Main Memory and the CPU Cache

B Trees

B Trees

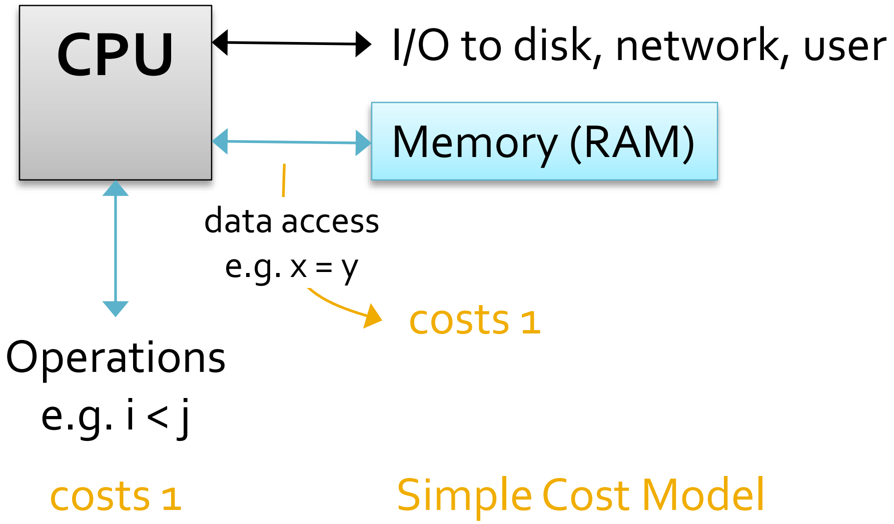
- CPU cache
- Unrolled linked lists
- B Trees

CPU Cache

Main Memory and O Notation

- Our model of main memory and the cost of CPU operations has been intentionally simplistic
  - The major focus has been on determining the broad growth rate of running times of algorithms
    - i.e. O notation
- Program running time is determined by
  - Algorithm
  - Hardware speed
  - Programming quality

Simple Hardware Model



Reality

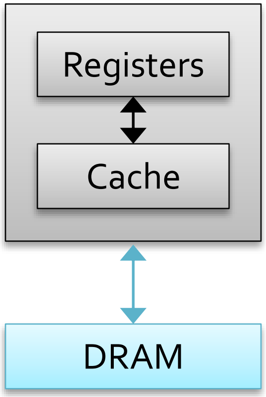
time to access an *int* in RAM

is more than 200 times

time to compare two *int*s in registers

CPU Cache

- CPU storage consists of
  - Registers
  - CPU cache
- Data to be used must be in a register
  - If it is not in a register
  - Request it from memory
- To request data from memory
  - First check the CPU cache
    - If it is in the cache (a cache *hit*) takes 2 … 20 cycles
    - If it is not in the cache (a *miss*) takes approximately 200 cycles



A cycle is the time of a CPU operation

Cache Lines

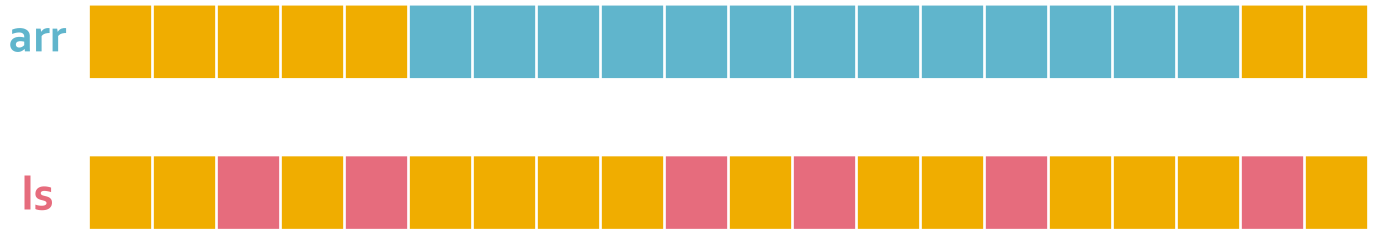
- Sections of the CPU cache are referred to as *cache lines*
  - The typical cache line size is 64 bytes
- When data is read from DRAM an entire block of data, the cache line, is copied into the cache
  - So if a 4 byte integer was to be read an entire 64 byte block is copied into the cache
    - This is fast as bandwidth is high
- Data remains in a cache until it is replaced
  - By the cache replacement algorithm
- Implication
  - Data related to a recent request may be copied to the cache if it was physically close to the requested data

CPU Data Cache Effects

- Consider two loops over an array
  - `int* arr = new int[1024 * 1024];`
- Loop 1: `for(int i=0; i < n; i++) arr[i] *=3;`
- Loop 2: `for(int i=0; i < n; i+=16) arr[i] *=3;`
- Loop 1 performs sixteen times as much work as loop 2
  - They are both O(n)
- Loop 1 does not take 16 times as long to process
  - Actual times vary but
  - Loop 1 and loop have the same number of RAM accesses
    - i.e. the same number of cache misses

Example: List Traversal

- Consider two different implementations of a list of integers
- Case 1: the list is stored in an array, *arr*
  - Elements of *arr* are contiguous to their neighbours
- Case 2: the list is stored in a linked list, *ls*
  - Elements of *ls* are fragmented – they are physically separate from each other in main memory



List Traversal: Array

- Assume the following costs
  - 4 cycles to read from a cache line
  - 200 cycles to read from DRAM
- Time to read from the array
  - 16 elements are read into a cache line at a time
  - The total number of cycles for each 16 elements
    - 200 (read into cache) + 64 (read from cache) = 264
    - Approximately 16 cycles per element

List Traversal: Linked List

- The list is fragmented over main memory
  - List elements are not contiguous
  - The probability that related elements are read into the same cache line is small
  - Therefore every read takes 200 cycles
    - 12.5 times slower than the array
- In practice arrays may be even faster than lists
  - The OS may use a *pre-fetching algorithm*
  - Where it predicts which area of main memory will be read next and fetches it before it is requested

Cache Aware Algorithms

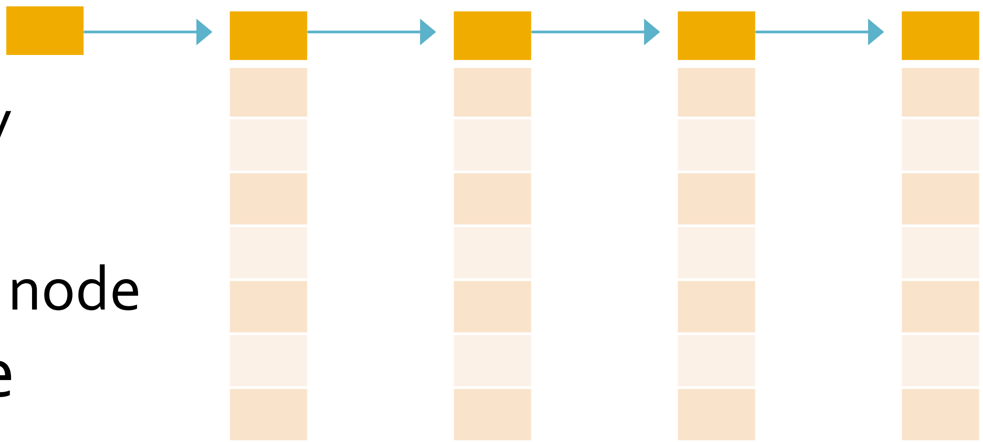
- *Cache aware* data structures and algorithms are designed to account for cache effects
  - Run time is improved by reducing the number of cache misses
  - We will look at two examples
    - Unrolled linked lists
    - B Trees
- The run-time of *cache oblivious* algorithms applies is unaffected by cache structure

Unrolled Linked Lists

Unrolled Linked List Introduction

- Linked lists are dynamic data structures
  - Adding a new element when the list is full is very fast
    - O(1)
  - In contrast to adding a new element to an array that is full
    - O(n)
- But traditional linked lists are cache oblivious
  - Since elements may be fragmented across main memory
  - Particularly when the list is constructed node by node over time
- Solution make a list of arrays

Unrolled Linked List



- List nodes contain
  - A partially filled array of elements
  - A pointer to the next node
- Each node has a size and capacity
  - The size is the current number of elements
  - The capacity is maximum number of elements

Traversing Unrolled Linked Lists

- To traverse an unrolled linked list
  - Traverse the linked list and at each node traverse the occupied part of the array
  - There are frequently cache misses visiting a new node
    - But few or zero cache misses for visiting each element of a single node's arrat
- Each array is at least half full
  - Maintained by the insertion and removal algorithms

A Comparison

- Assume the following
  - 5 cycles for a cache hit
  - 200 cycles for a cache miss
    - Traversal time in CPU instructions = 5n + 200 * cache misses
  - Item size is 4 bytes
- We will compare three structures
  - Array
  - Simple linked list
  - Unrolled linked list

Array

- The number of cache misses for an array is
  - $4n / 64 = n / 16$
  - i.e. the number of times data has to be read into a cache line
  - Ignores the effects of pre-fetching
- Traversal time in CPU instructions
  - $5n + 200n / 16 \approx 17.5n$

Simple Linked List

- The number of cache misses for a simple linked list is the number of nodes
  - n
- Traversal time in CPU instructions
  - $5n + 200n = 205n$

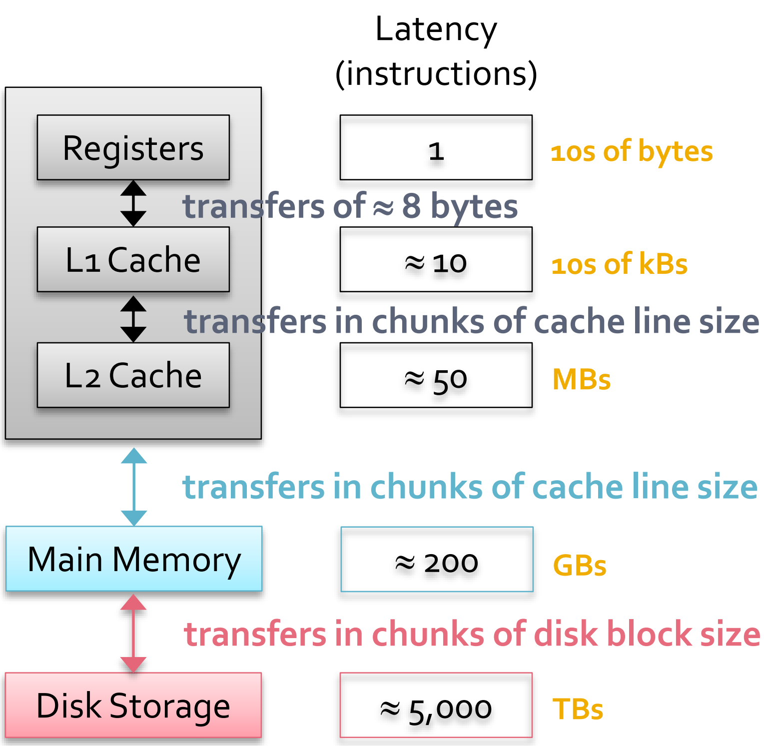
Unrolled Linked List

- The number of cache misses for an unrolled linked list is affected by two factors
  - Occupancy of the arrays, assume ½ full
  - Additional overhead per node, assume
    - 8 byte pointer
    - 4 byte size variable
    - 4 byte metadata
  - Cache misses = $(2n/16) * ((4*16 + 16) / 64) = 5n / 32$
- Traversal time in CPU instructions
  - $5n + 200n / 7 \approx 34n$

Summary

- Traversal time in CPU instructions
  - Array: $17.5n$
  - Unrolled linked list $34n$
  - Simple linked list $205n$
  - All are $O(n)$ with different leading constants
- All calculations are both approximate and simplistic and assume the worst case
  - But do indicate actual performance

Memory Hierarchy Revisited



B Trees

B Tree Introduction

- External indexes are optimized to minimize disk reads
  - Since reading disk from a hard disk is very slow
    - Many times slower than reading data from DRAM into a register
  - Known as B trees
  - They can be adapted to work with cache lines
- B trees have two desirable properties
  - They are multiple level indexes that maintain as many levels as are required for the file being indexed
  - Space on tree blocks is managed so that each block is at least ½ full

B Tree Structure

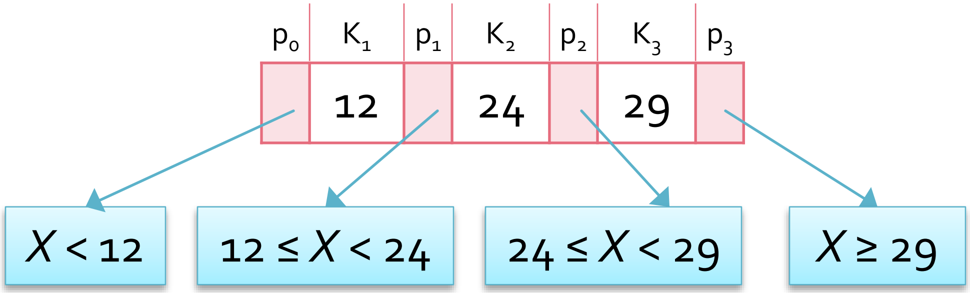
- B trees are *balanced* structures
  - All paths from the root to a leaf have the same length
  - Most B trees have a small number of levels
    - Any number of levels is possible
- B trees are similar to binary search trees
  - Except that B tree nodes have more than two children
  - That is, they have greater *fan-out*
  - In an exterior structure a single B tree node maps to a single disk block
    - As a main memory structure nodes map to cache lines

B+ Tree Node Structure

- We will look at a variant of B trees called B+ trees
- The number of data entries in a node is determined by the size of the search key
  - Up to *n* search key values and *n* + 1 pointers
  - Choose *n* to be as large as possible while still allowing *n* search keys and *n* + 1 pointers to fit in a cache line
  - *n* is therefore determined by the size of
    - Search key values
    - Pointers
    - Cache lines

Interior Nodes

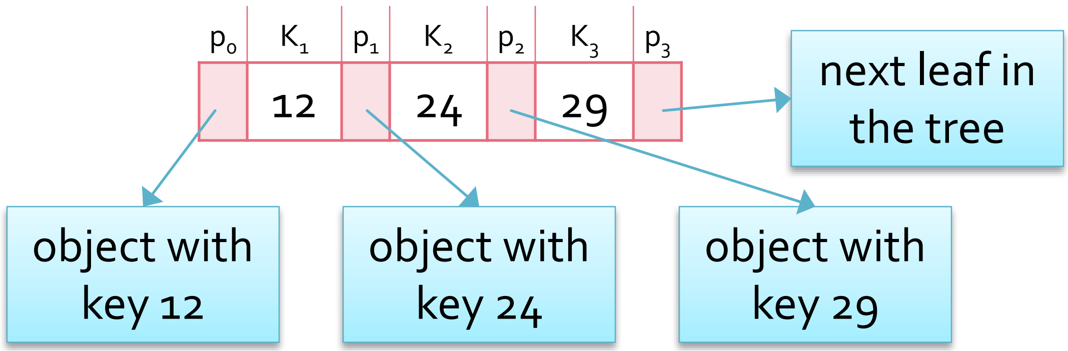
- In interior nodes, pointers point to next level nodes
  - Label the search keys K_1 to K_n, and pointers p_0 to p_n
    - Pointer p_0 points to nodes whose search key values are less than K_1
    - Other pointers, p_i, point to nodes with search keys greater than or equal to K_i and less than K_{i+1}
  - An interior node must use at least $\lceil (n+1)/2 \rceil$ *pointers*



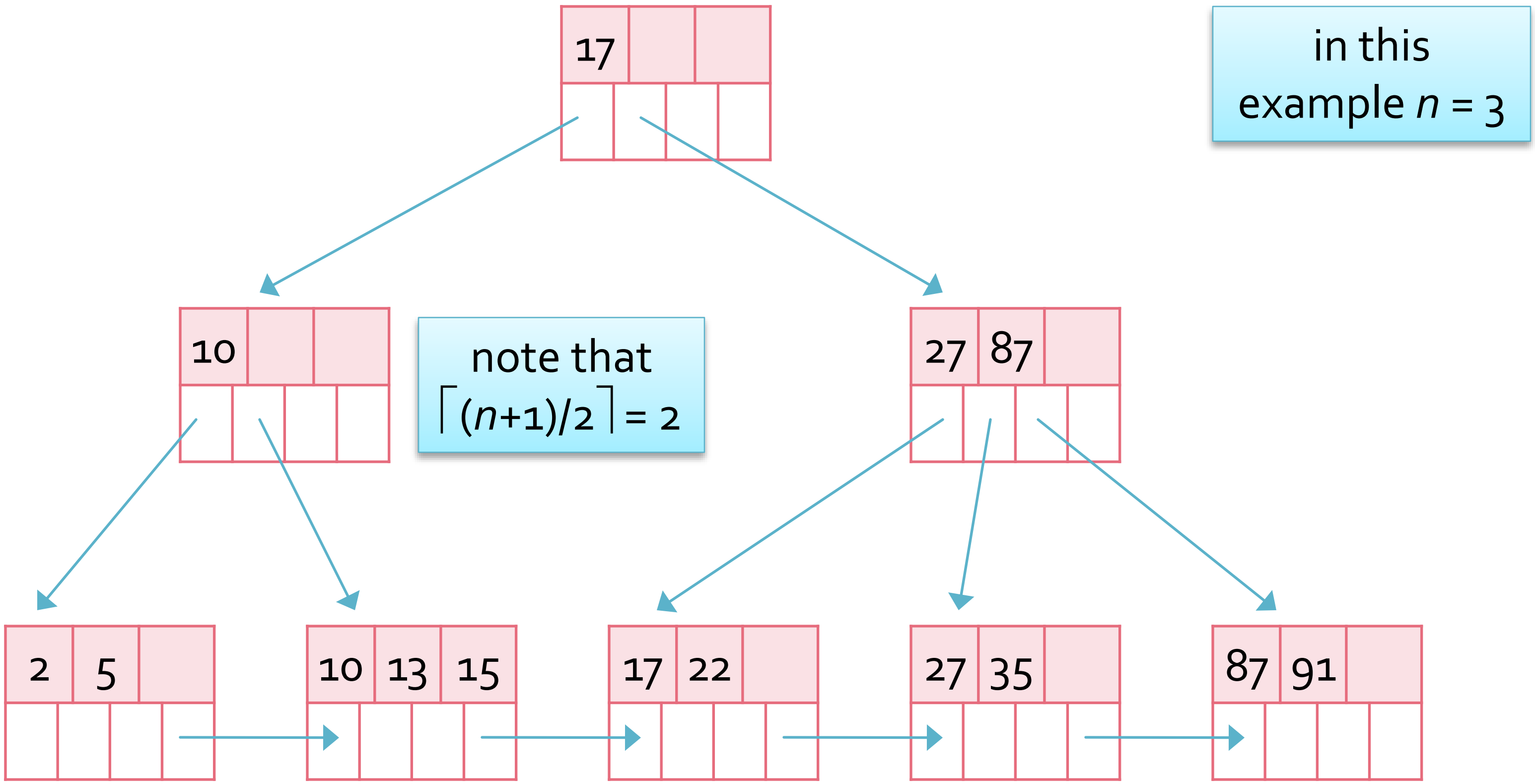
p_i points to subtrees with search key values, X

Leaf Nodes

- Leaf nodes contain data or pointers to data
  - The leaf nodes contain the keys in order
  - The left most *n* pointers point to data objects
    - A leaf node must use at least $\lfloor (n + 1)/2 \rfloor$ of these pointers
    - i.e. contain at least $\lfloor (n + 1)/2 \rfloor$ key values
  - The right most pointer points to the next leaf



Example B+ Tree



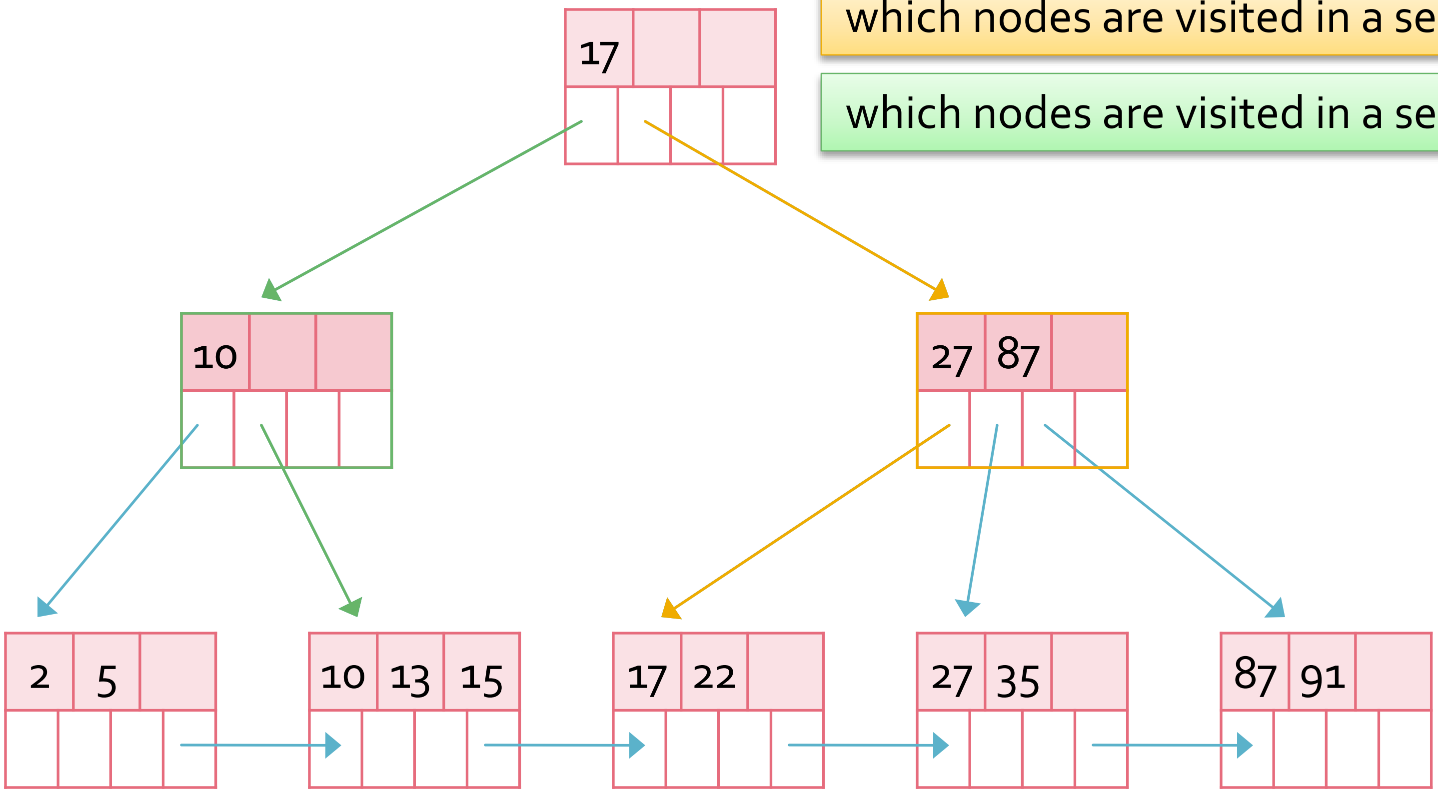
Searching a B+ Tree

- The B+ tree search algorithm is similar to a binary search tree
  - To search for a value *K* start at the root and end at a leaf
  - If the node is a leaf and the i^{th} key has the value *K* then follow the i^{th} pointer to the data
  - If the node is an interior node follow the appropriate pointer to the next (interior or leaf) node
- Searching a B+ tree visits a number of nodes equal to the number of levels of the tree (height + 1)
  - Each node entails a cache miss

B+ Tree Searches

which nodes are visited in a search for 22?

which nodes are visited in a search for 16?



Range Searches

- B+ trees can be used to retrieve a range of values range queries
- Assume we wish to retrieve values from *x* to *y*
  - Search the tree for the leaf that should contain value *x*
  - Follow the leaf pointers until a key greater than *y* is found
  - The tree can also be used to satisfy queries that have no lower bound or no upper bound

B+ Tree Insertions 1

- Insert the value in the appropriate place in a leaf node of the index
  - Use the search algorithm to find the leaf node
  - Insert value, if it fits the process is complete
- If the target leaf node is full then split it into two nodes
  - The first $\lceil (n + 1) / 2 \rceil$ values stay in the original node
  - Create a new node to the right of the original node with the remaining $\lfloor (n + 1) / 2 \rfloor$ values
  - Insert the first search key value from the new leaf and a pointer to the leaf in its parent node

B+ Tree Insertions 2

- Adding a value to an interior node may cause it to split
  - If so, after inserting a new entry there will be $n + 1$ keys and $n + 2$ pointers
  - The first $\lceil (n + 2) / 2 \rceil$ *pointers* stay in the original node
  - Create a new node with the remaining $\lfloor (n + 2) / 2 \rfloor$ pointers to the right of the original node
  - Leave the first $\lceil n / 2 \rceil$ keys in the original node and move the last $\lfloor n / 2 \rfloor$ keys to the new node
  - The remaining key 's value falls between the values in the original and new node
- This left over key is inserted into the parent of the node along with a pointer to the new interior node

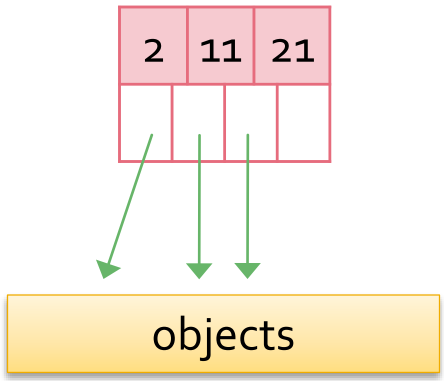
B+ Tree Insertions 3

- Moving a value to a higher, interior level of the tree, may again cause a split
  - The same process is repeated until no further splits are required, or until a new root node has been created
- If a new root is created it will initially have just one key and two children
  - So will be less than half full
  - This *is* permitted for the root (only)

B+ Tree Insertion Example

insert 2, 21 and 11

n = 3



the values are maintained in order in the index pages

Note that leaf nodes may just contain values, rather than pointers to values – it depends on what is being stored in the tree

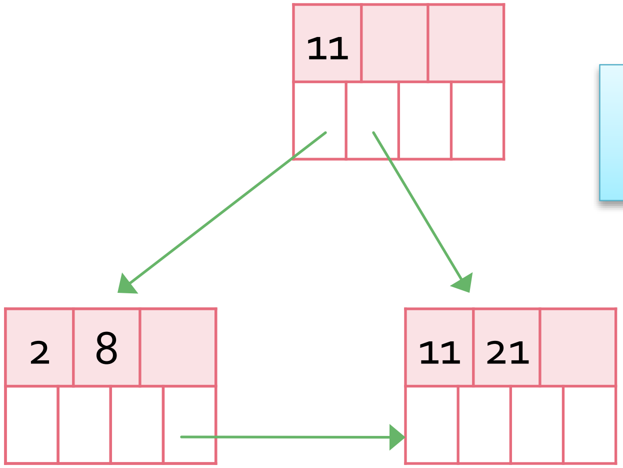
So a leaf might have this structure (only values):

2	11	21

B+ Tree Insertion Example

insert 8

n = 3



create new root with the first value of the new leaf node

create a new node with the last ½ of the values

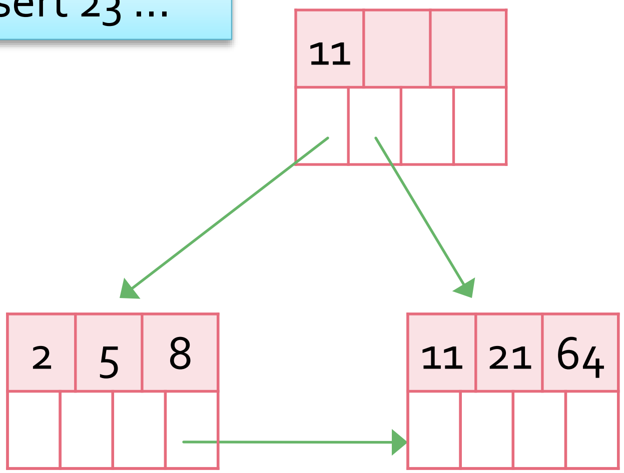
chain the new node to the original node

B+ Tree Insertion Example

insert 64, then 5

insert 23 …

n = 3

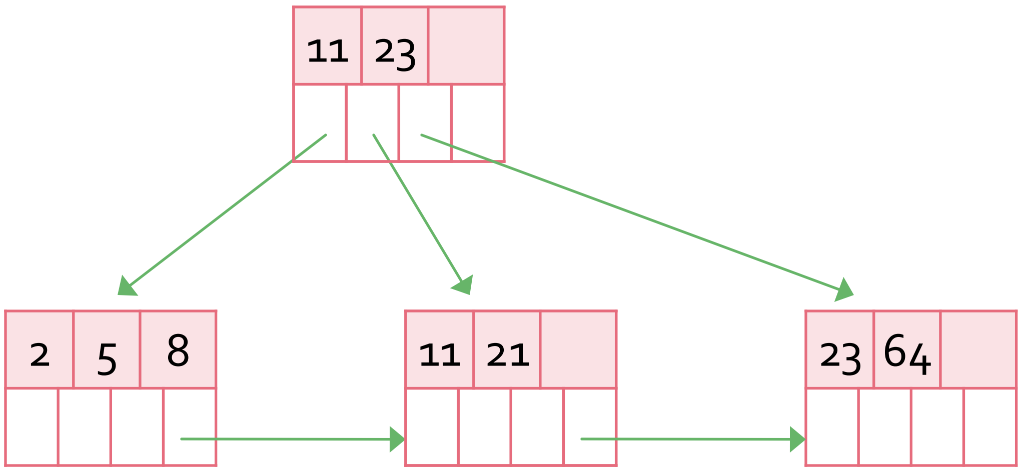


both leaf nodes are now full …

B+ Tree Insertion Example

... inserting 23 ...

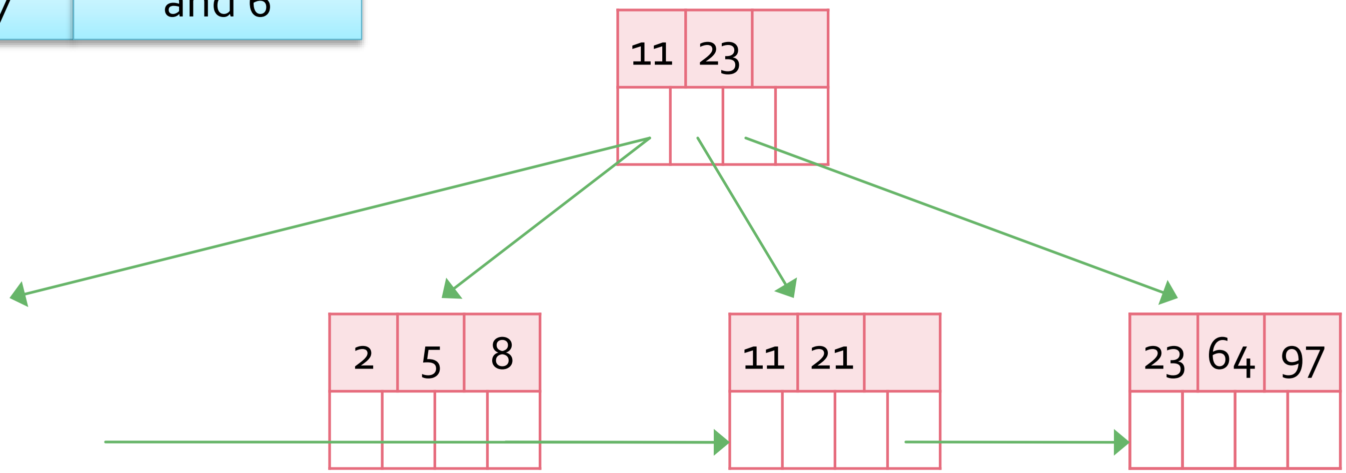
n = 3



B+ Tree Insertion Example

insert 97 | and 6

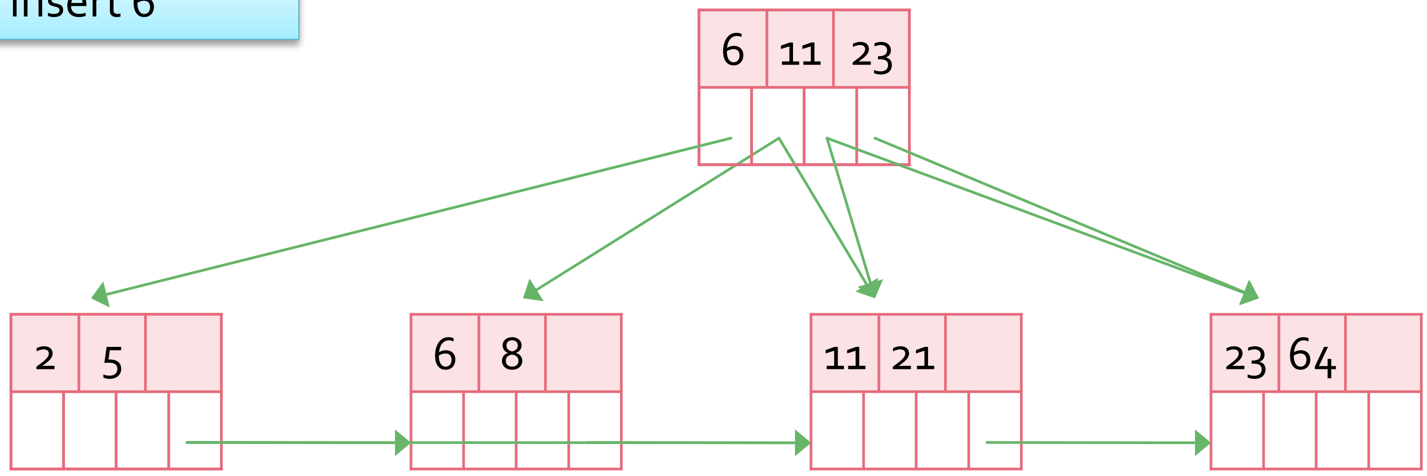
n = 3



B+ Tree Insertion Example

insert 6

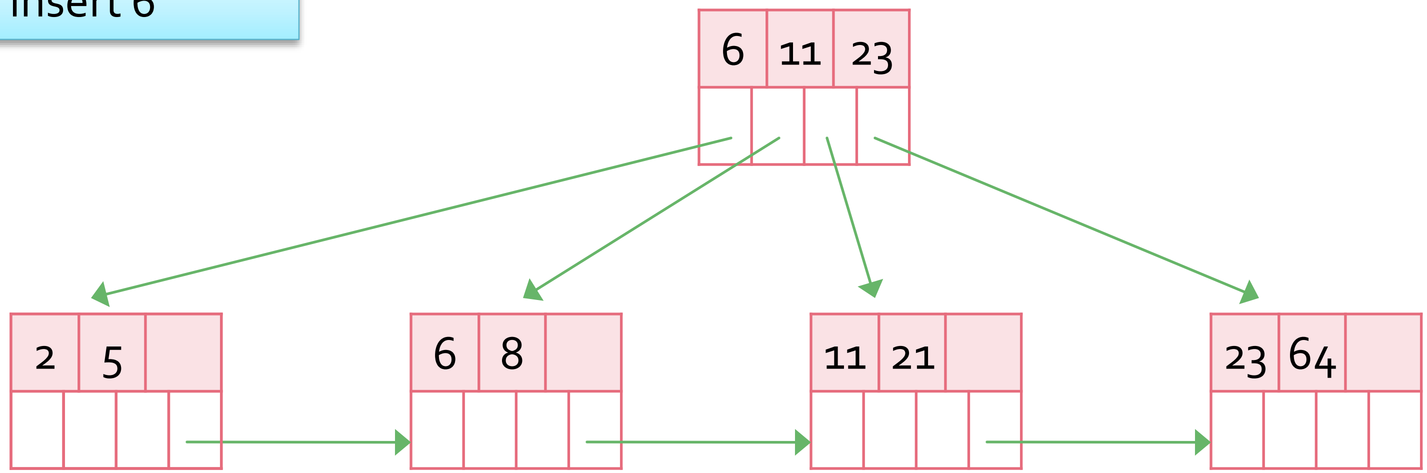
n = 3



B+ Tree Insertion Example

insert 6

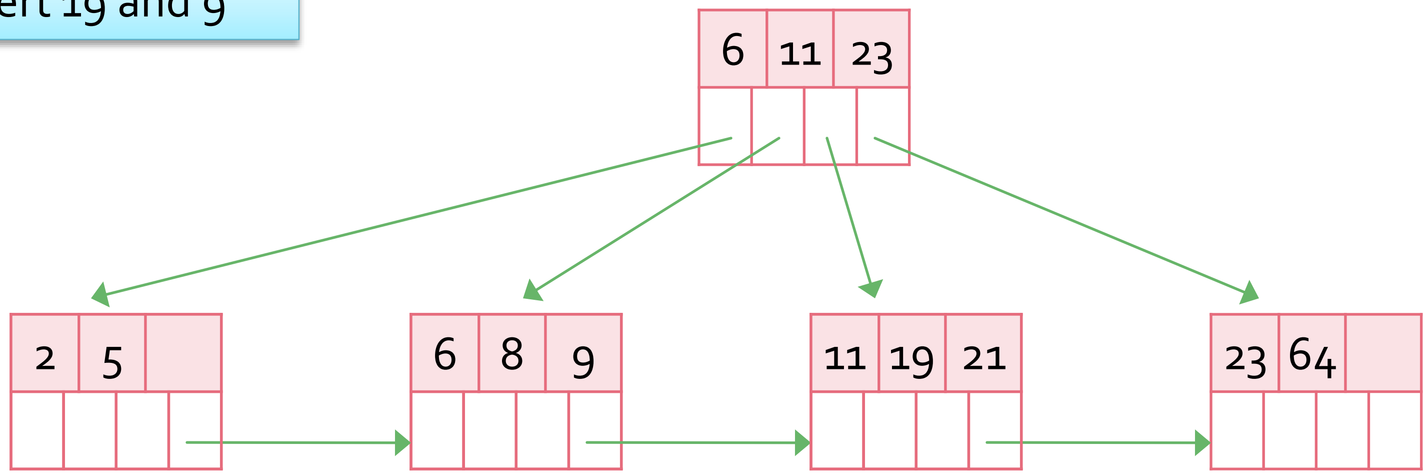
n = 3



B+ Tree Insertion Example

insert 19 and 9

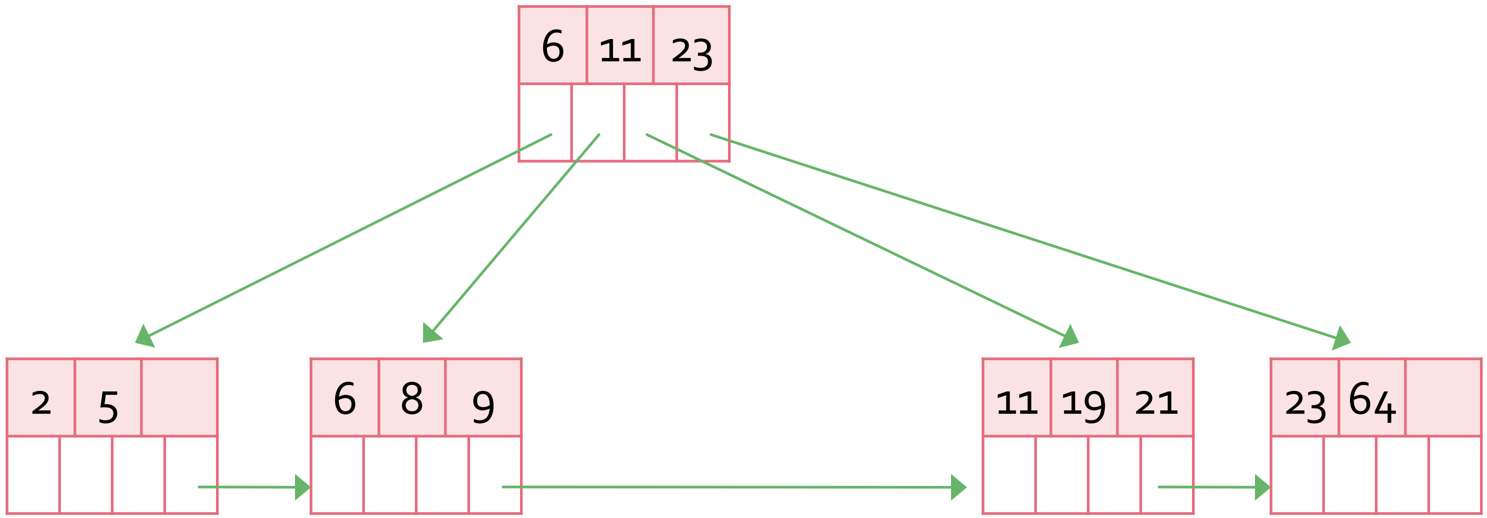
n = 3



B+ Tree Insertion Example

the same tree ...

n = 3



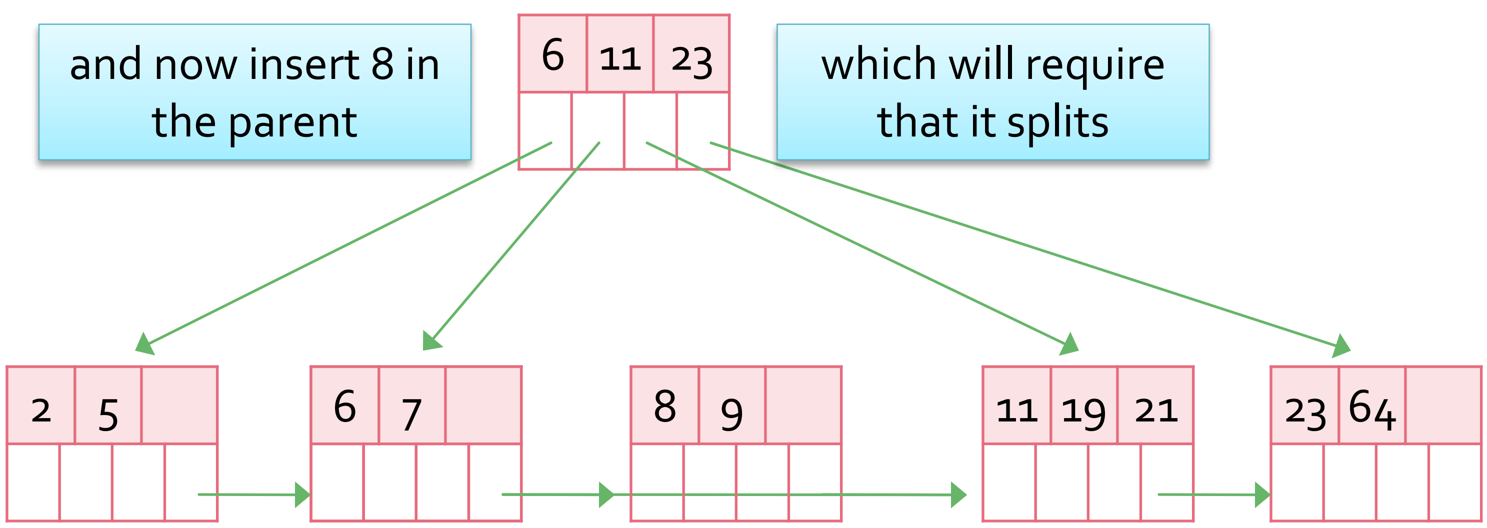
B+ Tree Insertion Example

insert 7

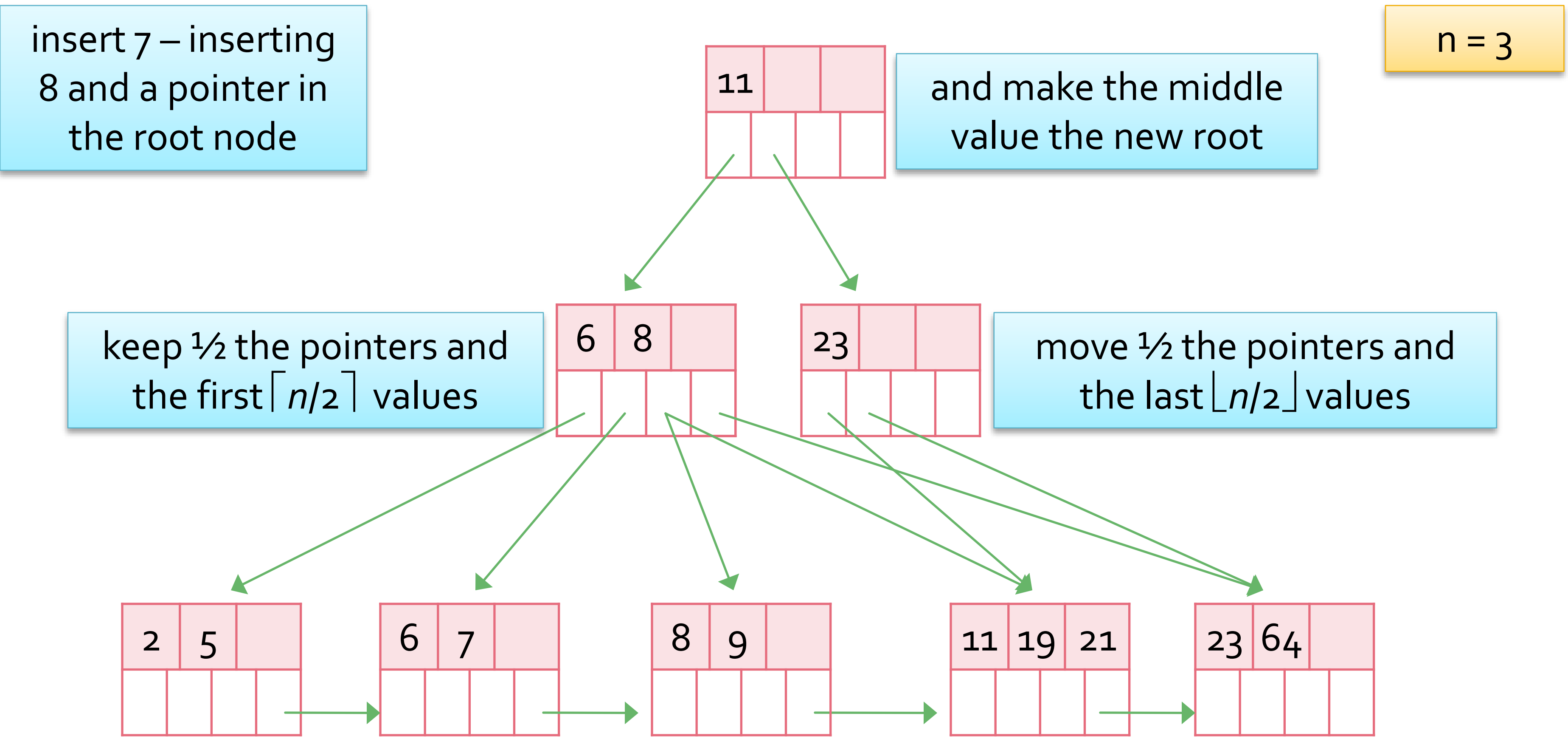
n = 3

and now insert 8 in the parent

which will require that it splits



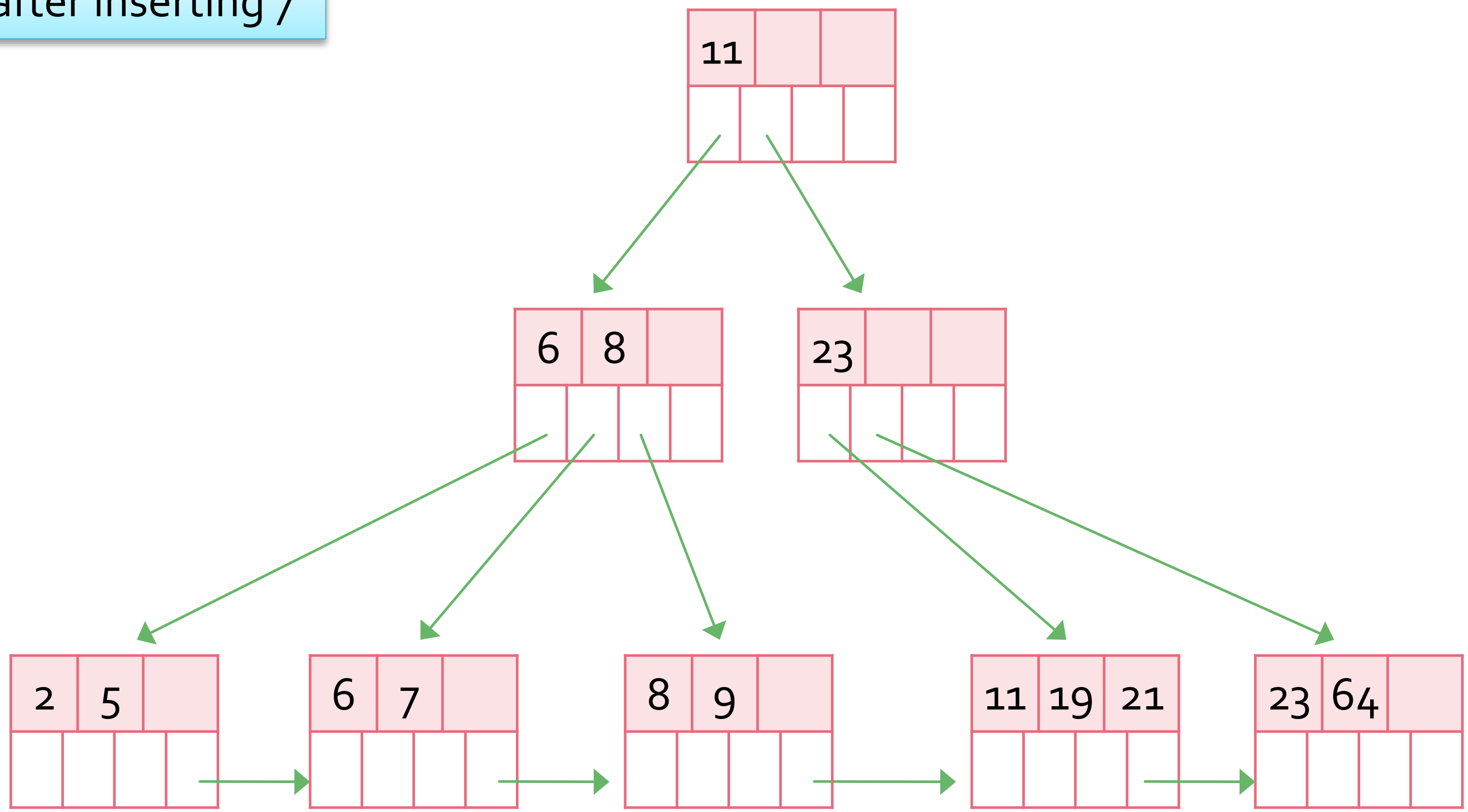
B+ Tree Insertion Example



B+ Tree Insertion Example

tree after inserting 7

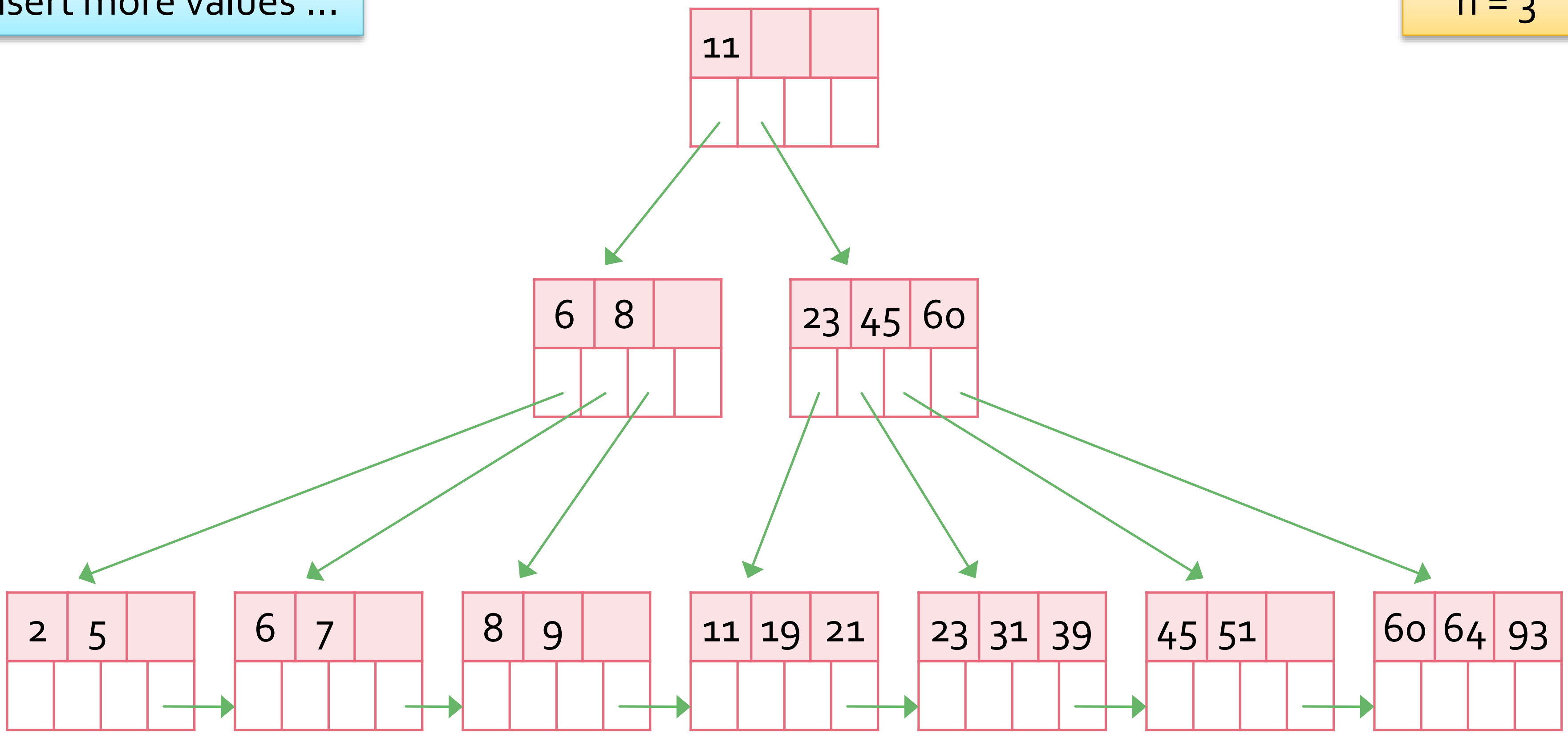
n = 3



B+ Tree Insertion Example

insert more values …

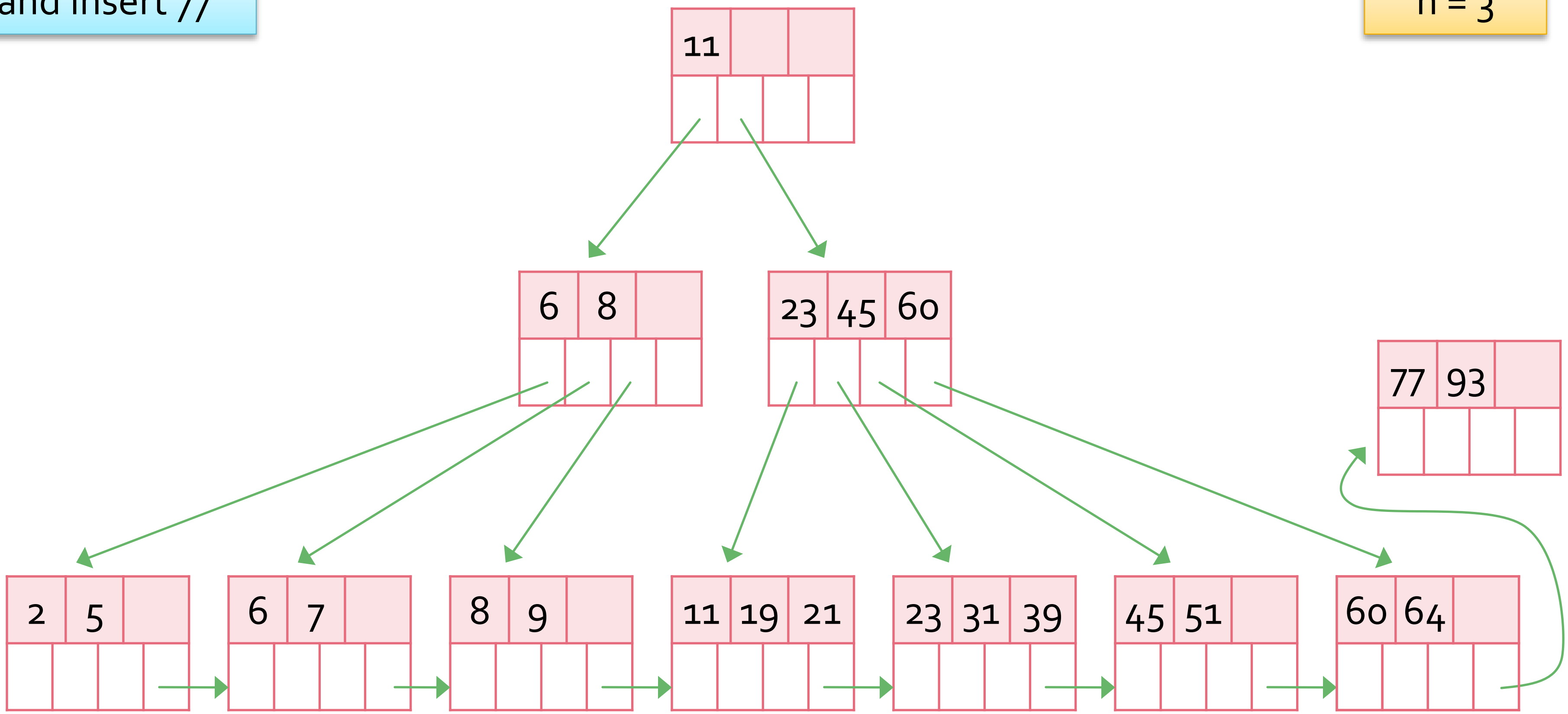
n = 3



B+ Tree Insertion Example

and insert 77

n = 3

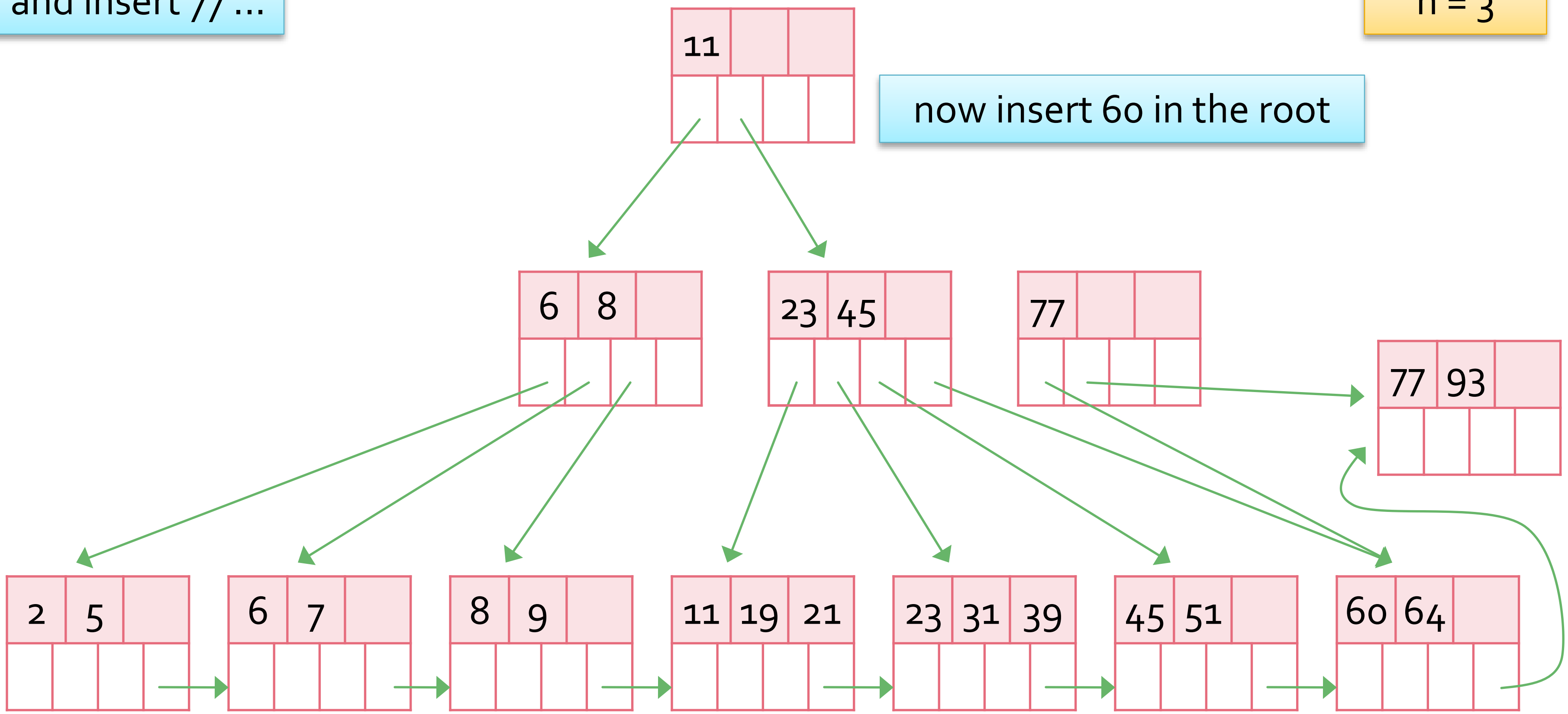


B+ Tree Insertion Example

... and insert 77 ...

n = 3

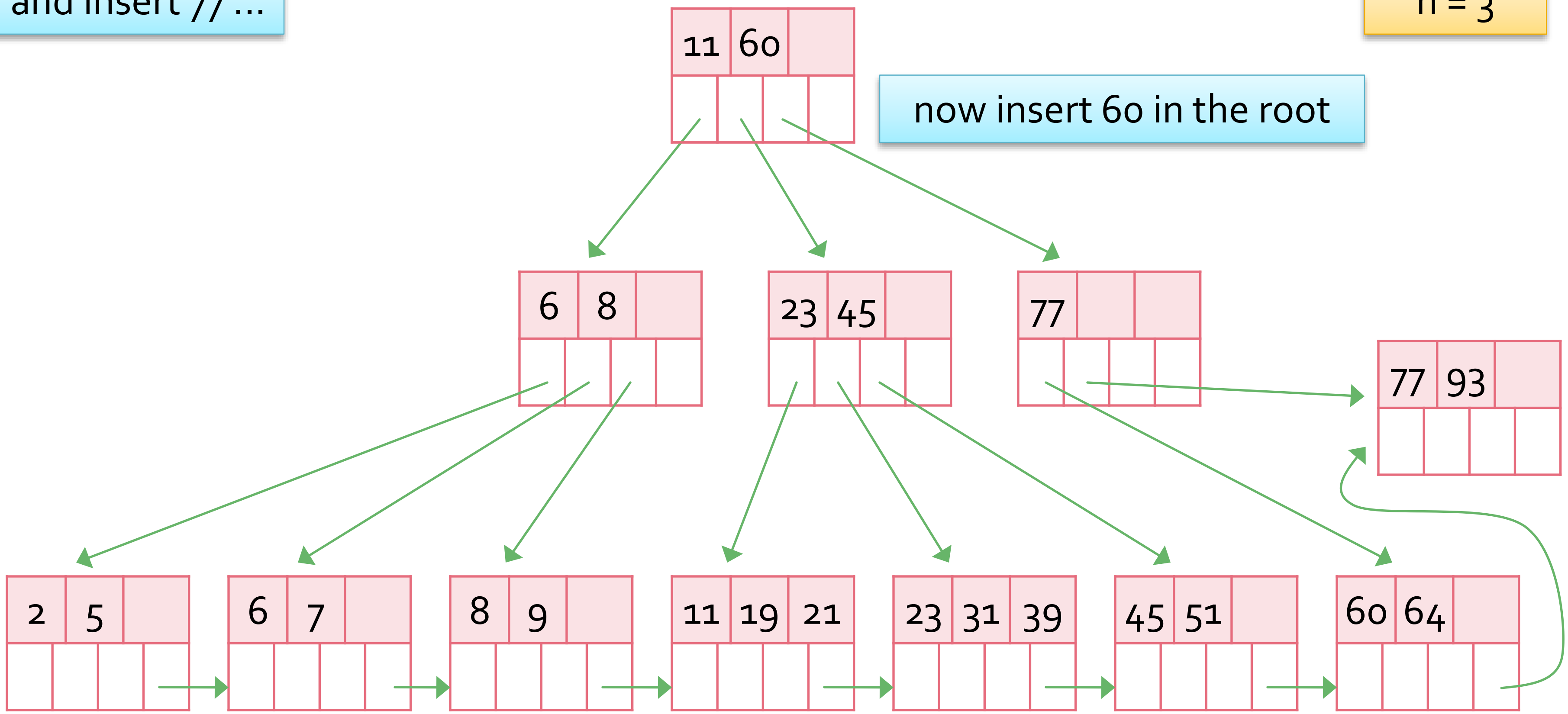
now insert 60 in the root



B+ Tree Insertion Example

... and insert 77 ...

n = 3



B+ Tree Removals

- Find the value in the leaf node and delete it
  - This may result in there being too few entries in the node
  - If so select an adjacent *sibling* of the node and
- Redistribute values between the two nodes
  - So that both nodes have enough entries
  - If this is not possible
- Coalesce the two nodes
  - Delete the appropriate value and pointer in the parent node

Redistributing Values

- A value and pointer are removed from an adjacent sibling and inserted in the node with insufficient entries
  - The sibling can be the left or the right sibling, although it makes a slight difference to the process
  - The chosen node *must be a sibling* to ensure that only a single parent node is affected
- After redistribution, one of the two nodes will have a different first search key value
  - The corresponding value in the parent node must be changed to this value
- If the node's sibling(s) have insufficient entries redistribution may not be possible

Coalescing Nodes

- When redistribution is not possible, two nodes can be combined (aka coalesced)
  - Keep track of the value in the parent between the pointers to the two nodes to be combined
  - Insert all of the values (and pointers) from one node into the other
  - Re-connect links between leaves (if the nodes are leaves)
- Make a recursive call to the removal process, removing the identified value in the parent node
  - This, in turn, may require non-leaf nodes to be coalesced

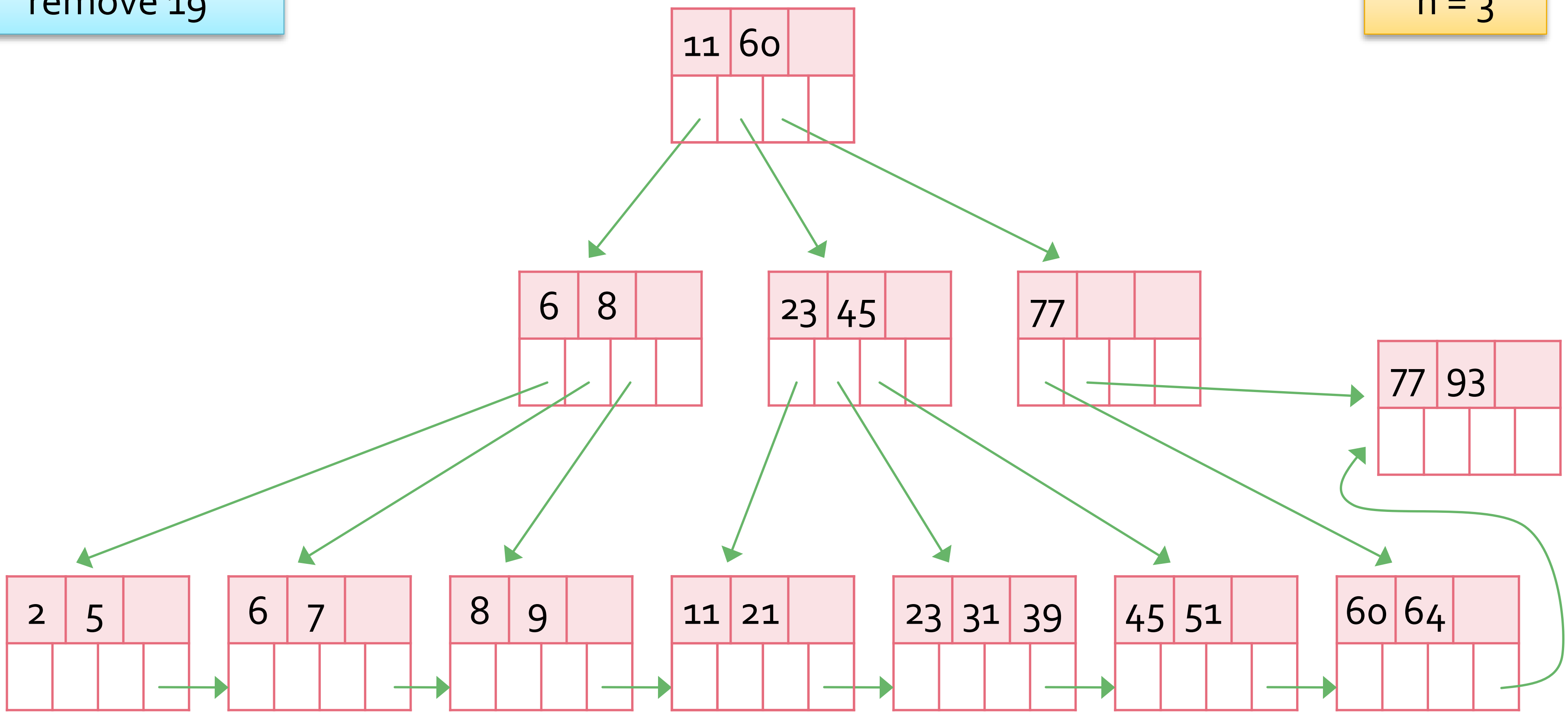
B+ Tree Removals Notes

- The removal algorithm requires a *choice* to be made between siblings
  - Such a choice has to be implemented in the algorithm
- Coalescing nodes requires more work
  - It may result in making changes up the tree, but
  - The tree height may be reduced
- Redistributing nodes requires less work, but does not impact the height of the tree

B+ Tree Removal Example 1

remove 19

n = 3

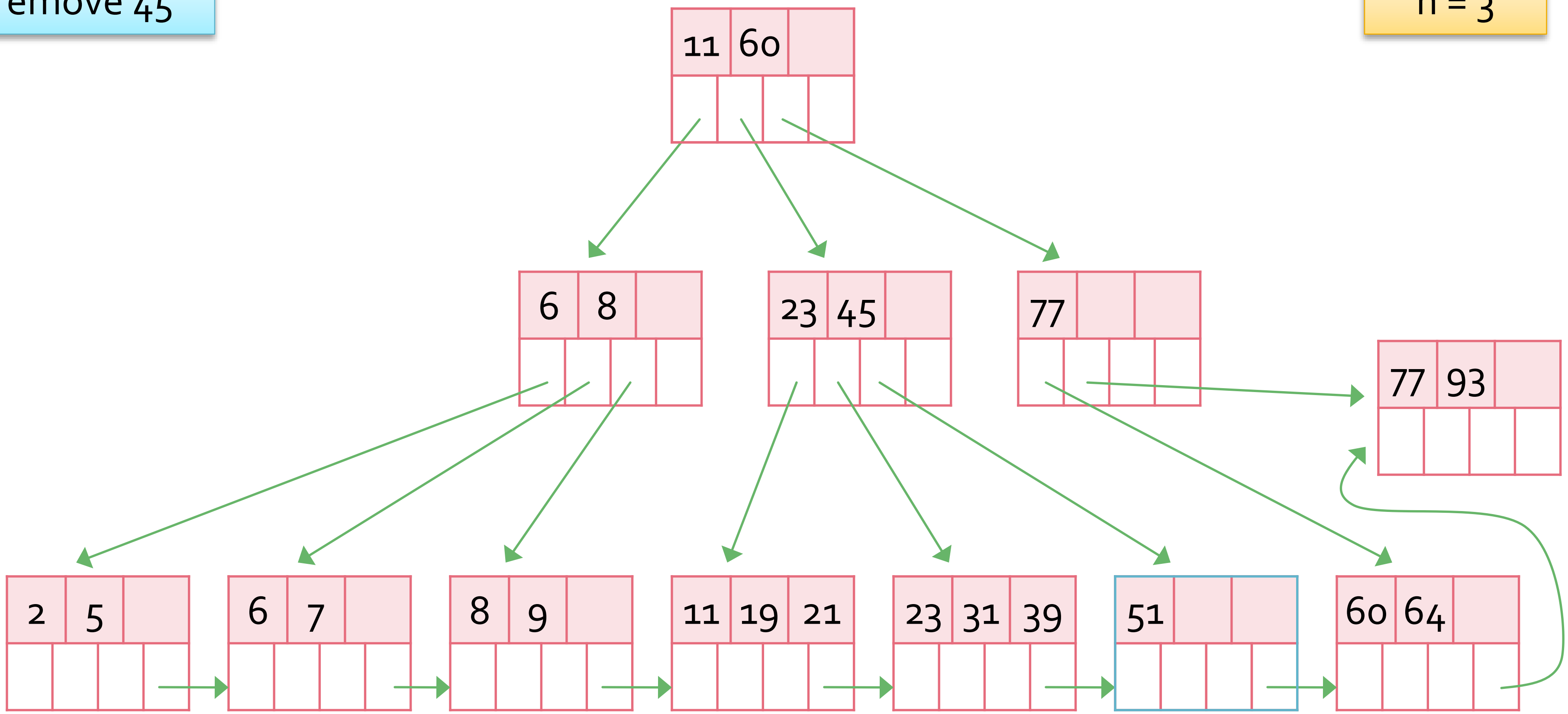


use search to find the value

B+ Tree Removal Example 2

remove 45

n = 3

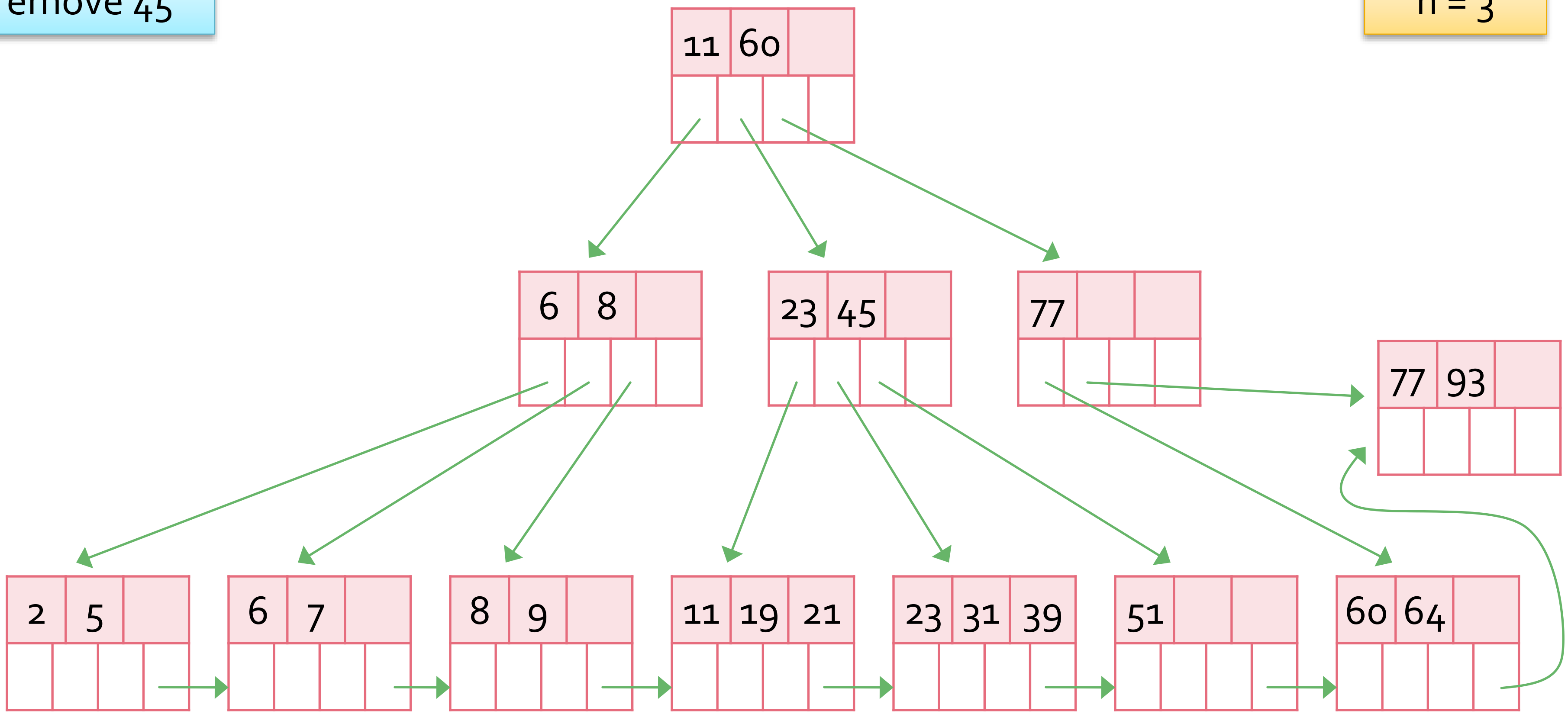


the node is less than half full: $\lfloor (n+1)/2 \rfloor$ pointers to records

B+ Tree Removal Example 2

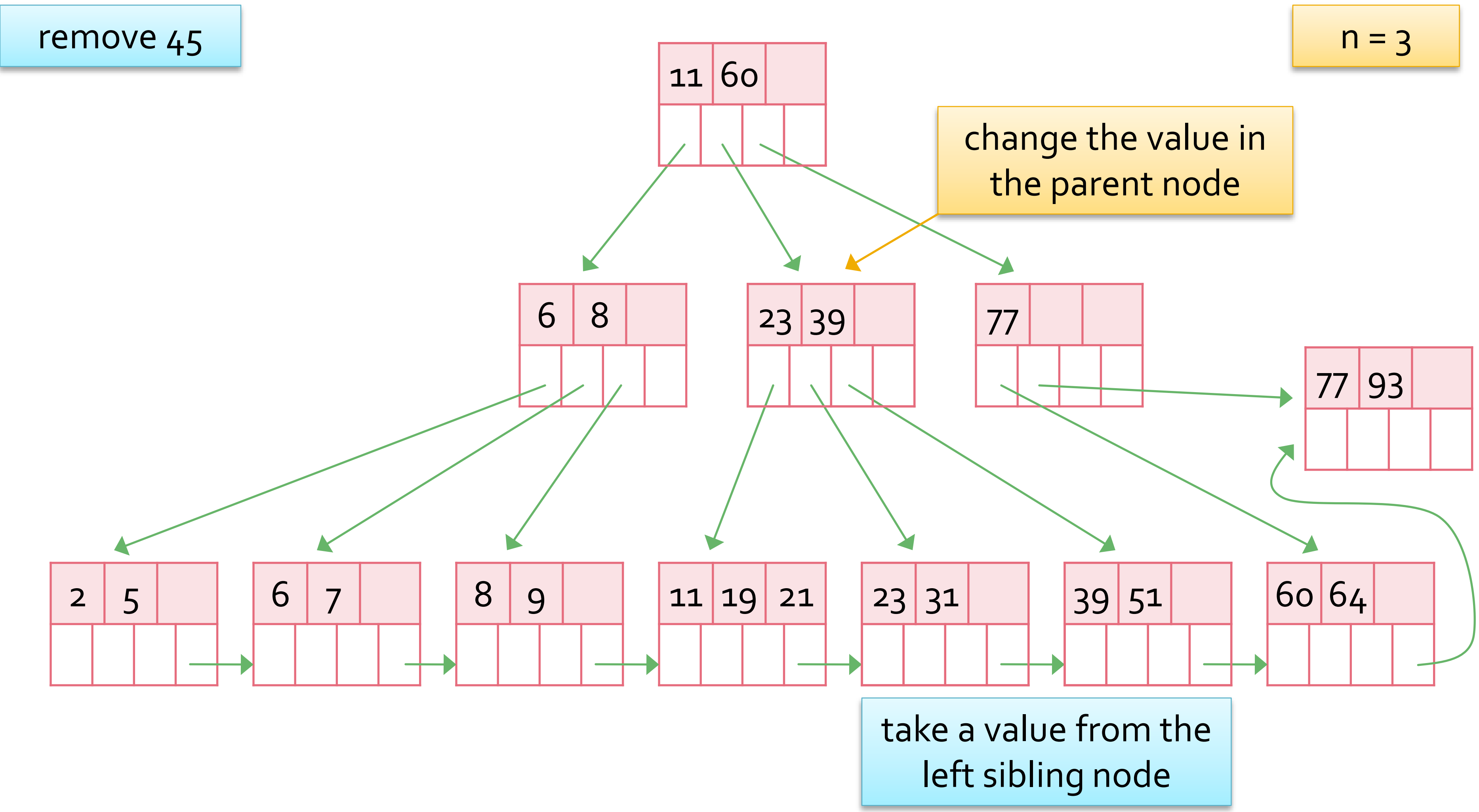
remove 45

n = 3



take a value from the left sibling node

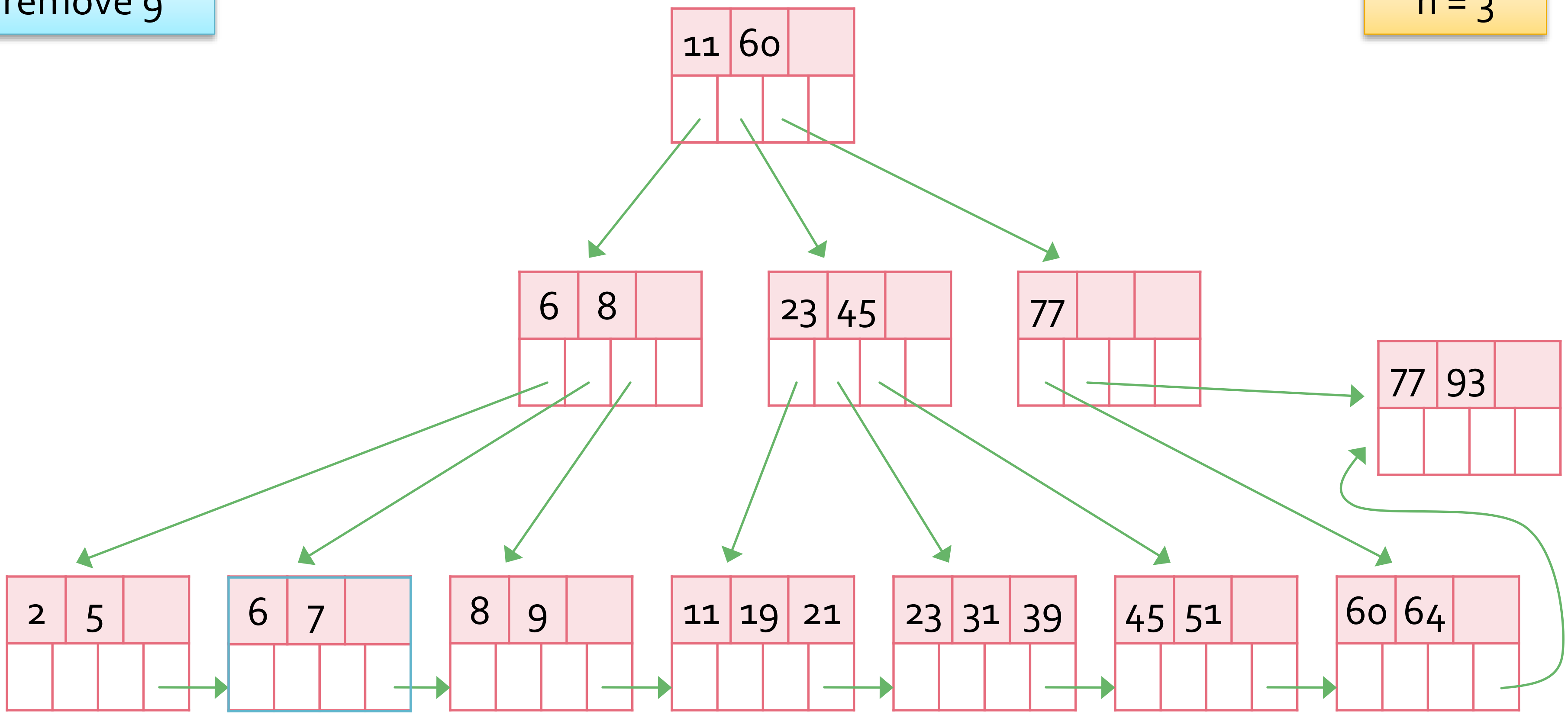
B+ Tree Removal Example 2



B+ Tree Removal Example 3

remove 9

n = 3

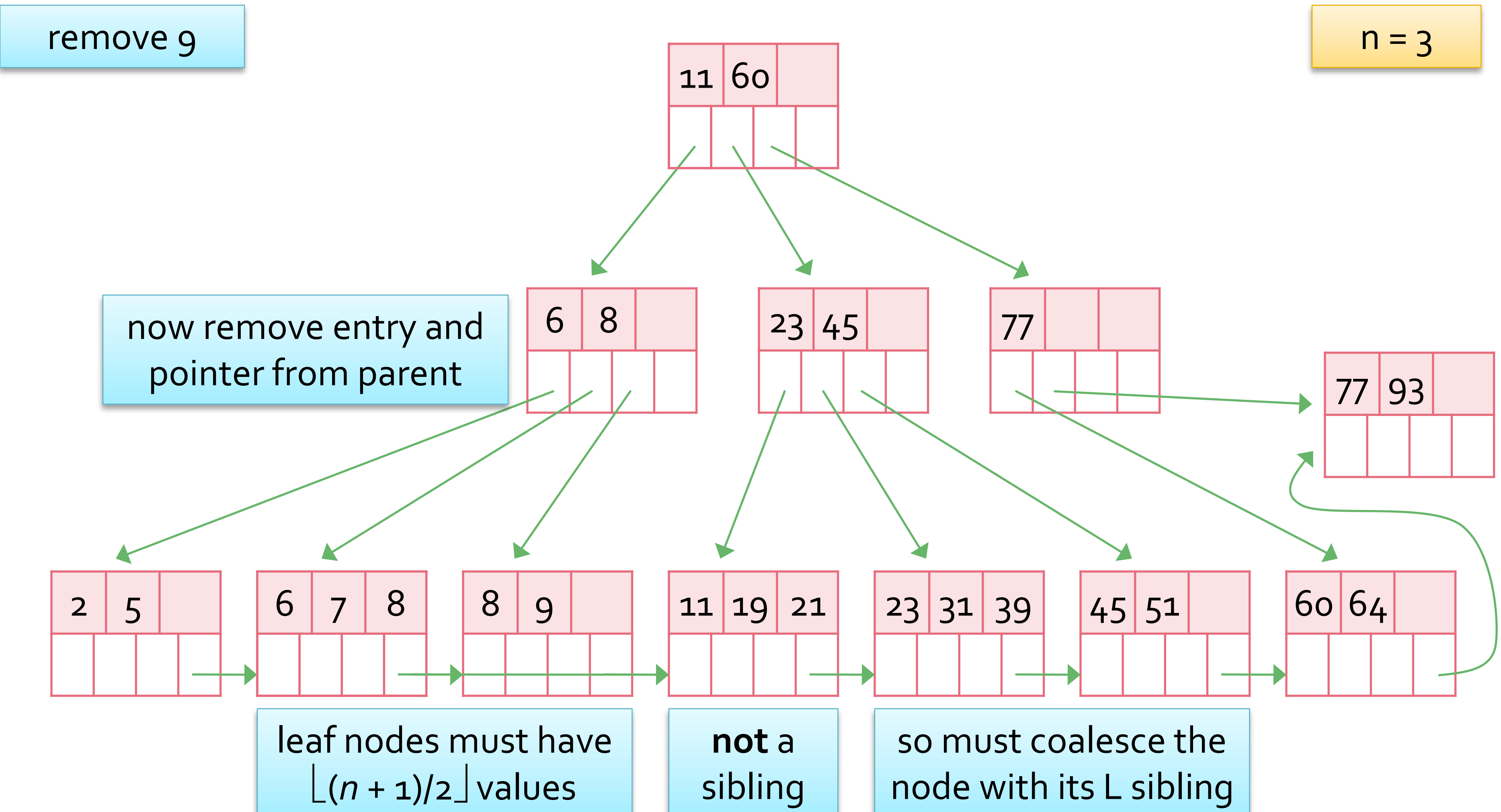


leaf nodes must have $\lfloor (n+1)/2 \rfloor$ values

not a sibling

so must coalesce the node with its L sibling

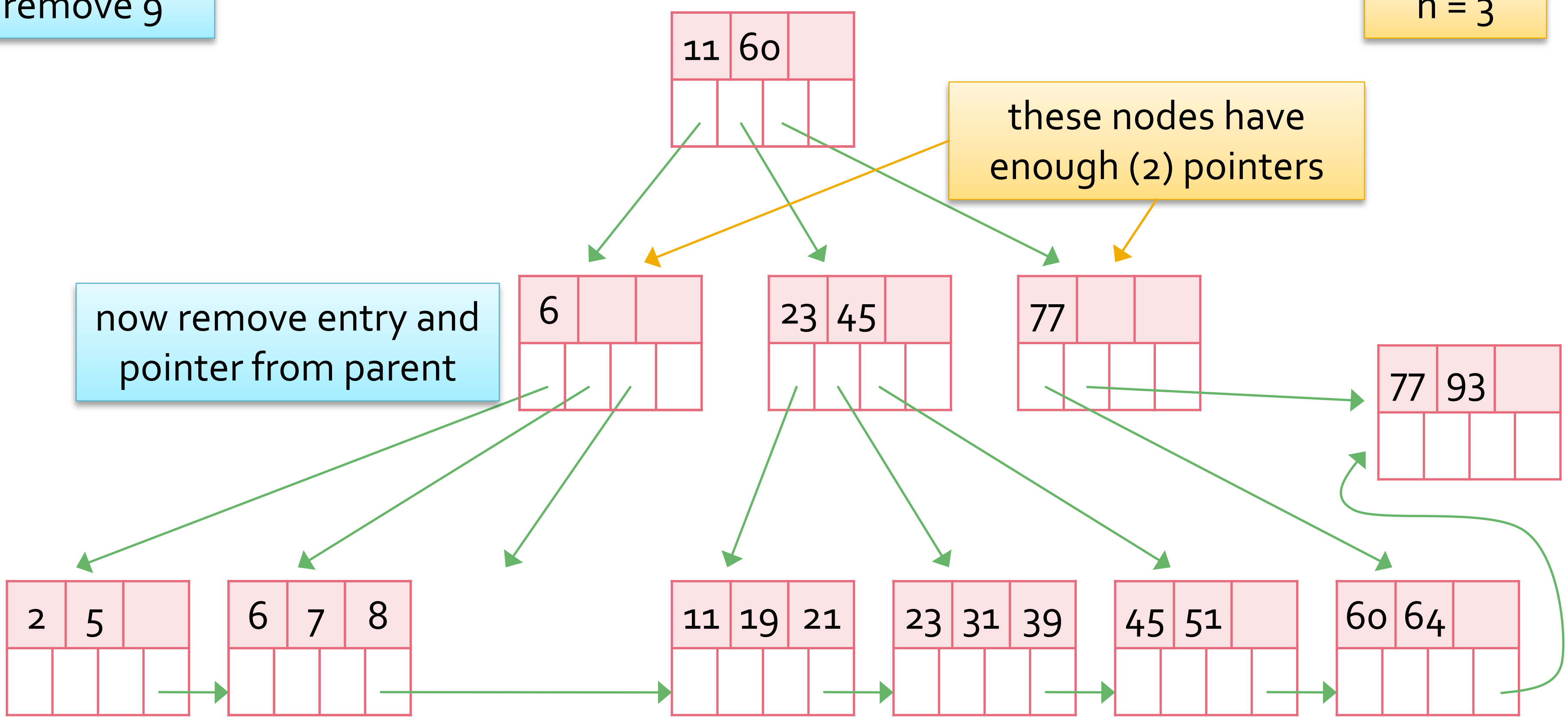
B+ Tree Removal Example 3



B+ Tree Removal Example 3

remove 9

n = 3

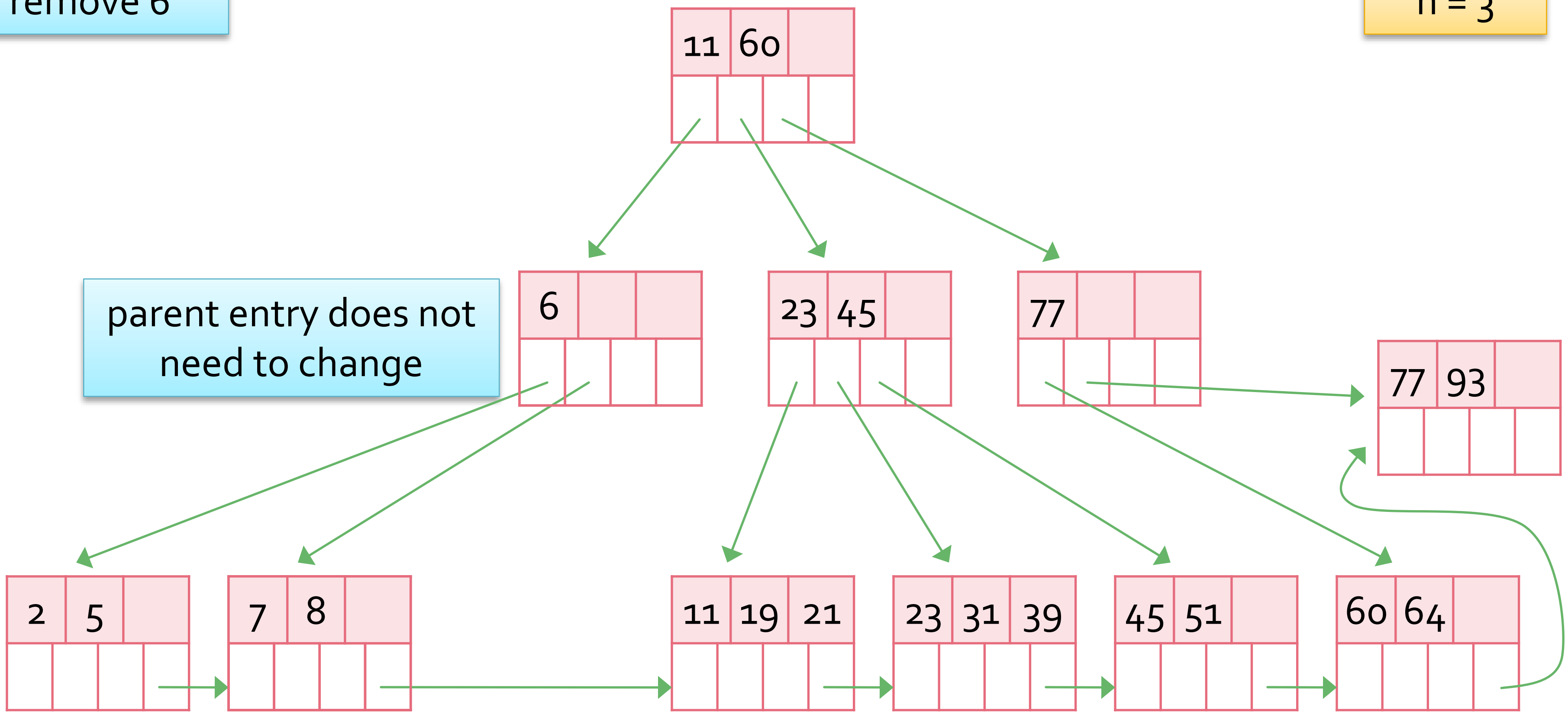


B+ Tree Removal Example 3

remove 6

n = 3

parent entry does not need to change

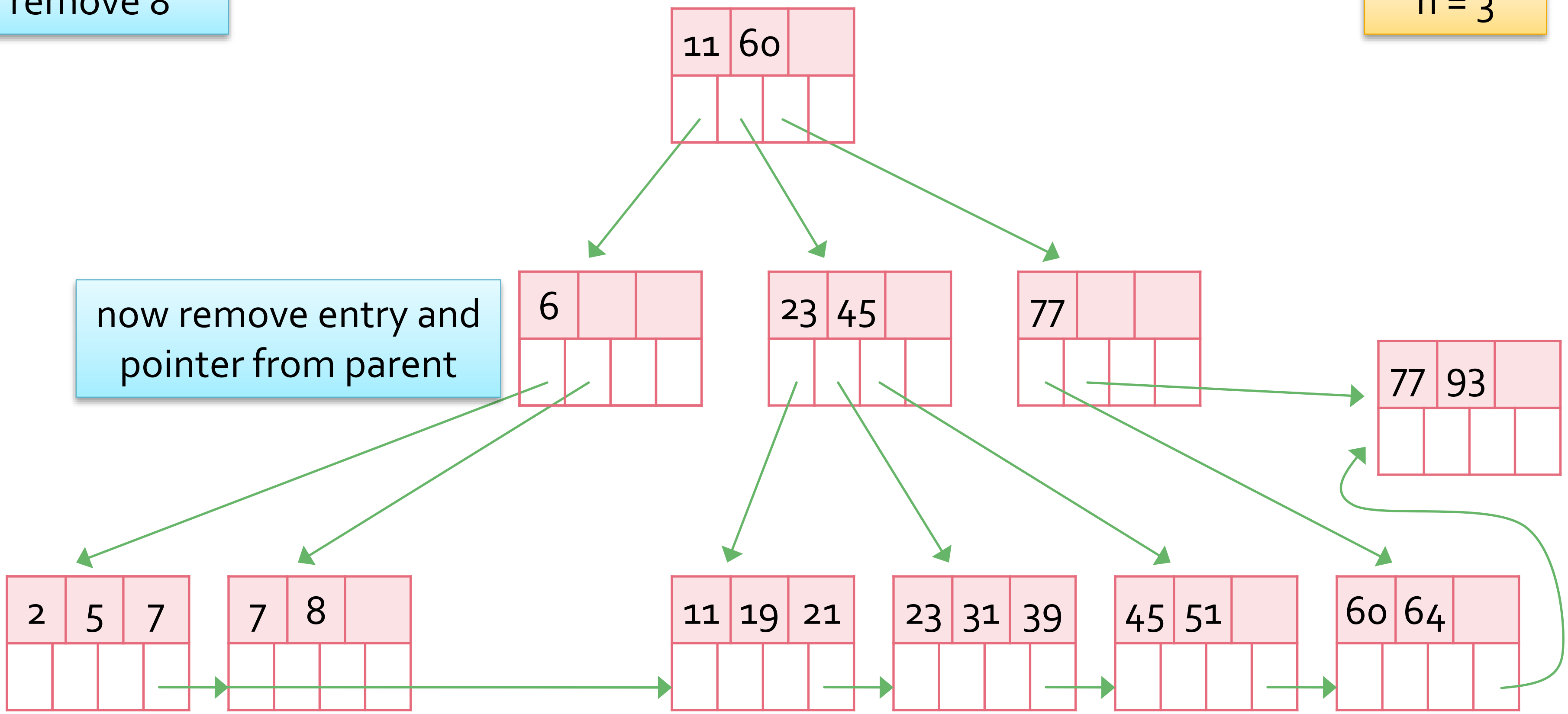


B+ Tree Removal Example 3

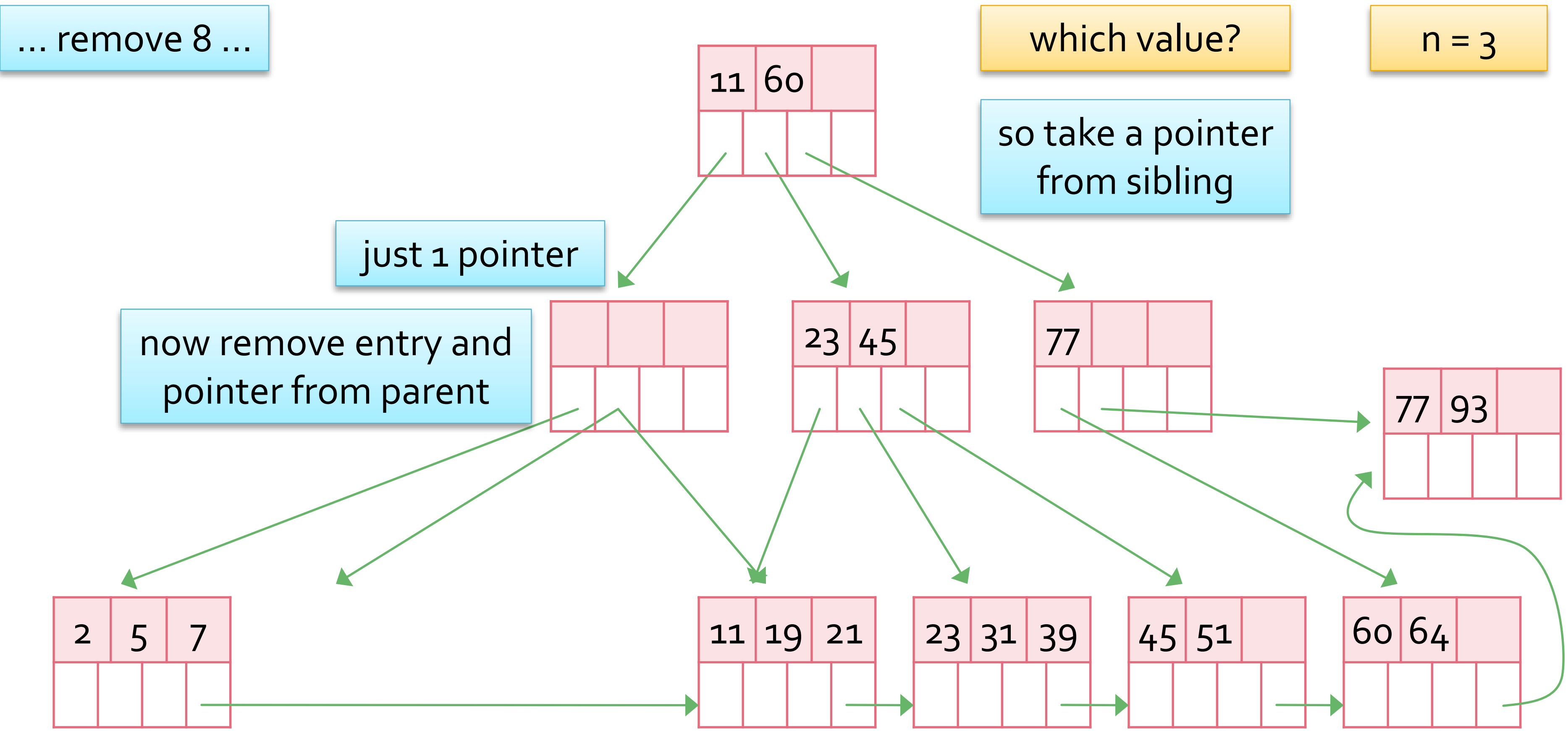
remove 8

n = 3

now remove entry and pointer from parent



B+ Tree Removal Example 3

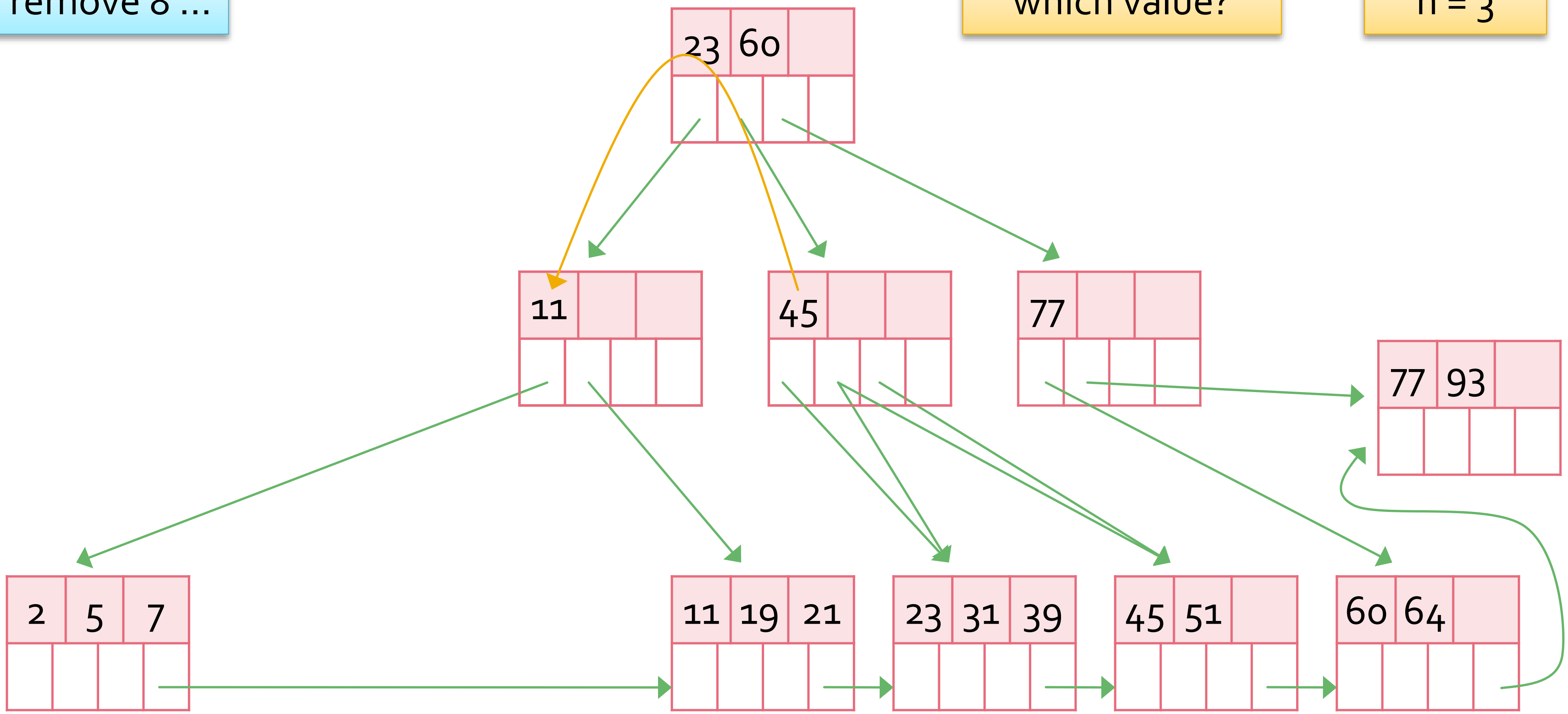


B+ Tree Removal Example 3

... remove 8 ...

which value?

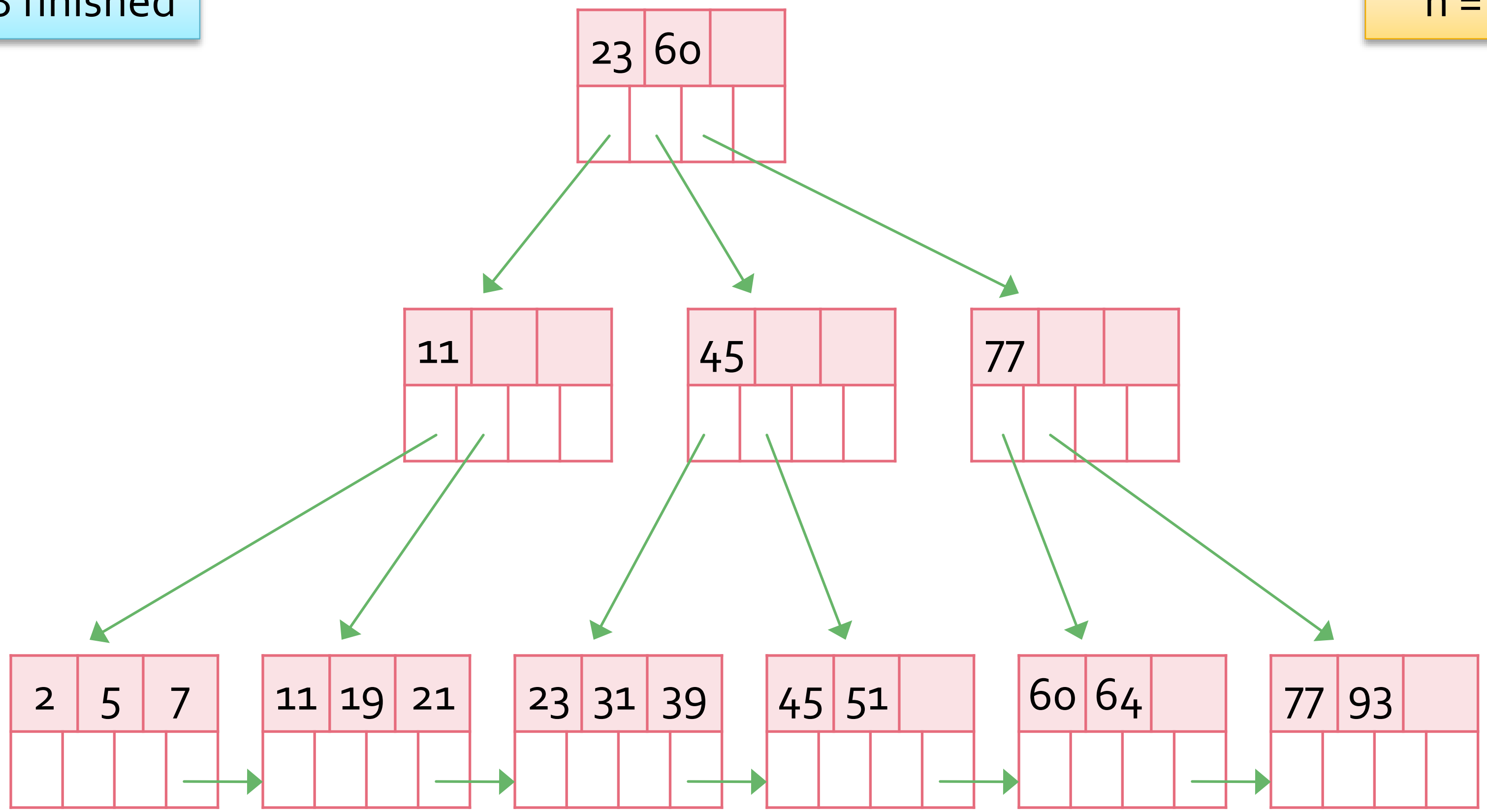
n = 3



B+ Tree Removal Example

… remove 8 finished

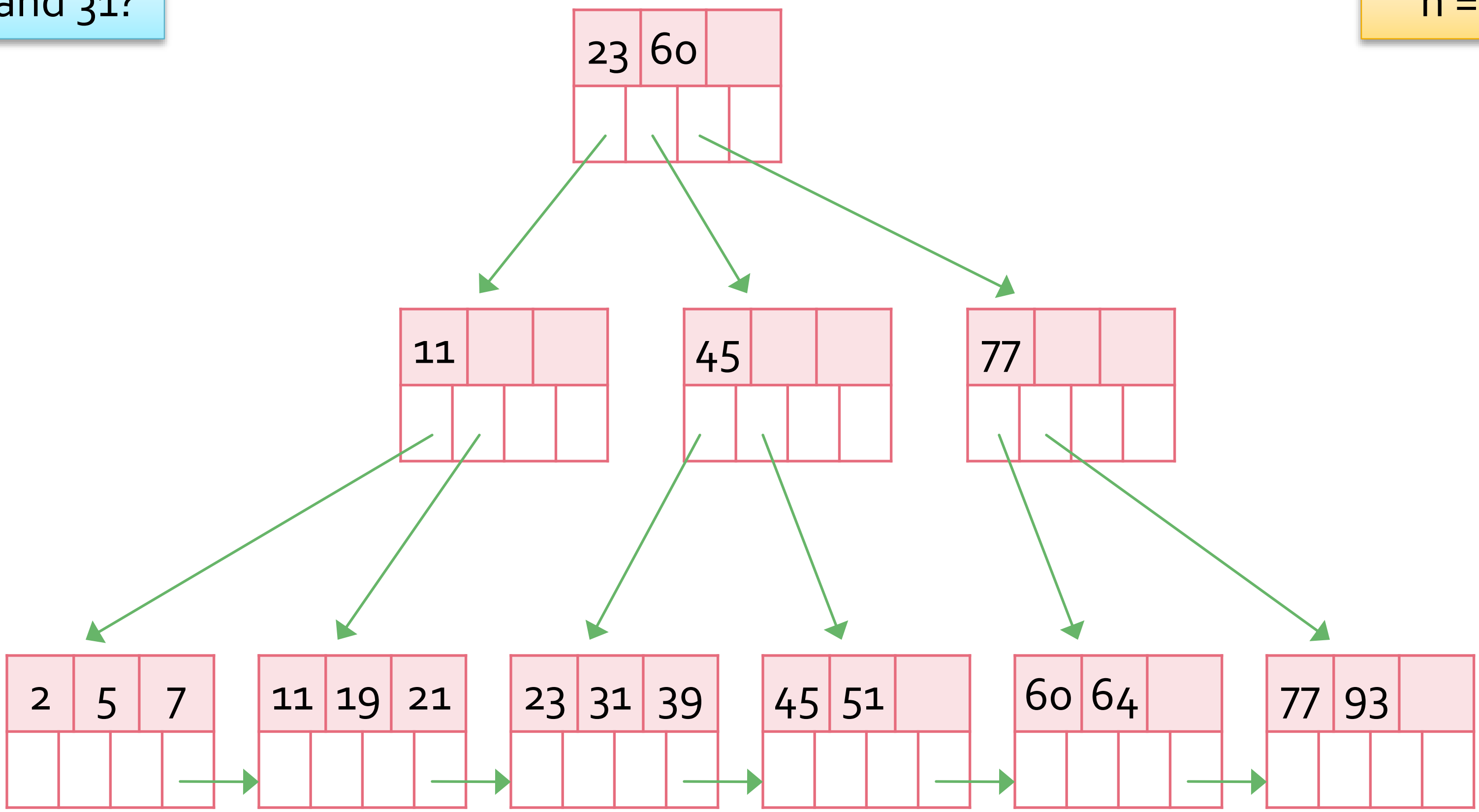
n = 3



B+ Tree Removal Example

remove 23 and 31?

n = 3



B+ Tree Efficiency

- Splitting and merging of index blocks is rare
  - Typically the value of *n* will be greater than 3!
  - Most splits or merges are limited to two leaves and one parent
- The number of cache misses is based on the tree height
  - If *capacity* = 16, a B+ tree would contain approximately one billion values
    - Assuming nodes are half full
  - The number of cache misses is equal to
    - $\log_{capacity/2}$(number of value in tree)