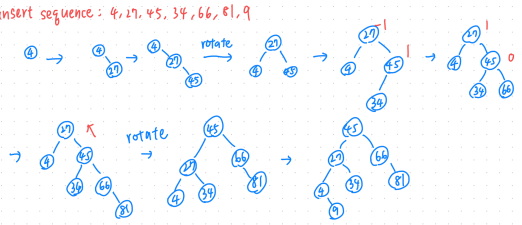
based on BST rule



Insert 5



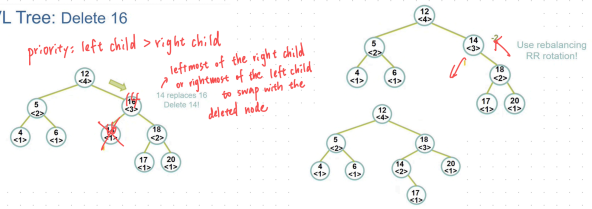
insert sequence: 4, 27, 45, 34, 66, 81, 9



AVL Tree: Delete 16

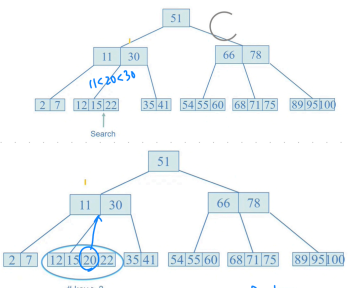
priority: left child > right child

leftmost of the right child or rightmost of the left child to swap with the deleted node



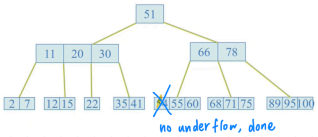
Example of Insertion

Insert 20

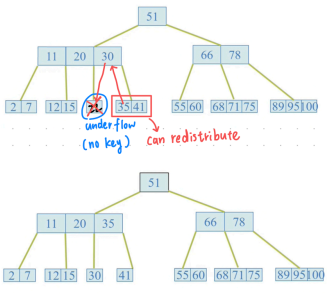


Example of Deleting A Leaf

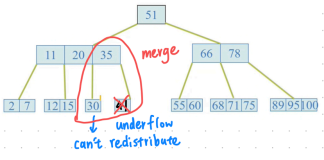
Case 1 (Just delete): delete 54



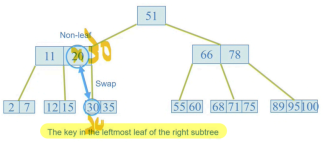
Case 2 (Redistribute): delete 22



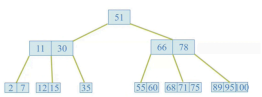
Case 3 (Merge): delete 41



Case 4 (non-leaf): delete 20



Case 4 (non-leaf): delete 20



B-tree: Flow Chart of Deletion