CS2351
Data Structures

Lecture 15:
B-tree

About this tutorial

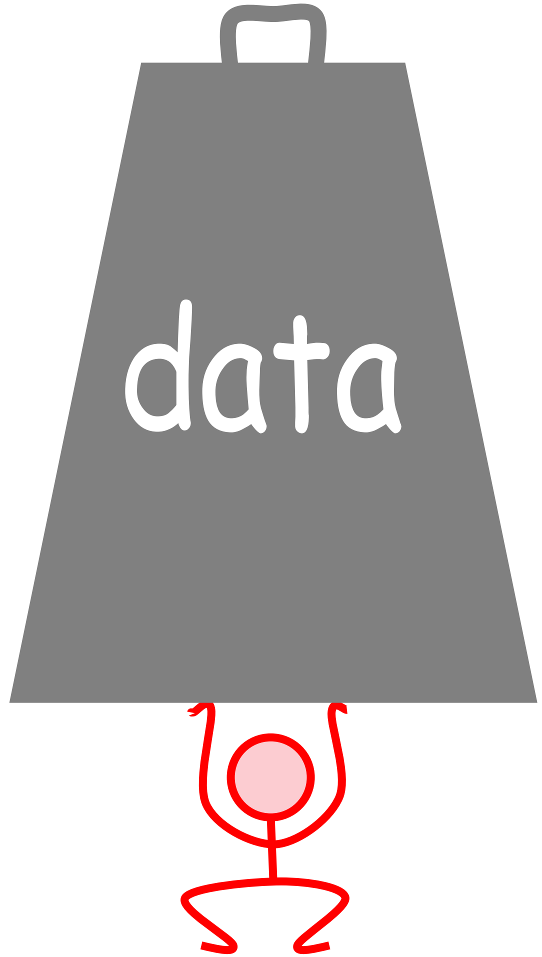
- Introduce External Memory (EM) Model
  - Proposed by Aggarwal and Vitter (1988)

- How to perform searching and updating efficiently when data is on the hard disk ?
  - B-tree, B^+-tree, B*-tree

The EM Model

Dealing with Massive Data

- In some applications, we need to handle a lot of data
  - so much that our RAM is not large enough to handle
- Ex 1: Sorting most recent 8G Google search requests
- Ex 2: Finding longest common patterns in Human and Mouse DNAs



Dealing with Massive Data

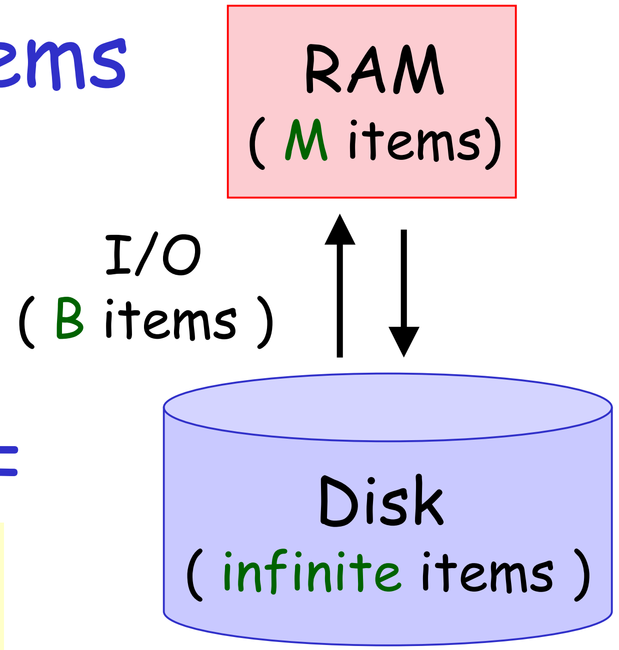
- Since RAM is not large enough, we need the hard-disk to help the computation
- Hard-disk is useful:
  1. can store input data (obvious)
  2. can store intermediate result
- However, there are new concern, because accessing data in the hard-disk is much slower than accessing data in RAM

EM Model [Aggarwal-Vitter, 88]

- Computer is divided into three parts:
  CPU, RAM, Hard-disk
- CPU can work with data in RAM directly
  - But not directly with data in hard-disk
- RAM can read data from hard-disk, or write data to hard-disk, using the I/O (input/output) operations

EM Model [Aggarwal-Vitter, 88]

- Size of RAM = M items
- Hard-disk is divided in pages
  - Size of a disk page = B items
- In one I/O, we can
  - read or write one page



- Complexity of an algorithm =

  number of I/Os used

➔ That means, CPU processing is free !

Test Our Understanding

- Suppose we have a set of N numbers, stored contiguously in the hard-disk
- How many I/Os to find max of the set?

  Ans. $O(N/B)$ I/Os

- Is this optimal ?

  Ans. Yes. We must read all #s to find max, which needs at least N/B I/Os

B-tree

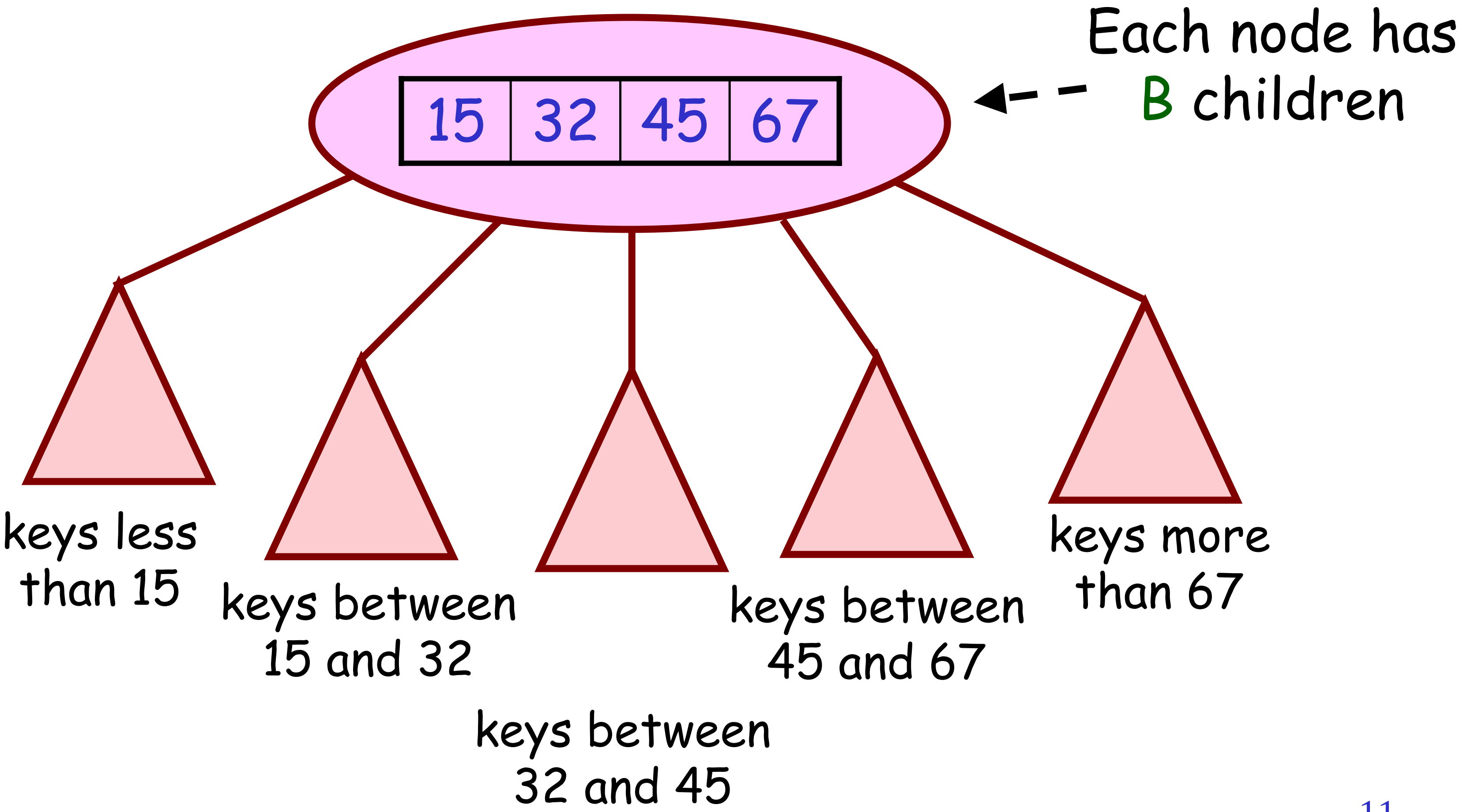
Search Tree in EM Model

- BST search needs O(log n) comparisons
  - This is optimal (why?)
  - Key idea of BST : each comparison reduces the search space by nearly half

- In EM model, each page contains B items
  - We can compare more things in 1 I/O
  - Can we take advantage of this to minimize search I/Os ?

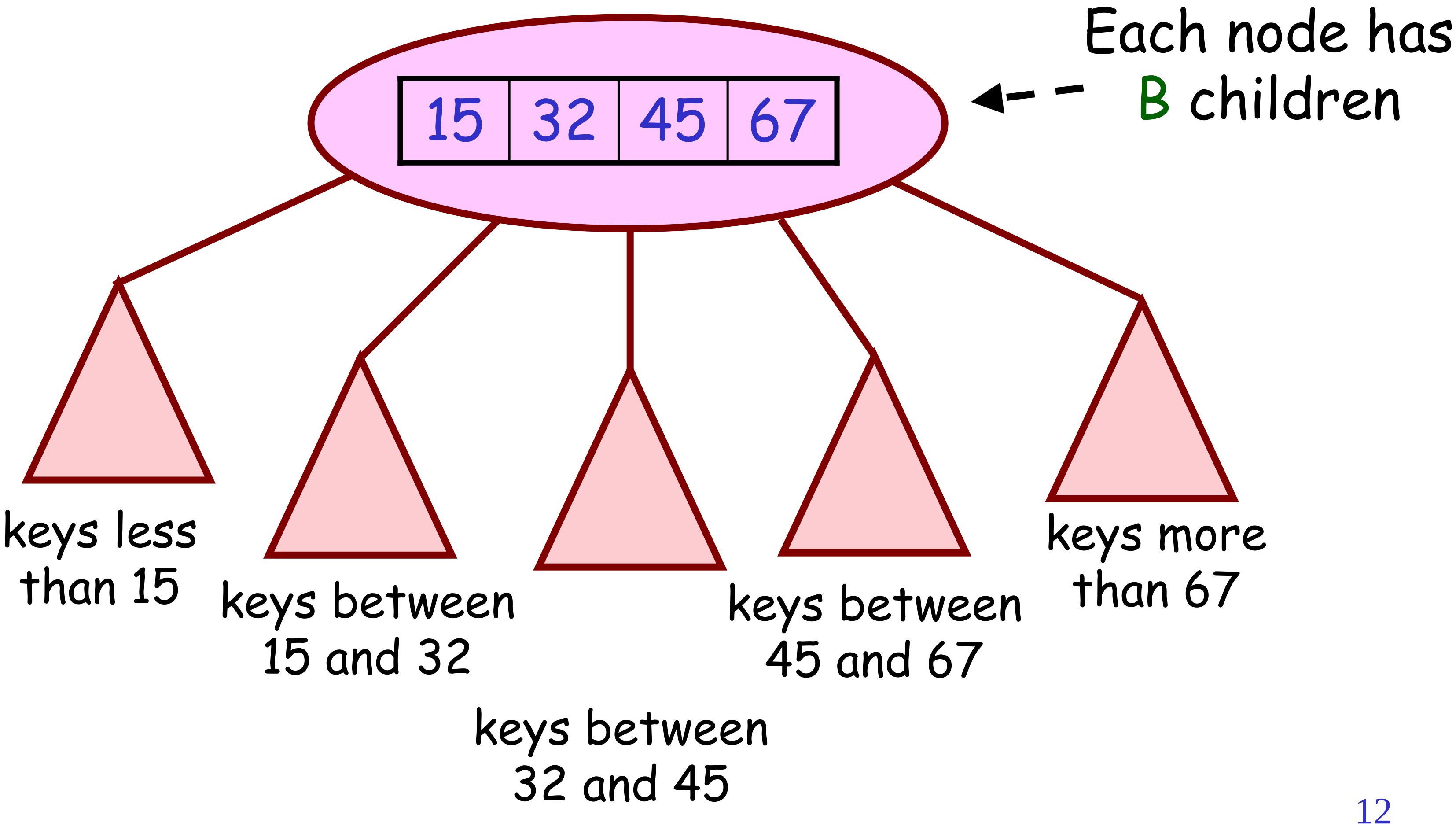
Search Tree in EM Model

- Yes ! Let us use a degree-B tree



Search Tree in EM Model

- Search can be done in $O(\log_B n)$ I/Os



B-tree

- We now introduce B-tree which uses the above concept to support fast searching
  - But in order to support fast updating, the definition is slightly modified
- Precisely, B-tree is a search tree, where

1. Root has 2 to B children ; each other internal node has B/2 to B children
2. All leaves are on the same level

Flexibility in node degree allows fast updating

B-tree

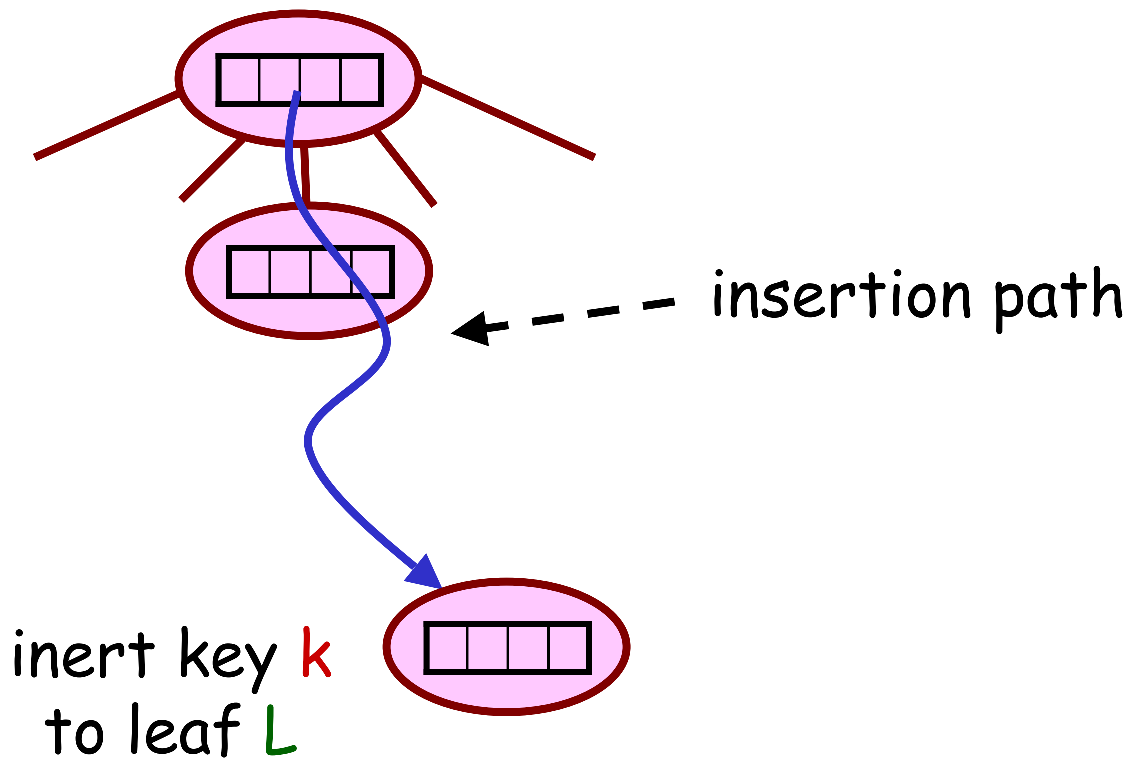
- Based on the definition of B-tree
  - What is the height of the tree ?
  - How many I/Os to search ?
  - Is it optimal ? Why ?

- Next, we describe how to perform fast updates, which is done by two powerful operations : merge and split

Updates in a B-tree

Insertion

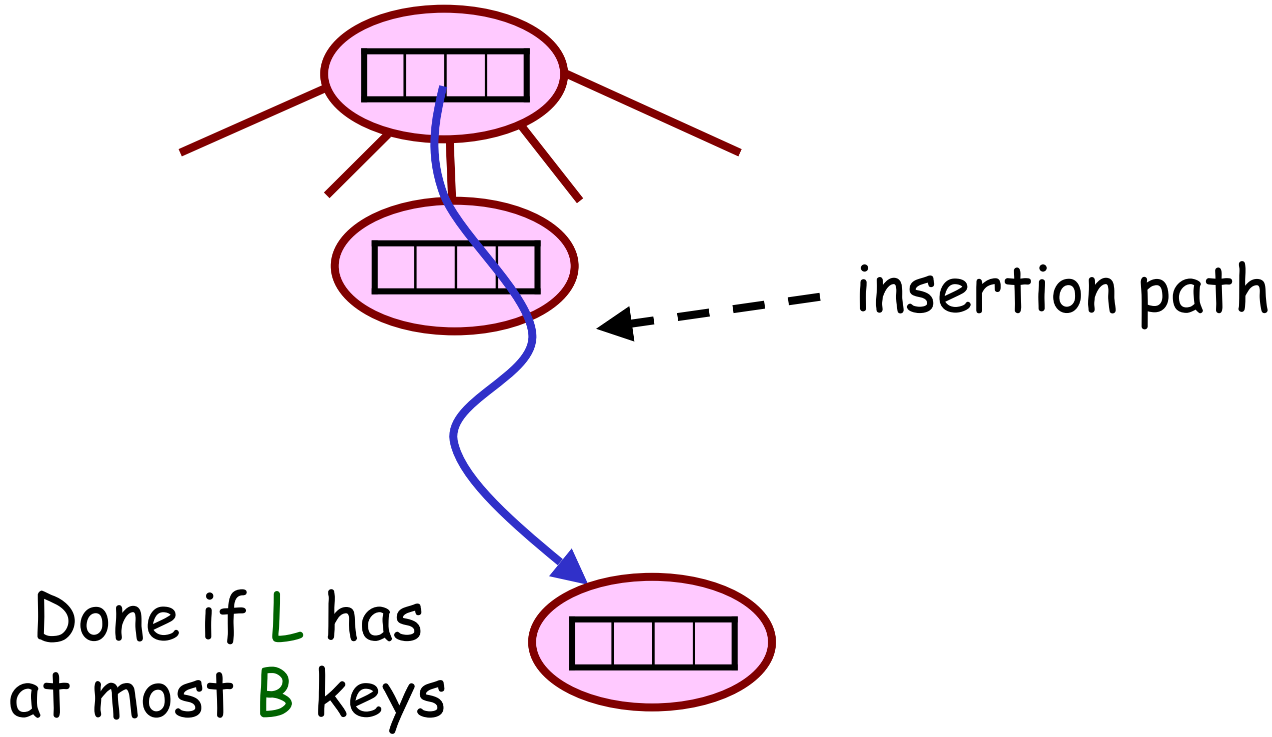
- Insertion of a key k first inserts k to the leaf L that should contain it



Insertion : Case 1

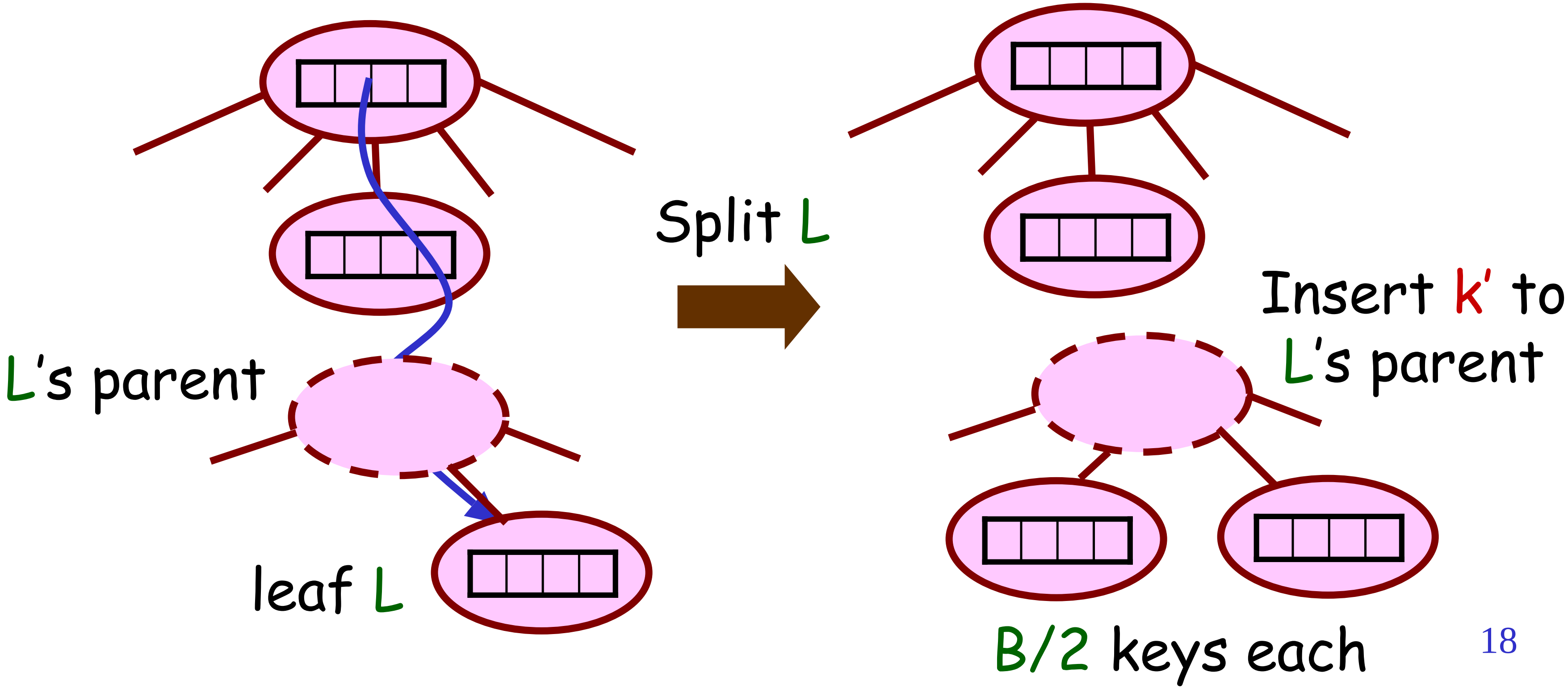
- If the leaf L still has at most B keys

  → Done !



Insertion : Case 2

- If the leaf L now has B+1 keys (overflow)
  → Split L into two nodes
  → Insert middle key k' to parent of L



Insertion : Case 2

- If L's parent now has at most B children
  → Done
- Else if L's parent now overflows
  → Recursively split and insert middle key to its parent

- Special case: If the current root is split into two nodes, we create a new root and joins it to the two nodes

Insertion Performance

In both cases :

- The number of I/Os is $O(\log_B n)$
- The number of operations is $O(B \log_B n)$
- All properties of B-tree are maintained after insertion

Remarks :

Tree height is increased only when the root is split

Deletion

- Deletion of a key k is done as follows :

1. If k is in some leaf L, delete k ;
2. Else, k is in some node X.

   → We locate k's successor s which must be in some leaf L; (why?)

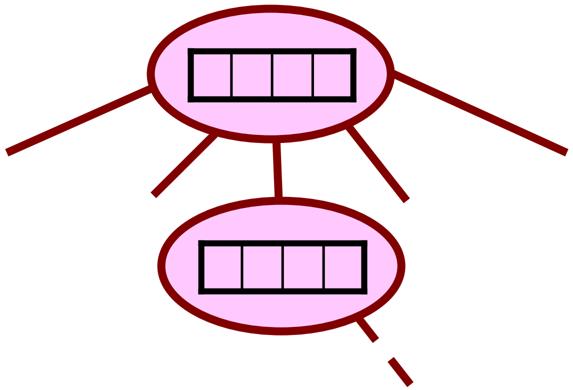
   → Replace k by s in the node X, and delete s from the leaf L

➔ So we can assume that we always delete a key from some leaf L

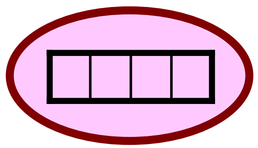
Deletion : Case 1

- If the leaf L still has at least B/2 keys

  → Done !



Done if L has
at least B/2 keys



Deletion : Case 2

- If leaf L now has B/2 - 1 keys (underflow) → Merge L with a sibling L'
- Now, two sub-cases may happen :

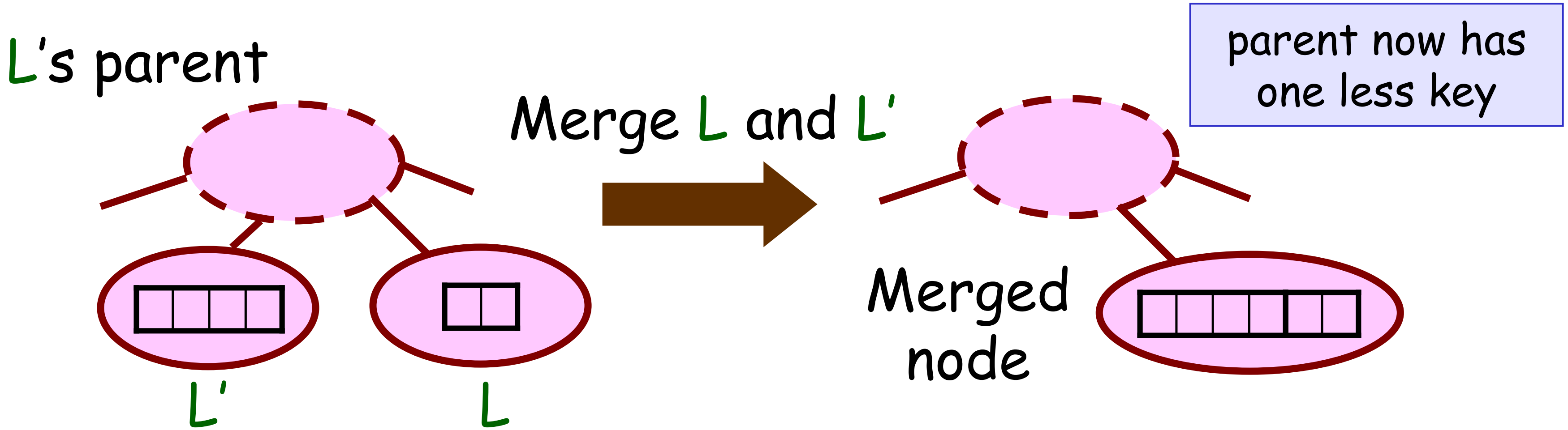
  Case 2.1 : overflow occurs

  - Split the merged node, and update the key in the parent → Done !

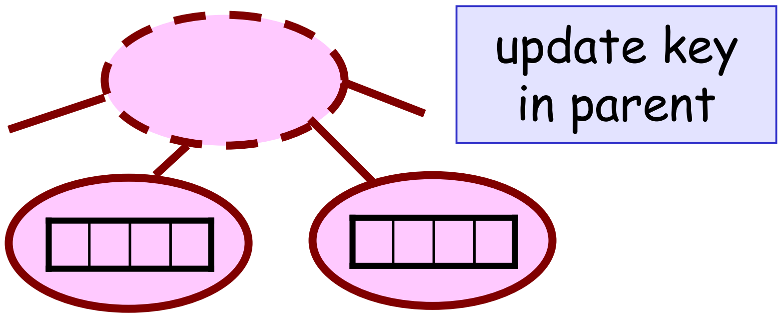
  Case 2.2 : no overflow

  - Delete a key from L's parents
  - Recursively update by merge and split

Deletion : Case 2

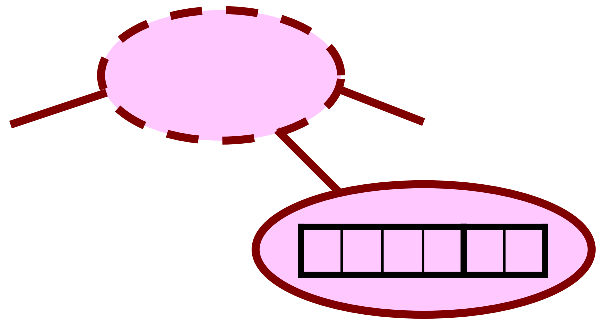


Case 2.1 : overflows



Split merged node ➔ each has B/2 to B keys

Case 2.2 : no overflow



Recursively delete key in parent

Deletion Performance

In both cases :

- The number of I/Os is $O(\log_B n)$
- The number of operations is $O(B \log_B n)$
- All properties of B-tree are maintained after insertion

Remarks : The root is deleted when it has only one child → this child becomes new root → Tree height decreased by 1

Final Remarks

- When B = O(1), each operation is done in O(log n) time (We need B $\geq$ 3. Why?)
  - When B = 3, the corresponding B-tree is called a 2-3 tree
  - When B = 4, it is called a 2-3-4 tree, which is equivalent to a Red-Black tree

- B-tree has two famous variants, B^+-tree and B*-tree (check wiki for more info)