Space-Efficient B Trees via Load-Balancing

Tomohiro I
Department of Artificial Intelligence
Kyushu Institute of Technology
Japan

Dominik Köppl
Tokyo Medical and Dental University
Japan

setting

want to store *n* keys, each of *k* bits
in a data structure with the operations

- predecessor
- insert
- delete
  (not in this talk: done by symmetry)

related work

operation time	space in bits	author(s)	year
O(lg n)	$2nk + 2n \lg\lg n + o(n)$	Prezza	‘17
O(lg n)	$nk + O(nk / \lg^{0.5} n)$	González, Navarro	‘09
O(lg n / lg lg n)	$nk + O(nk / \lg^{0.5} n)$	He, Munro	‘10
O(lg n) ※	$nk + O(\lg n)$	Franceschini, Grossi	‘06
O(lg n)	$nk + O(nk / \lg n)$	this work	

all in word RAM model

※ assuming *realloc* in O(1) time

goal

$nk + O(nk / \lg n)$ bits, $O(\lg n)$ operation time

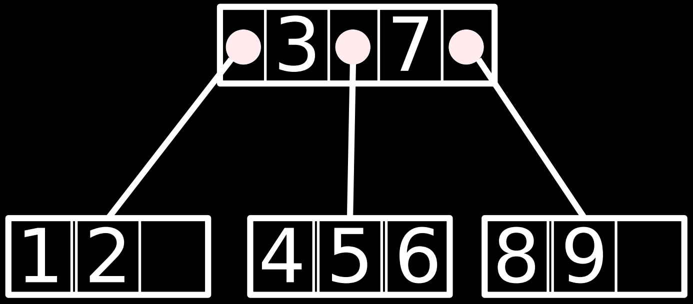
(arXiv: $O(\lg n / \lg\lg n)$ time)

note:
$nk + o(nk)$ considered as *succinct* if keys are incompressible, e.g., keys are pointers: store two keys k_1 and k_2 in the order $k_1^* < k_2^*$

B tree

idea: use B trees!

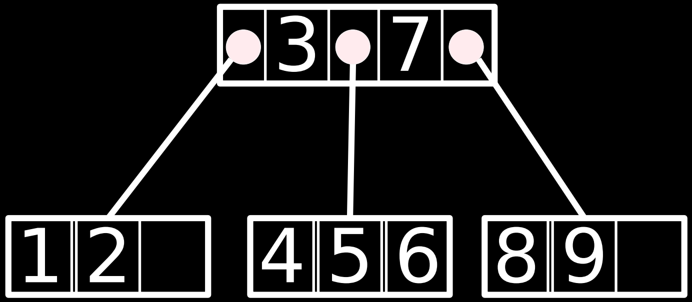
- standard data structure in database systems
- practically more efficient than binary search trees due to data locality



B tree : brief review

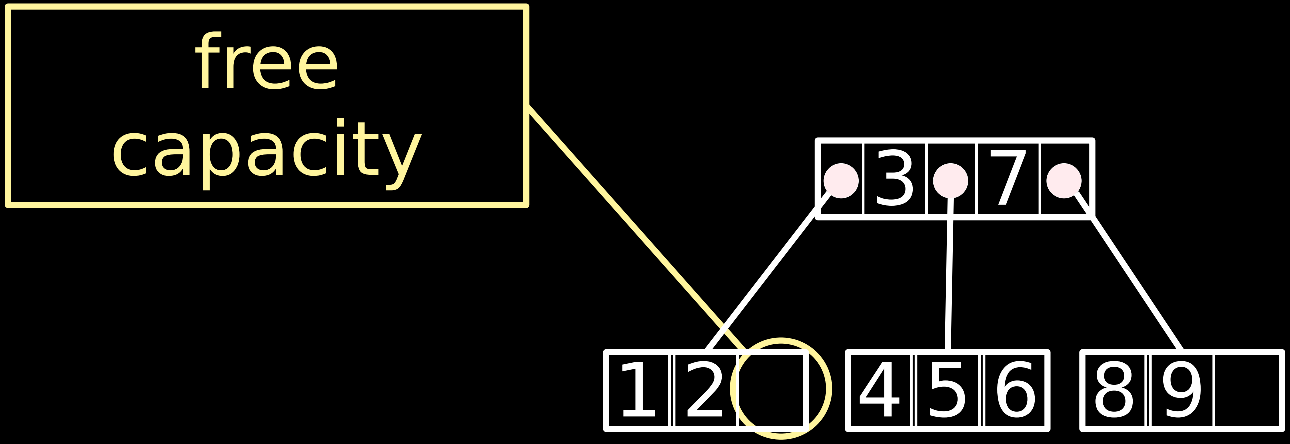
- occupancy = #stored keys
- occupancy of a node is in $[\lceil t/2 \rceil \ldots t]$, where t : constant
- if occupancy violated: split / merge
- tree variations:
  - B tree
  - B+ tree
  - B* tree

combination used in this talk



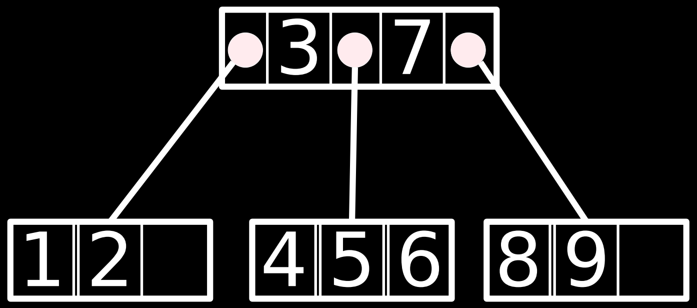
B tree example

- $t = 3$
- each internal node has [2..3] children
- each leaf stores [2..3] keys


free capacity

comparison

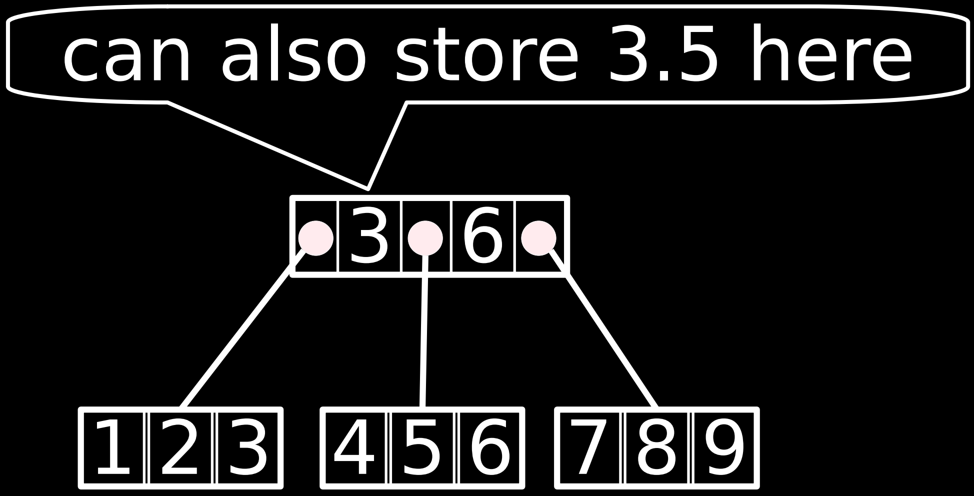
B tree



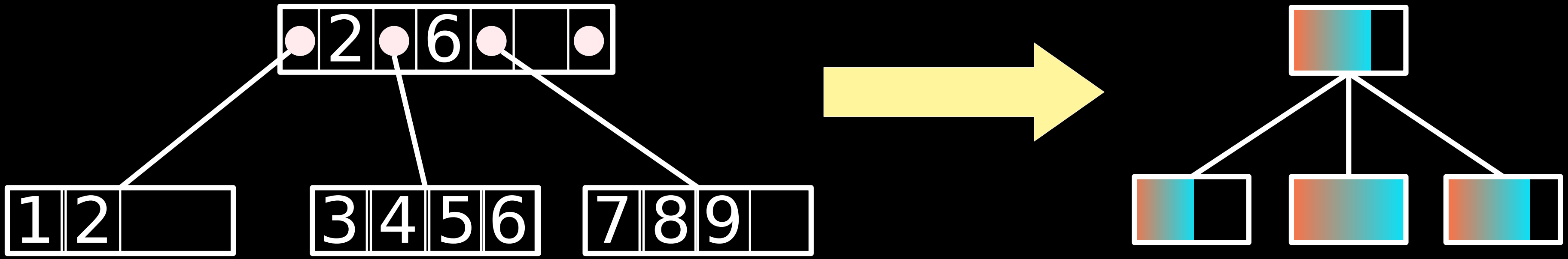
keys are the numbers [0..9]

B+ tree

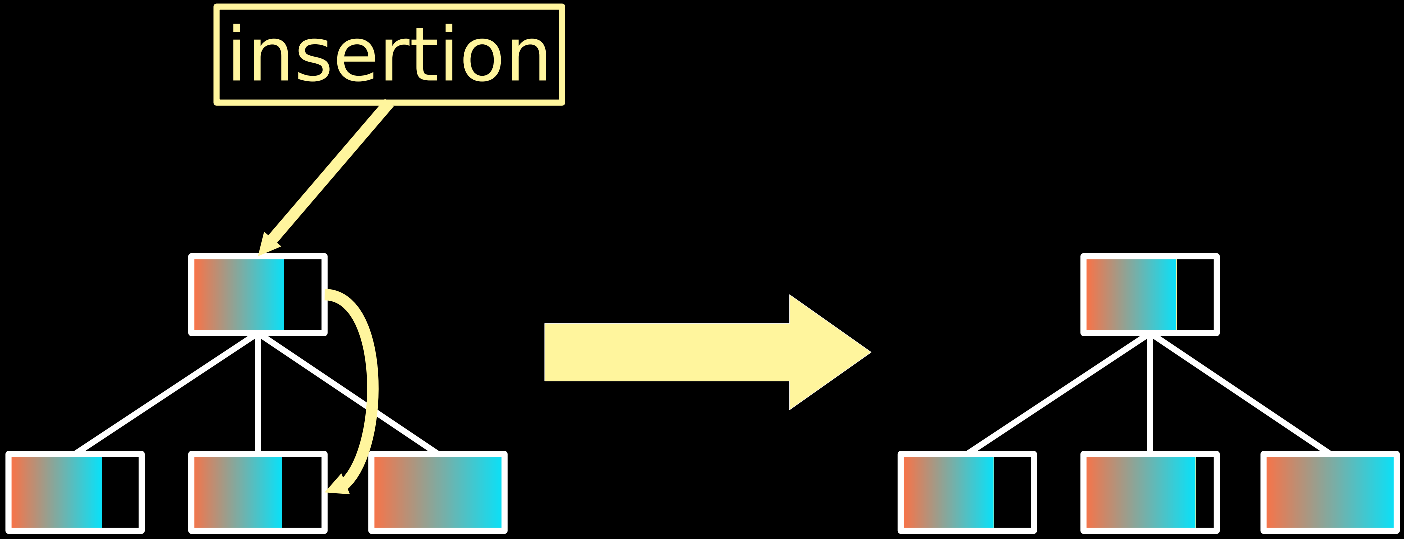
- internal nodes store comparators (not necessarily keys)
- all keys are stored in the leaves


can also store 3.5 here

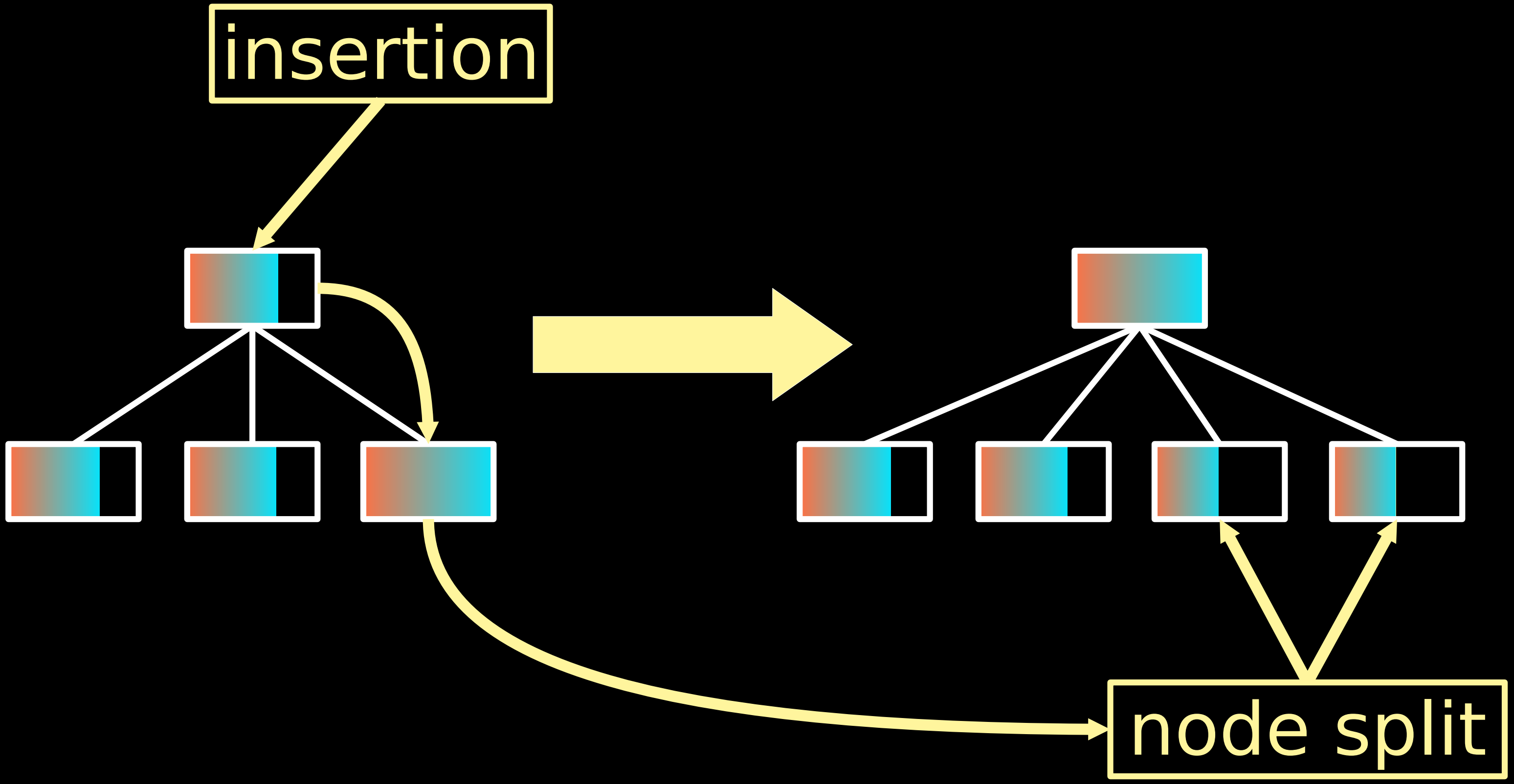
to keep things small:
consider filling instead of actual numbers



insert into non-full leaf



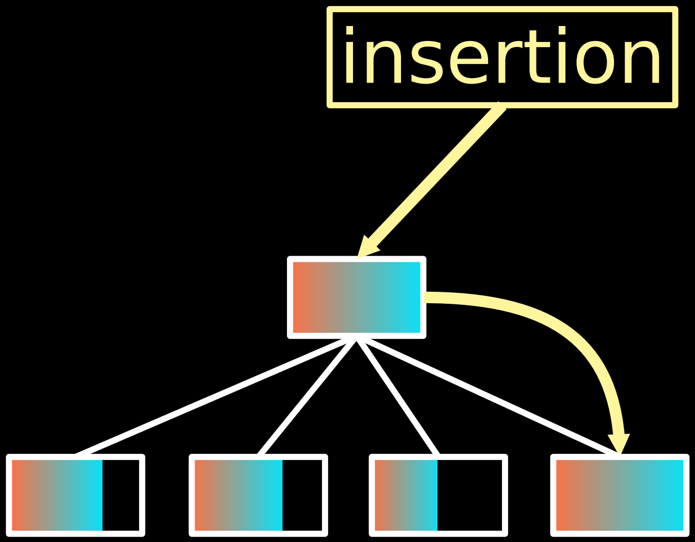
insert into full leaf



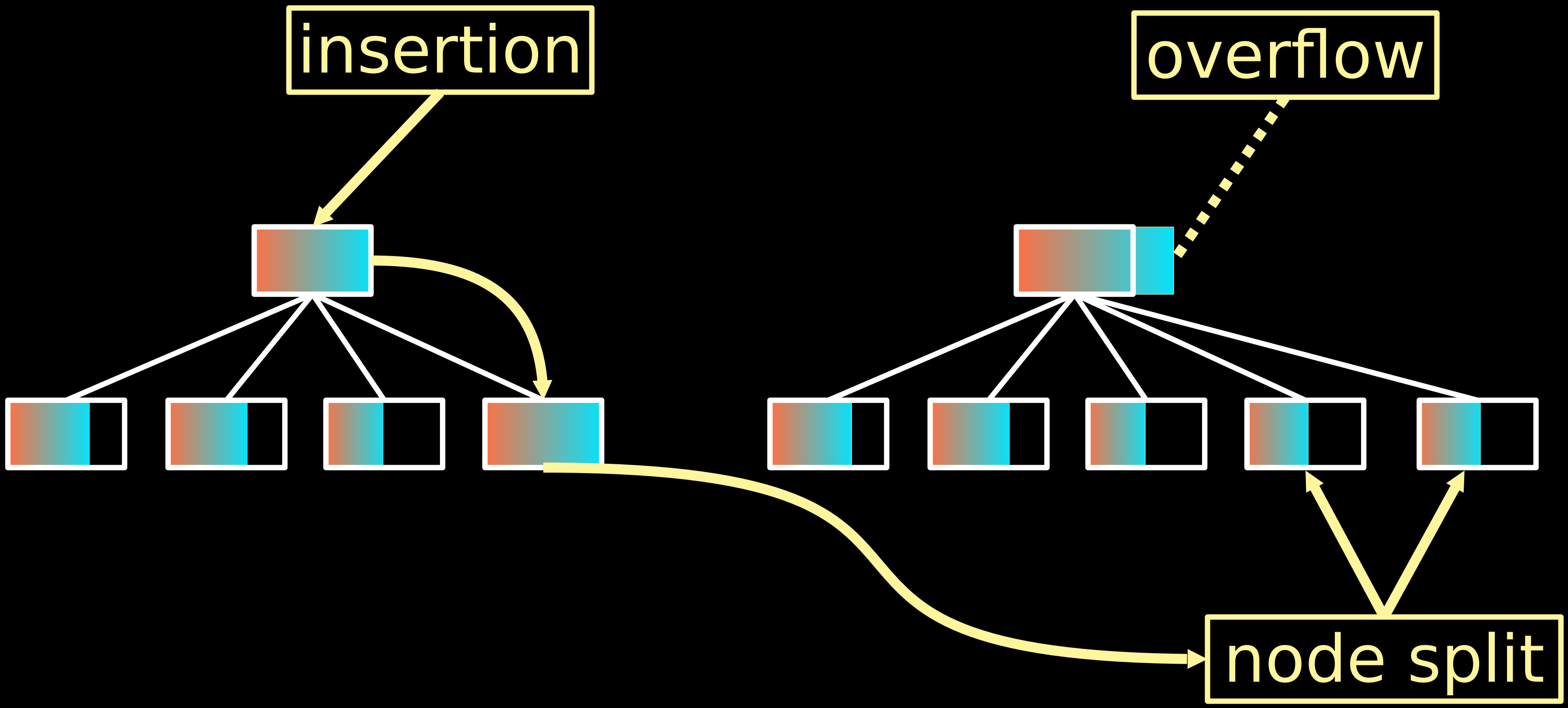
insertion

node split

insert with recursive split



insert with recursive split

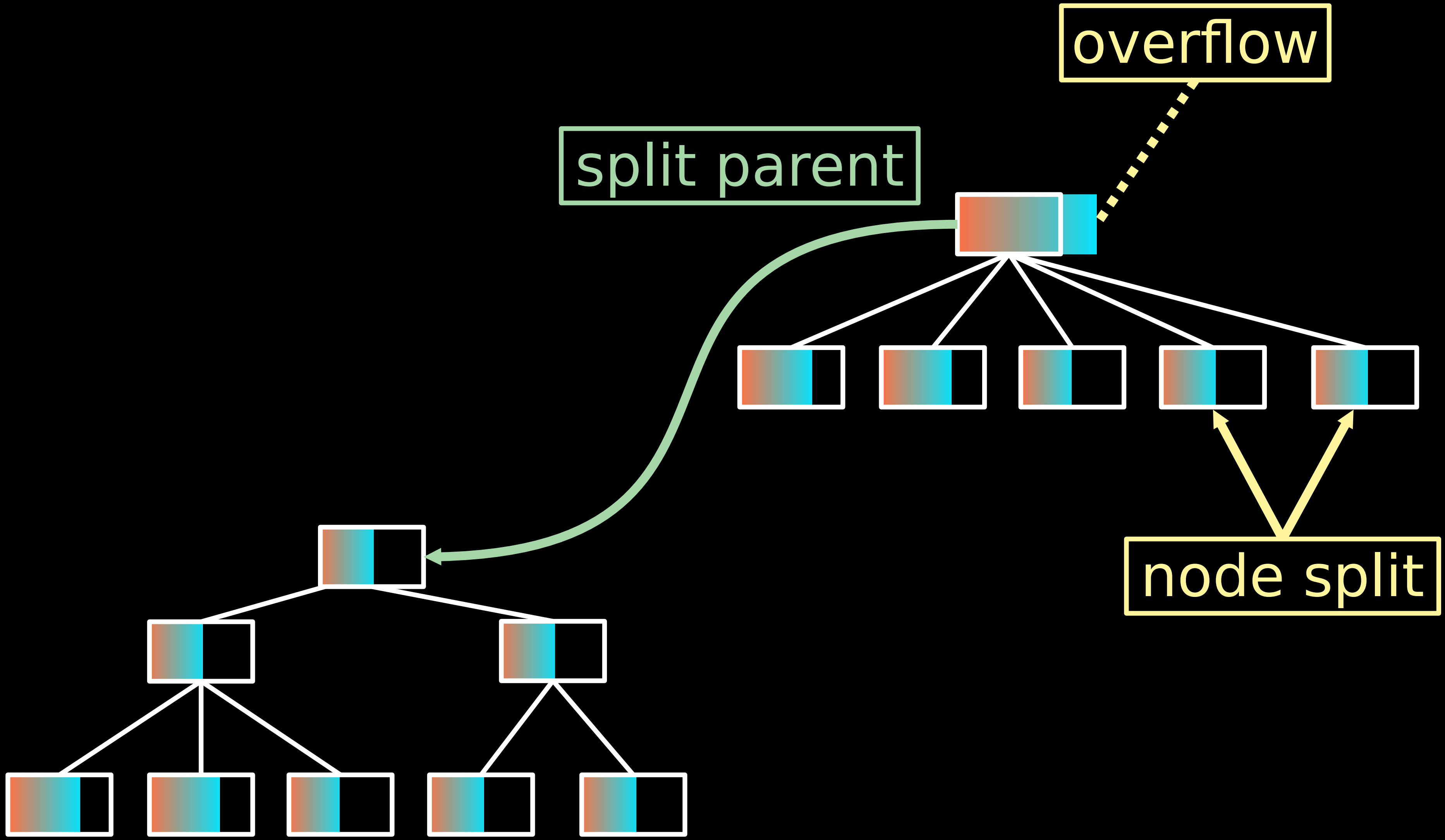


insertion

overflow

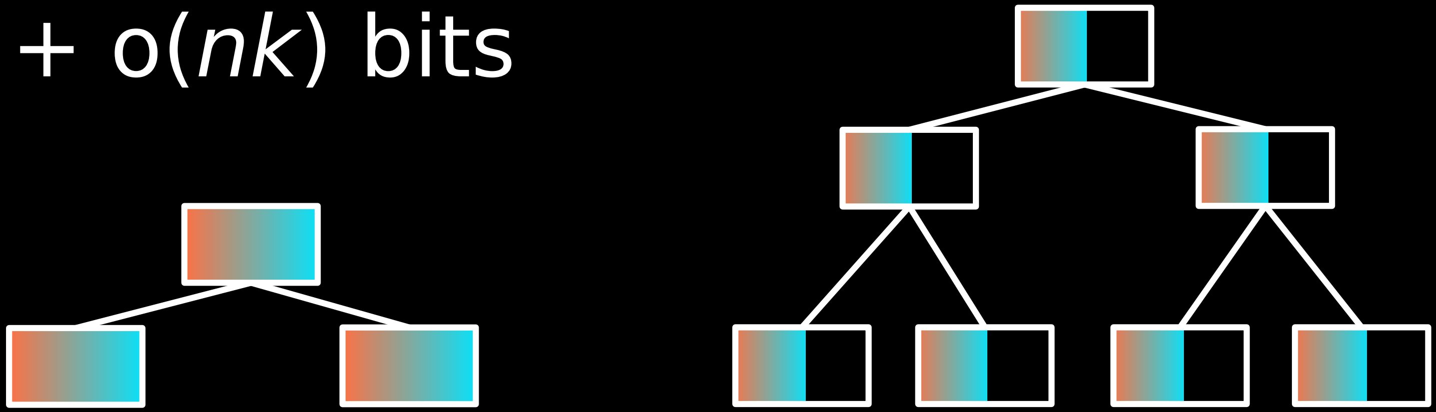
node split

insert with recursive split



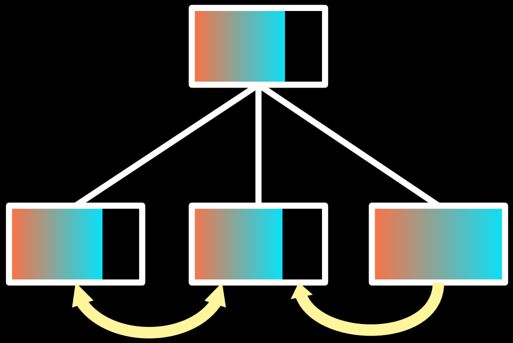
occupancy

- not space efficient: can be just 50% full
- worst case:
  - $2nk$ bits for the leaves
  - #leaves: $n \cdot t/2$ ⇒ #internal nodes: $O(n / t^2)$
- internal nodes: $O(n \lg n / t)$ bits
- aim: $nk + o(nk)$ bits



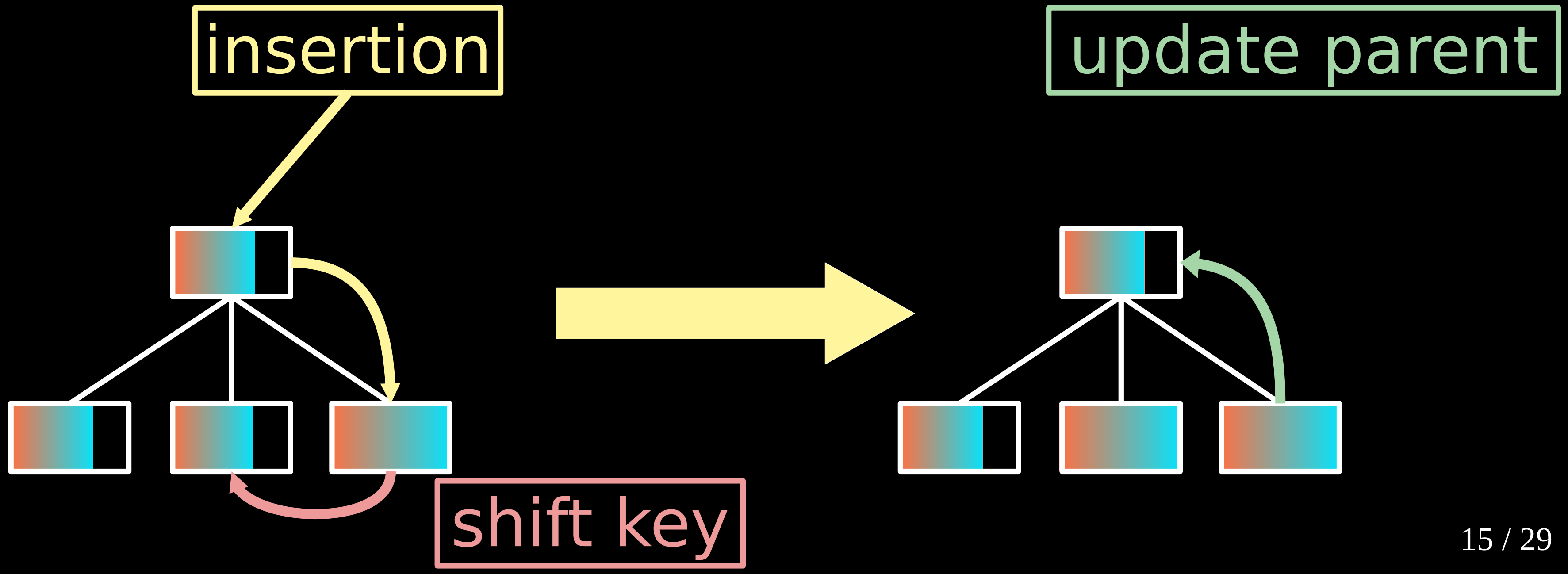
B* tree

- each leaf has a designated sibling called *buddy*



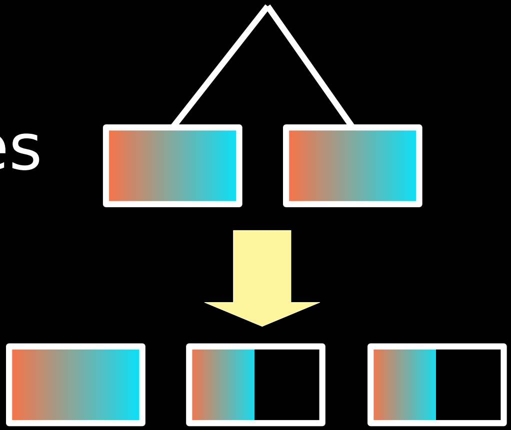
B* tree: insert

- on insert:
  - if leaf is full but buddy not:
    - move key to its buddy
    - update information in ancestor nodes



B* tree

- on insert:
  - if leaf is full but buddy not:
    - move key to its buddy
    - update information in ancestor nodes
  - if both are full ⇒ split
- occupation after split ≥ 2/3
- better than ≥ 1/2 for standard B+ tree
- can idea be generalized?



our tricks

1) generalize B* tree technique from one buddy to $\Theta(\lg n)$ buddies

2) let a leaf store $b := w \lg n / k$ keys

w : machine word size in bits with

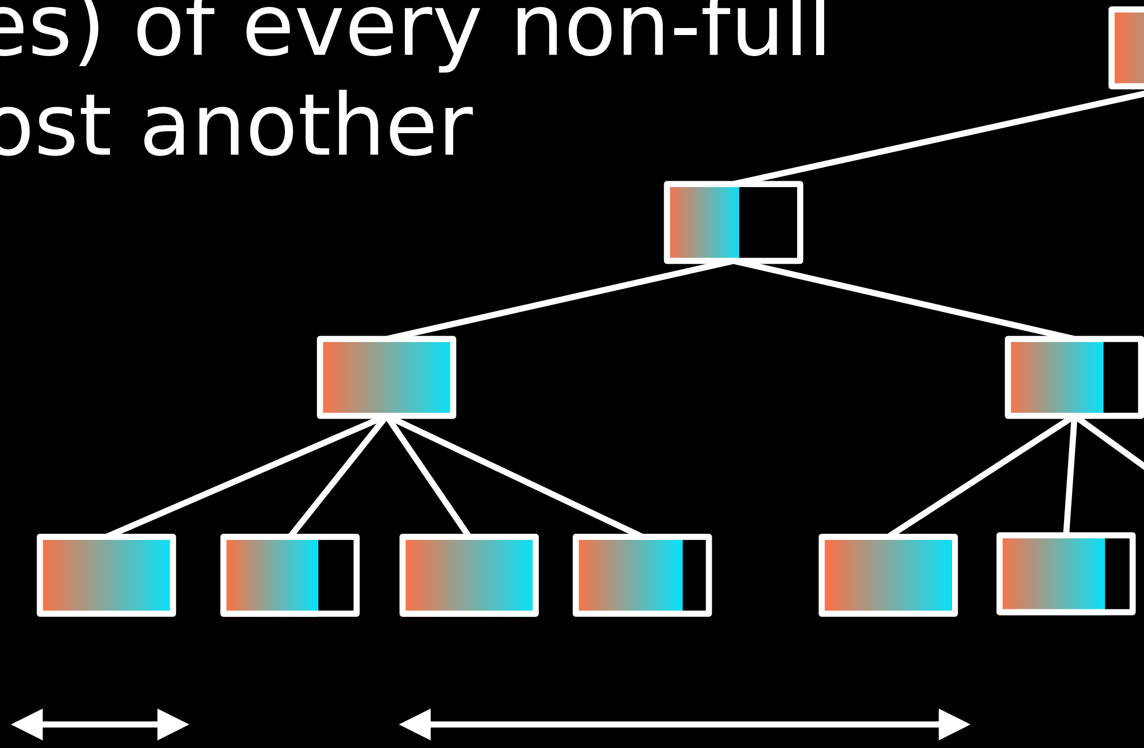
- $k = O(w)$ and
- $n = O(2^w) \cap \Omega(w \lg^2 n / k)$

we need a different data structure for fewer keys

invariant

obey the following invariant for a $c \geq 1$

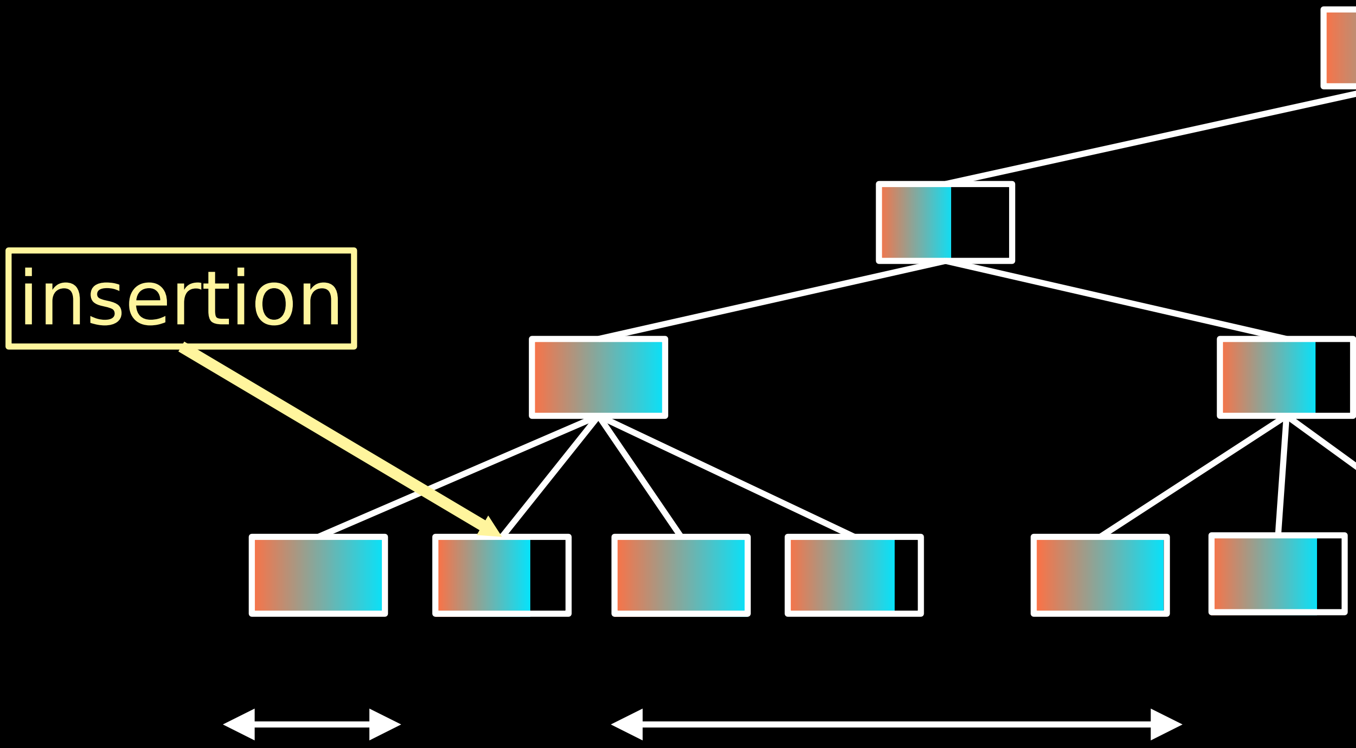
among c lg n assigned buddies (neighboring leaves) of every non-full leaf, there is at most another non-full leaf



c lg n buddies

insert

if leaf is not full: just insert

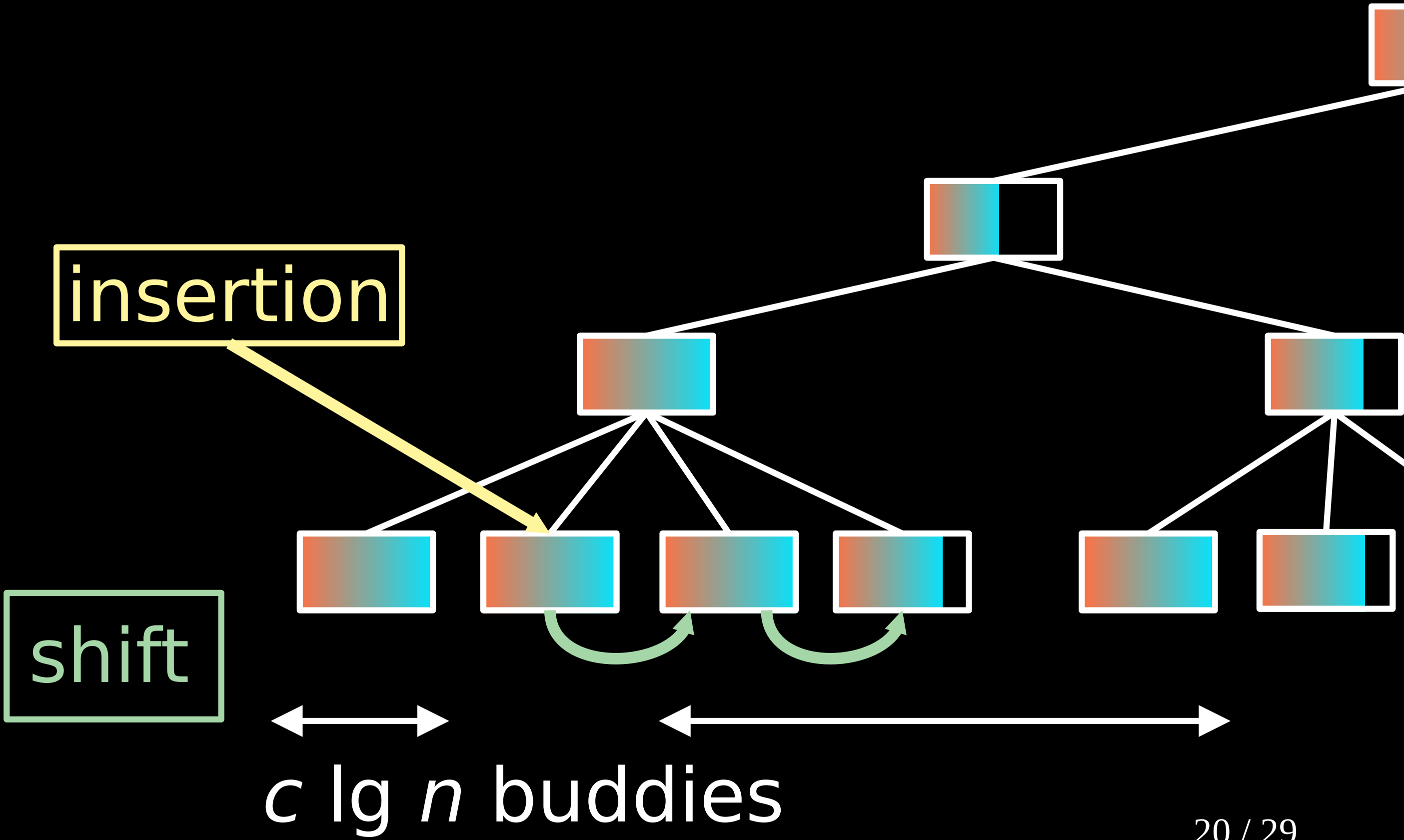


insertion

c lg n buddies

insert

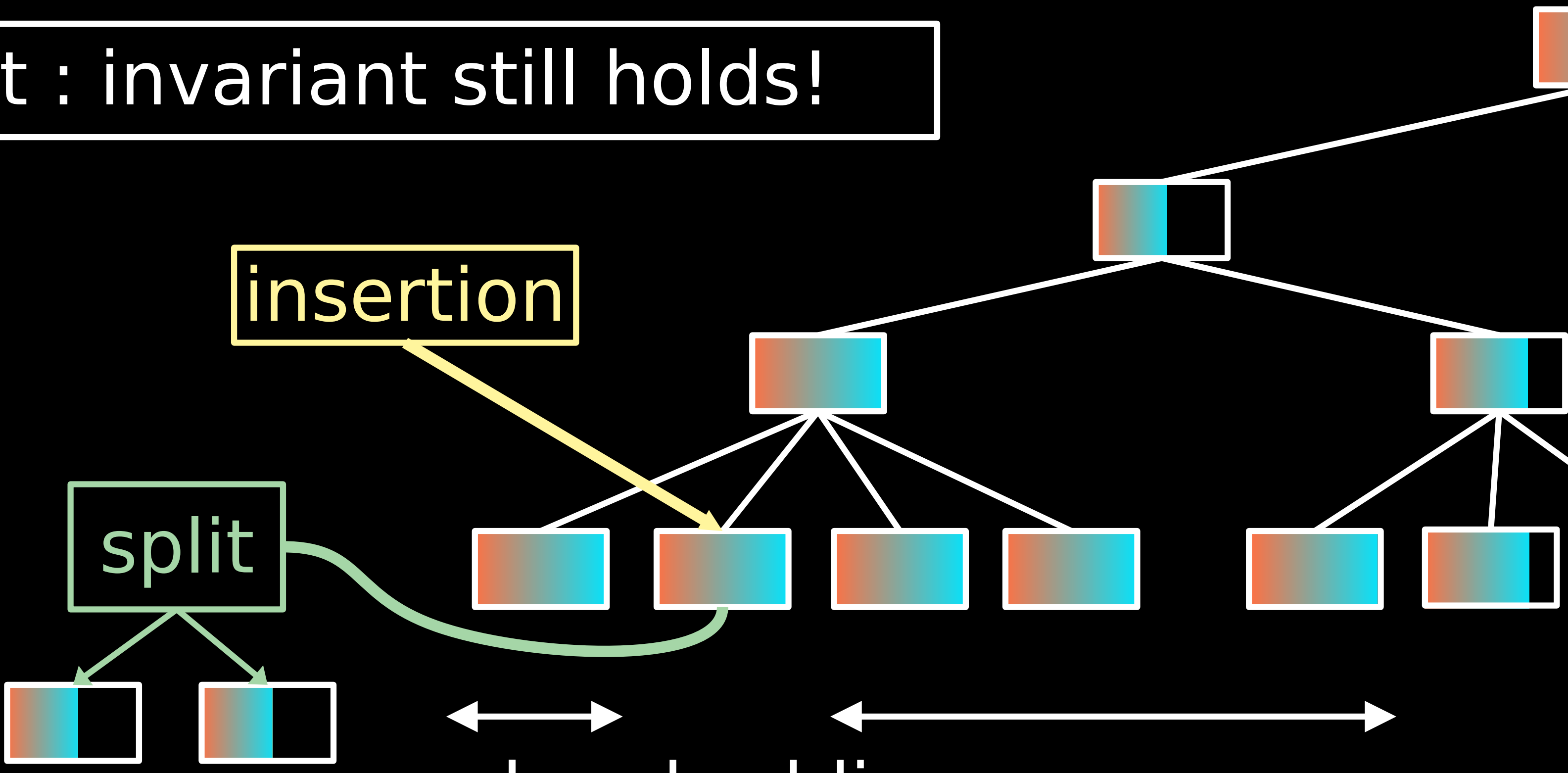
if leaf is full & a buddy is not full:
    shift key to buddy



insert

if leaf is full & all buddies are full:
    split leaf

after split : invariant still holds!



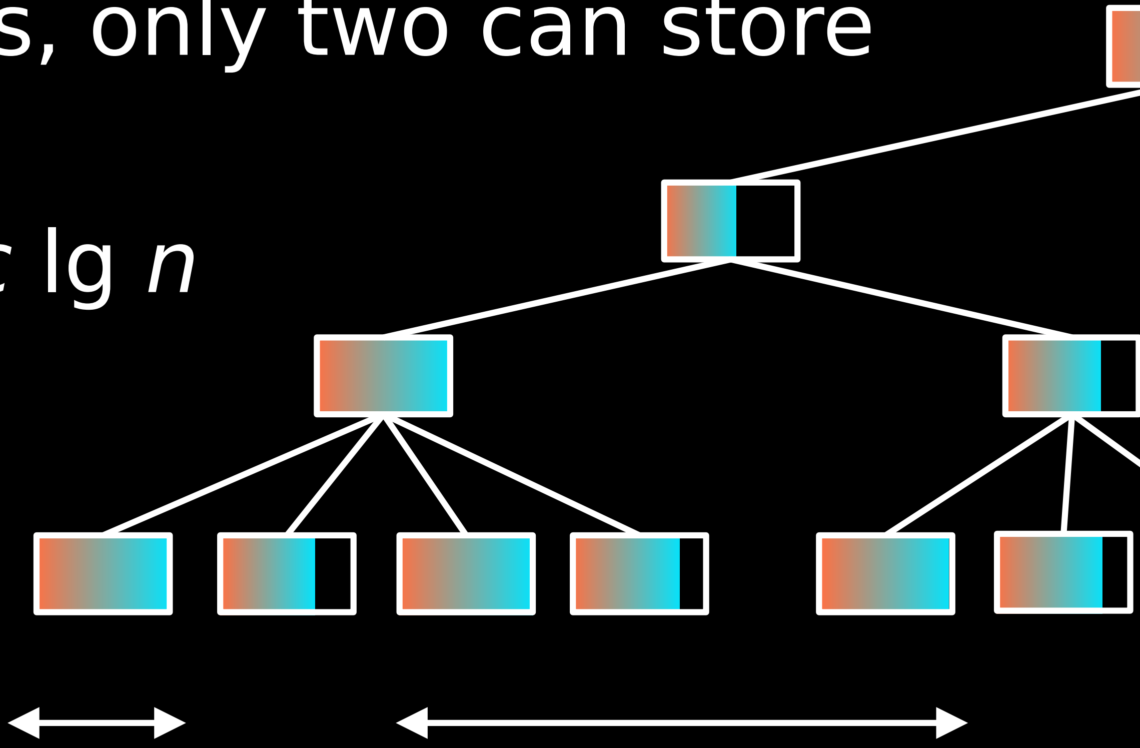
insertion

split

c lg n buddies

leaves : space

- occupation rate is $\geq (c \lg n - 1) / (c \lg n)$
- a leaf stores $[t/2..t]$ keys, but within $c \lg n$ consecutive leaves, only two can store less than t keys
- full occupation: $t\, c \lg n$
- achieved:
  $\geq t\,(c \lg n - 2)$
  $+ 2 \cdot t/2$



$c \lg n$ buddies

leaves : space

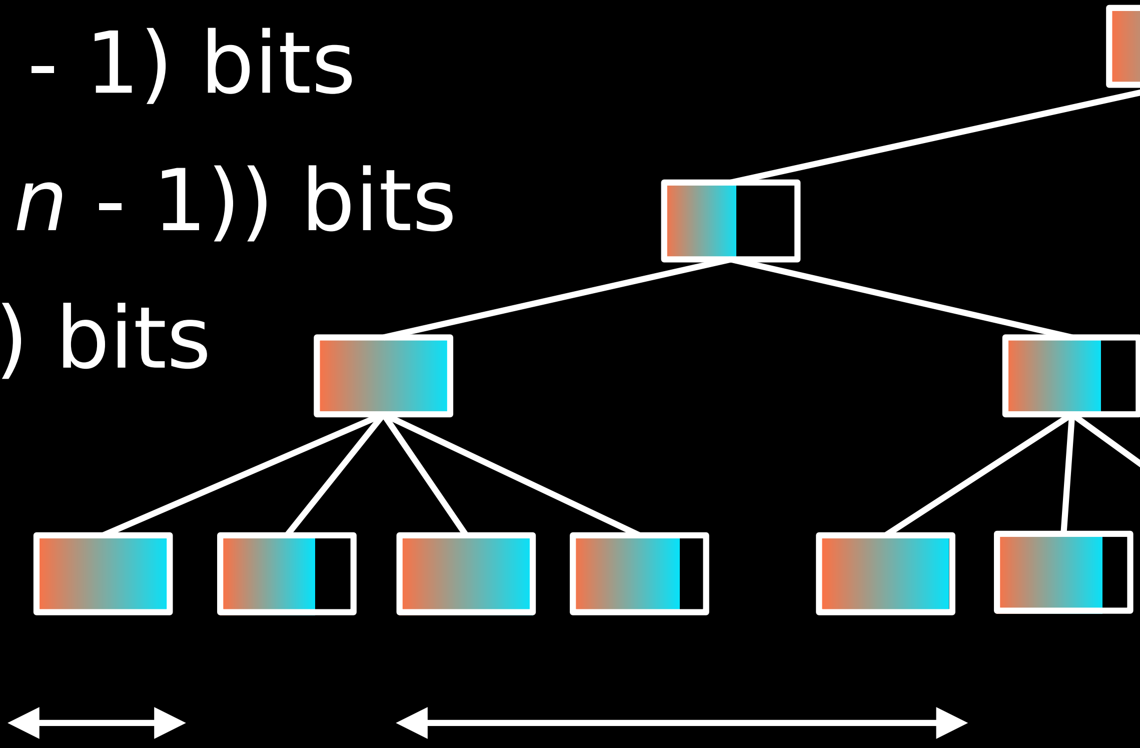
- occupation rate is ≥ (*c* lg *n* - 1) / (*c* lg *n*)
- leaves use at most

  nk (*c* lg *n*) / (*c* lg *n* - 1) bits

  = *nk* (1 + 1 / (*c* lg *n* - 1)) bits

  = *nk* + O(*nk* / lg *n*) bits

$$\frac{x}{x-1} = 1 + \frac{1}{x-1}$$



c lg *n* buddies

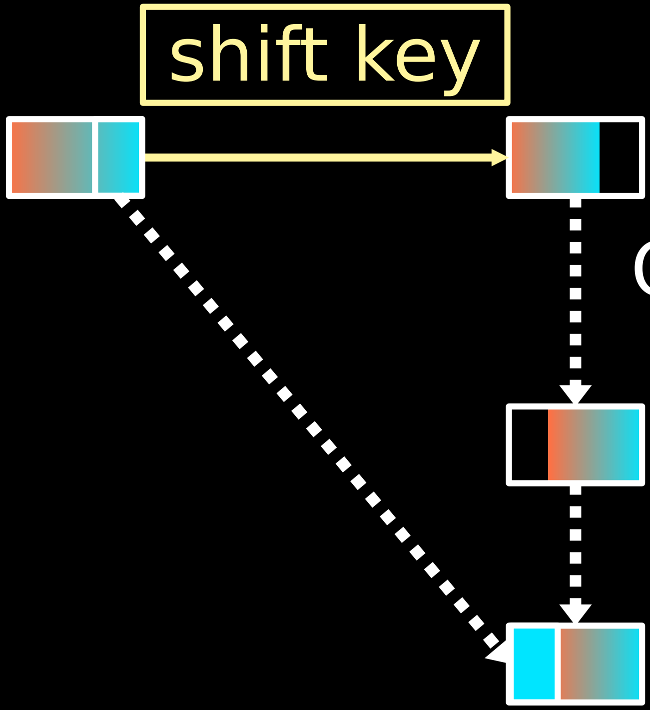
total space

- let a leaf store $[b/2..b]$ keys with $b := w \lg n / k$
- then there are at most $O(n / (t\, b)) = O(nk / (t\, w \lg n))$ internal nodes
- each internal node stores $O(t \lg n)$ bits

  $\Rightarrow O(nk / w)$ bits total for internal nodes
- leaves: $nk + O(nk / \lg n)$ bits
- total: $nk + O(nk / \lg n)$ bits

shifting in large leaves

shifting a key can take O(b) time from one leaf to another ⇒ O($b \lg n$) time for insertion!



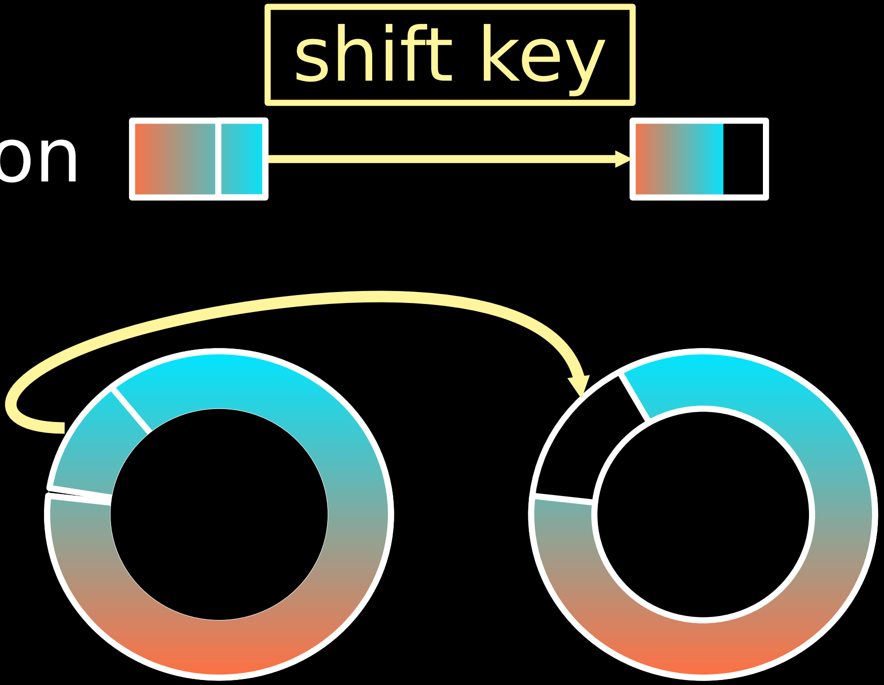
shift key

array representation

O(b) time

ring buffer

- use ring buffer instead of array
  - tail/head insertion/removal in O(1) time



shift key

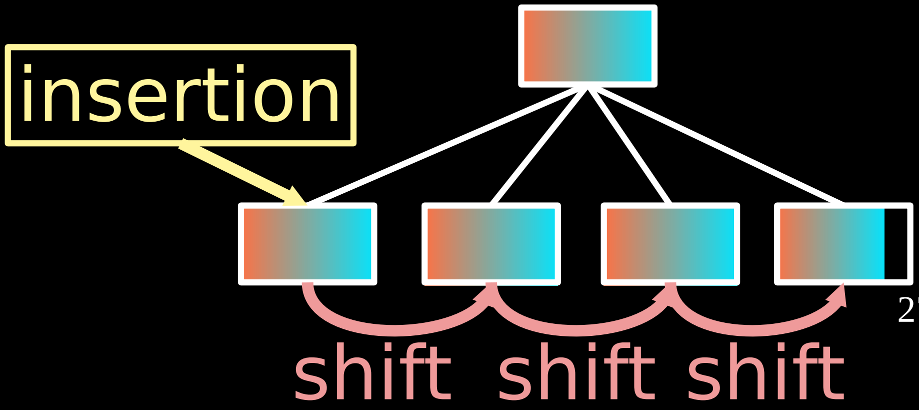
array representation

ring buffer

word packing

- shift: O(1) time
- save time by packing w/k keys into a machine word
- insert: O(lg n) time since ring buffer size is $b = w \lg n / k$ keys
- total: O(lg n) + $c \lg n \cdot$ O(1) time

  = O(lg n) time



full paper on arXiv

- augmentation with aggregated values like prefix-sum / minimum / maximum key in the internal nodes
- $O(\lg n / \lg \lg n)$ operation time by dynamic fusion trees [Patrascu, Thorup '14]
- $n\,O(\lg 2^k / n) + O(n)$ bits by compression [Blandford and Blelloch '04] if $w = \Theta(\lg n)$
- works in external memory with the same $\Theta(\log_B n)$ I/Os as classic B trees

summary

succinct B+ tree variant

- generalize B* tree technique from one buddy to $\Theta(\lg n)$ buddies
- leaves store large number of keys
- shift keys among ring buffers representing leaves

result:

- $nk + O(nk / \lg n)$ bits
- $O(\lg n)$ operation time