B-Tree Variants. Amortized Analysis

B-Tree Variants. Accounting method. Splay Trees

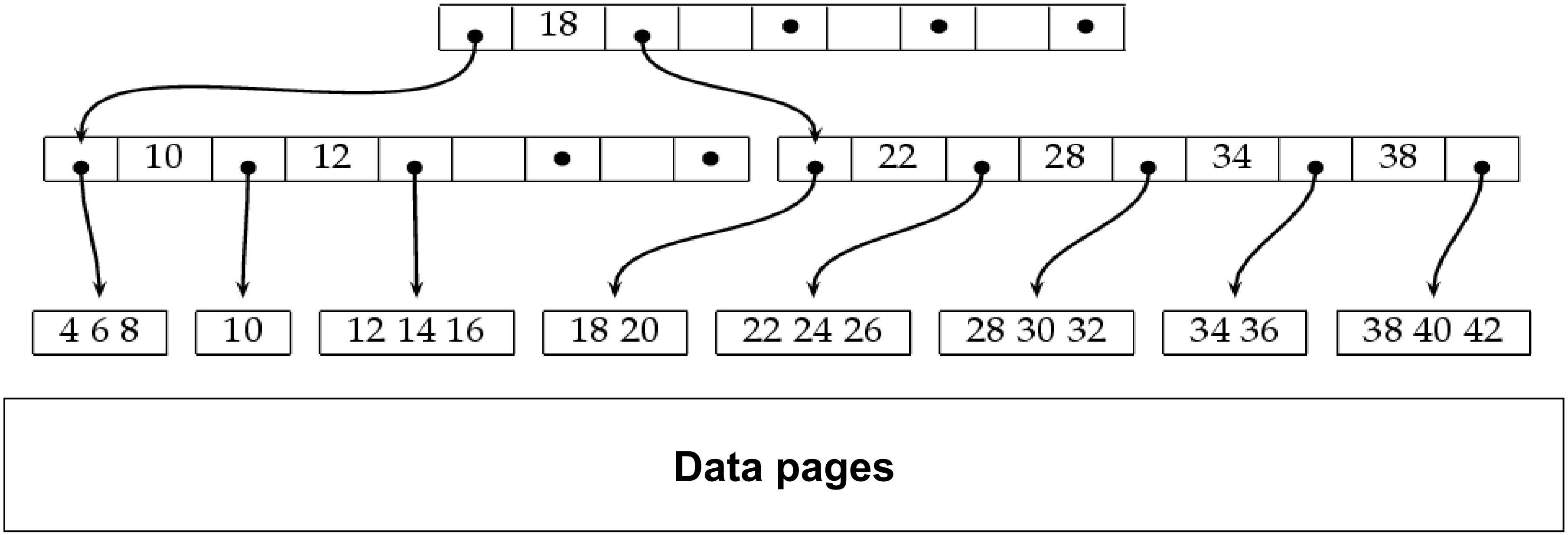
Other Access Methods

- B-tree variants: B^+-trees, B^*-trees
- B^+-trees used in data base management systems
- General scheme for access methods (used in B^+-trees, too):
  - Data keys stored only in leaves
  - Each entry in a non-leaf node stores
    - a pointer to a subtree
    - a compact description of the set of keys stored in this subtree

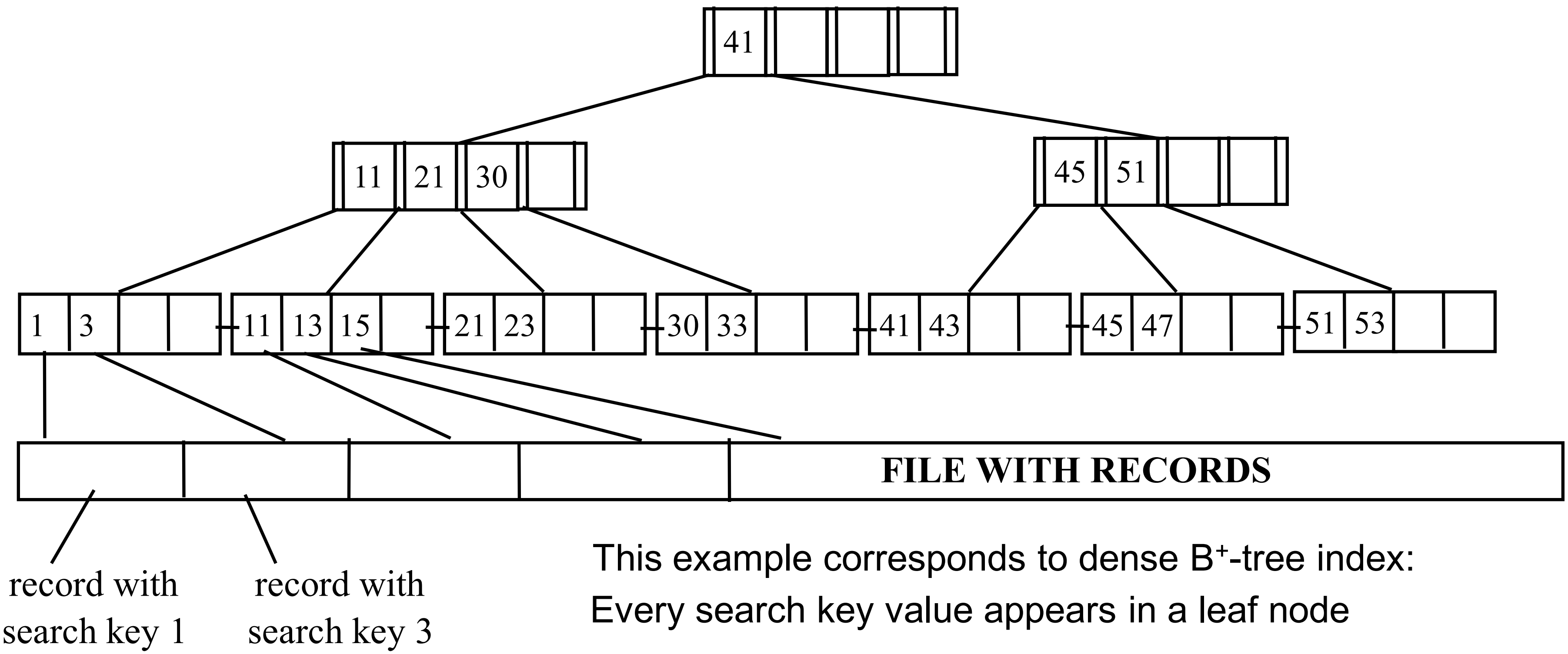
B^+ Tree Definition

- At most n sub-trees and $n-1$ keys
- At least $\left\lceil \frac{n}{2} \right\rceil$ sub-trees and $\left\lceil \frac{n}{2} - 1 \right\rceil$ keys
- Root: at least 2 sub-trees and 1 key
- The keys can be repeated in non-leaf nodes
- Only the leafs point to data pages
- The leafs are linked together with pointers
- The Most Widely Used Index

B+ Tree Example



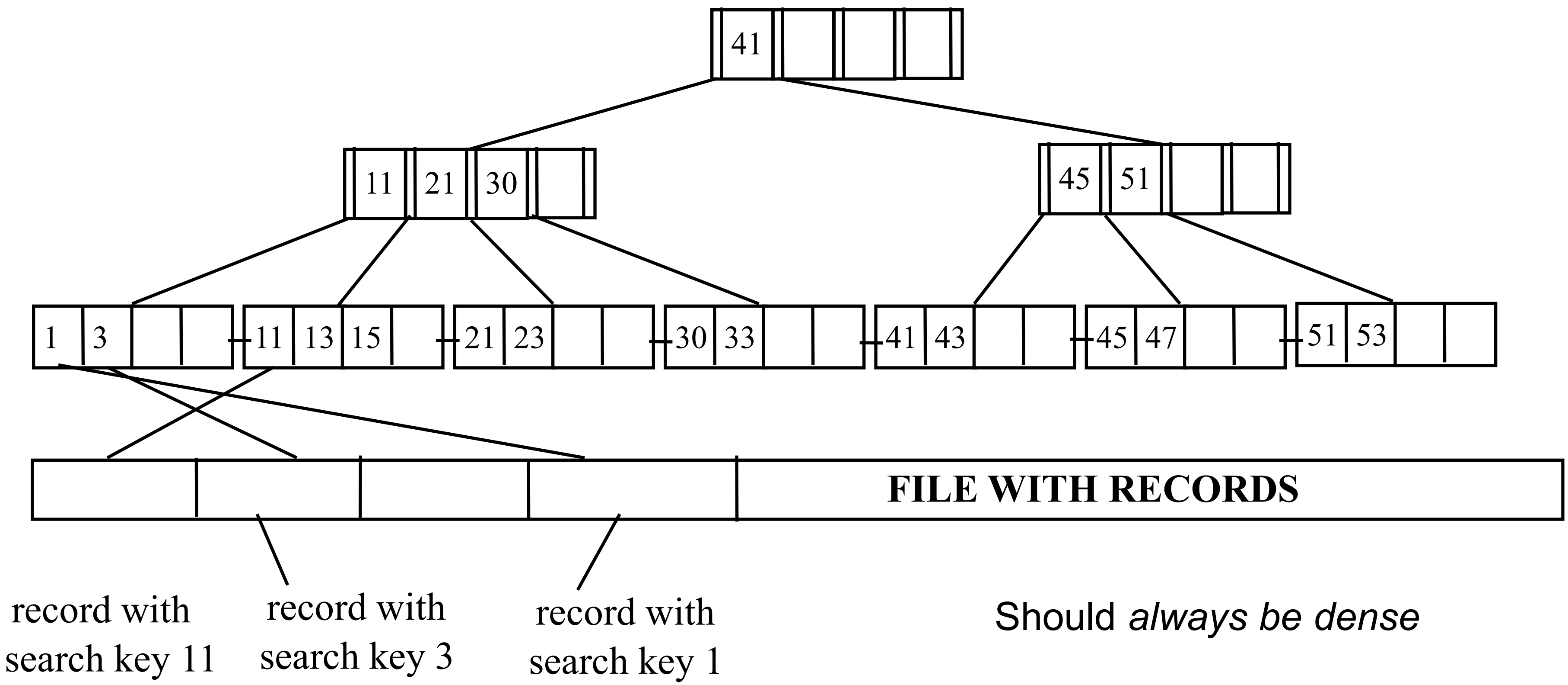
Example of Clustering (primary) B$^+$ Tree on Candidate Key



This example corresponds to dense B$^+$-tree index: Every search key value appears in a leaf node

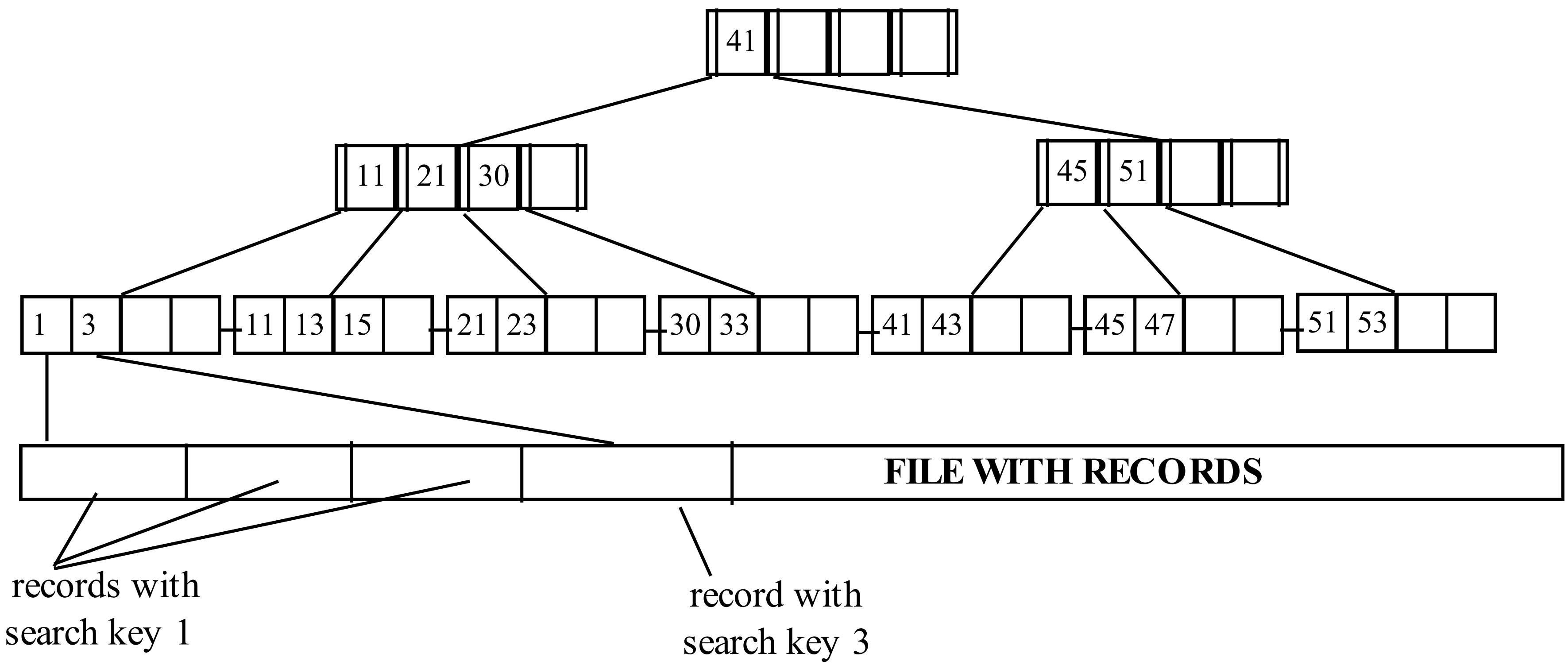
You may also have sparse B$^+$-tree, e.g., entries in leaf nodes correspond to pages

Example of Non-clustering (Secondary) B+ Tree on Candidate Key

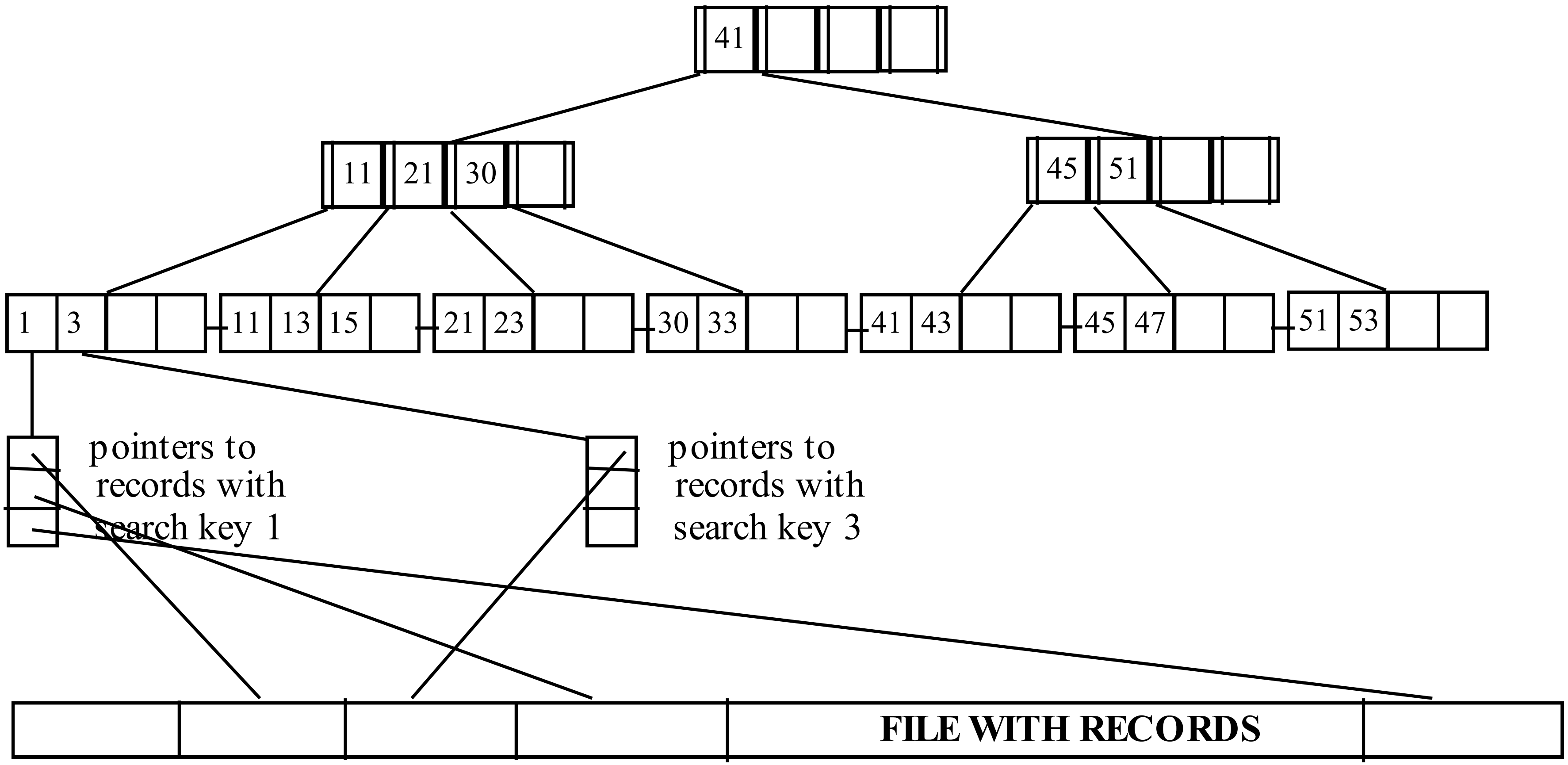


Should *always be dense*

Example of Clustering B+ Tree on Non-candidate Key



Example of Non-clustering B^+ Tree on Non-candidate Key



B$^+$ Tree Insertions

- Find appropriate leaf node
- If it is full:
  - allocate new node
  - split its contents and
  - insert separator key in father node
- If the father is full
  - allocate new node and split the same way
  - continue upwards if necessary
  - if the root is split create new root with two sub-trees

B^+ Tree Deletions

- Find and delete key from the leaf
- If the leaf has < $n/2$ keys

  a) borrowing if its neighbor leaf has more than $n/2$ keys update father node (the separator key may change) or

  b) merging with neighbor if both have < n keys
     - causes deletion of separator in father node
     - update father node

- Continue upwards if father node is not the root and has less than $n/2$ keys

B^+ Tree Performance

- *B^+ Trees* are better than B-trees for *range searching*
- *B Trees* are better for *random accesses*
- *The search must reach a leaf before it is confirmed*
  - internal keys may not correspond to actual record information (can be separator keys only)
  - insertions: leave middle key in father node
  - deletions: do not always delete key from internal node (if it is a separator key)

Applications of B+ Trees

- A B^+-tree can serve as a *dense index*: there is a (key,pointer) in leaf nodes for every record in a data file
  - search key in B^+-tree is the primary key of the data file
  - data file may or may not be sorted according to its primary key
- A B^+-tree can serve as a *sparse index*: there is a (key,pointer) in leaf nodes for every block of a data file that is sorted according to its primary key
- A B^+-tree can serve as a *secondary index*: if the file is sorted by a non-key attribute, there is a (key,pointer) in leaf nodes pointing to the first of records having this sort-key value
- Multiple occurrences of search keys are allowed in certain variants
  - must change the structure of internal nodes

B^+/B-Trees Comparison

- B-trees:
  - no key repetition,
  - better for random accesses (do not always reach a leaf),
  - data pages on any node□
- B^+-trees:
  - key repetition,
  - data page on leaf nodes only,
  - better for range queries,
  - easier implementation

Amortized Analysis

- We examined worst-case, average-case and best-case analysis performance
- In amortized analysis we care for the cost of one operation if considered in a sequence of n operations
  - In a sequence of n operations, some operations may be cheap, some may be expensive (*actual cost*)
  - The amortized cost of an operation equals the *total* cost of the n operations divided by n.

Amortized Analysis

- Think of it this way:
  - You have a bank account with 1000€ and you want to go shopping and purchase some items…
  - Some items you buy cost 1€, some items you buy cost 100€
  - You purchase 20 items in total, therefore…
  - …the amortized cost of each purchase is 5€

Amortized Analysis

- AMORTIZED ANALYSIS:
  - You try to estimate an upper bound of the **total work *T(n)*** required for a sequence of ***n*** operations…
  - Some operations may be cheap some may be expensive. Overall, your algorithm does ***T(n)* of work for n operations**…
  - Therefore, by simple reasoning, the ***amortized cost*** of each operation is ***T(n)/n***

Amortized Analysis

- Imagine $T(n)$ (the budget) being the number of CPU cycles a computer needs to solve the problem
- If computer spends $T(n)$ cycles for n operations, each operation needs $T(n)/n$ amortized time

Amortized Analysis

- We prove amortized run times with the accounting method. We present how it works with two examples:
  - Stack example
  - Binary counter example
- We describe Insert/Search/Delete/Join/Split in Splay Trees. Accounting method can show that these operations have $O(\log n)$ amortized cost (run time) and they are "balanced" just like AVL trees
  - We do not show the analysis behind the $O(\log n)$ run time

Amortized Analysis: Stack Example

Consider a stack ***S*** that holds up to ***n*** elements and it has the following three operations:

PUSH(S, x) .... pushes object ***x*** in stack ***S***

POP(S) .... pops top of stack ***S***

MULTIPOP(S, k) ... pops the ***k*** top elements of ***S*** or pops the entire stack if it has less than ***k*** elements

Amortized Analysis: Stack Example

- How much a sequence of n **`PUSH()`**, **`POP()`** and **`MULTIPOP()`** operations cost?
  - A **`MULTIPOP()`** may take $O(n)$ time
  - Therefore (a naïve way of thinking says that): a sequence of n such operations may take $O(n*n) = O(n^2)$ time since we may call n **`MULTIPOP()`** operations of $O(n)$ time each

With **accounting method** (**amortized analysis**) we can show a better run time of $O(1)$ per operation!

Amortized Analysis: Stack Example

- Accounting method:
- Charge each operation an amount of euros €:
  - Some money pays for the actual cost of the operation
  - Some is deposited to pay for future operations
  - Stack element credit invariant: 1€ deposited on it

Actual cost

PUSH	1
POP	1
MULTIPOP	min(k, S)

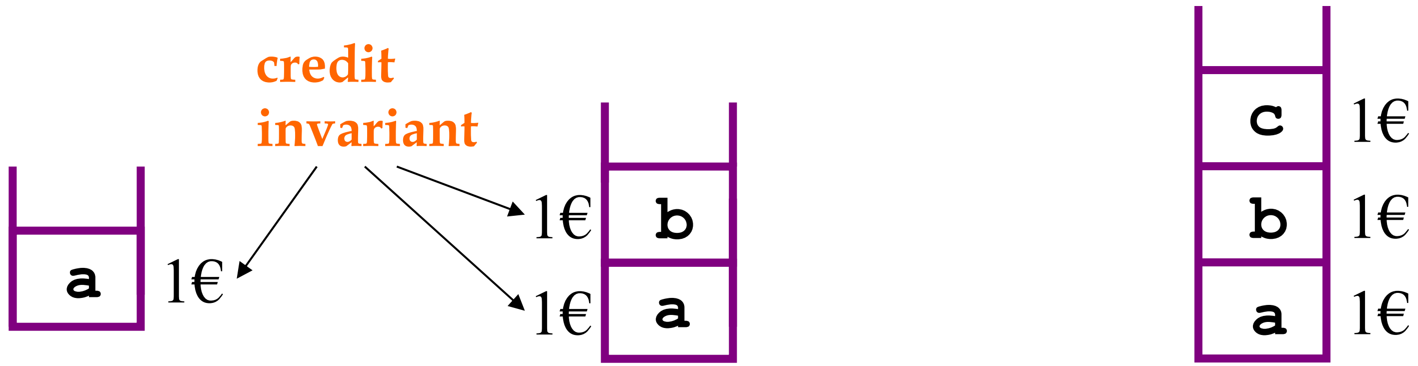
Amortized cost

PUSH	2
POP	0
MULTIPOP	0

Amortized Analysis: Stack Example

- In amortized analysis with accounting method we charge (amortized cost) the following €:
  - We let a POP() and a MULTIPOP() cost nothing
  - We let a PUSH() cost 2€:
    - 1€ pays for the actual cost of the operation
    - 1€ is deposited on the element to pay when/if POP-ed

Amortized Analysis: Stack Example



`Push(a)` = 2€

1€ pays for push and 1€ is deposited

`Push(b)` = 2€

1€ pays for push and 1€ is deposited

`Push(c)` = 2€

1€ pays for push and 1€ is deposited

`MULTIPOP()` costs nothing because you have the 1€ bills to pay for the pop operations!

Accounting Method

- We charge operations a certain amount of money
- We operate with a budget $T(n)$
  - A sequence of n `POP()`, `MULTIPOP()`, and `PUSH()` operations needs a budget $T(n)$ of at most $2n$ €
  - Each operation costs

$$T(n)/n = 2n/n = O(1) \text{ amortized time}$$

Binary Counter Example

- Let n-bit counter $A[n-1]\ldots A[0]$ (counts from 0 to 2^n):
  - How much work does it take to increment the counter n times starting from zero?
  - Work $T(n)$: how many bits do you need to flip (0→1 and 1→0) as you increment the counter …

Binary Counter Example

```
INCREMENT(A)
    i=0;
    while i < length(A) and A[i]=1 do
        A[i]=0;
        i=i+1;
    if  i < length(A) then
        A[i] = 1
```

This procedure *resets* the first ***i***-th sequence of 1 bits and *sets* ***A***[***i***] equal to 1 (ex. 0011 → 0100, 0101 → 0110, 0111 → 1000)

Binary Counter Example

4-bit counter:

Counter value	COUNTER	Bits flipped (work T(n))
0	0 0 0 **0**	0
1	0 0 0 **1**	1
2	0 0 **1 0**	3
3	0 **0** 1 **1**	4
4	0 **1 0 0**	7
5	0 1 0 **1**	8
6	0 1 **1 0**	10
7	0 1 1 **1**	11
8	**1 0 0 0**	15
	$A_3A_2A_1A_0$	

highlighted are bits that flip at each increment

Binary Counter Example

- A naïve approach says that a sequence of n operations on a n-bit counter needs $O(n^2)$ work
  - Each INCREMENT() takes up to $O(n)$ time. n INCREMENT() operations can take $O(n^2)$ time
- Amortized analysis with accounting method
  - We show that amortized cost per INCREMENT() is only $O(1)$ and the total work $O(n)$
  - OBSERVATION: In example, $T(n)$ (work) is never twice the amount of counter value (total # of increments)

Binary Counter Example

- Charge each 0→1 flip 2€ in line 6
  - 1€ pays for the 0→1 flip in line 6
  - 1€ is deposited to pay for the 1→0 flip later in line 3
- Therefore, a sequence of n `INCREMENTS()` needs

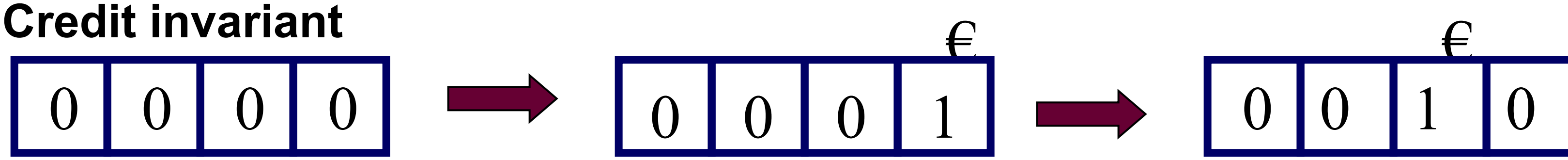
$$T(n) = 2n \text{ €}$$

- …each INCREMENT() has an amortized cost of

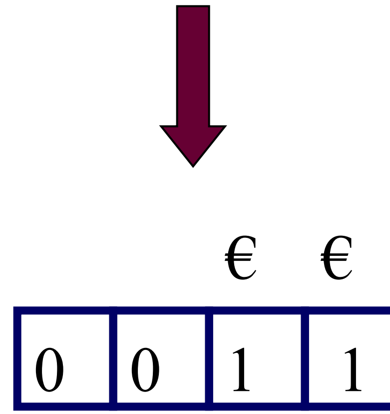
$$2n/n = O(1)$$

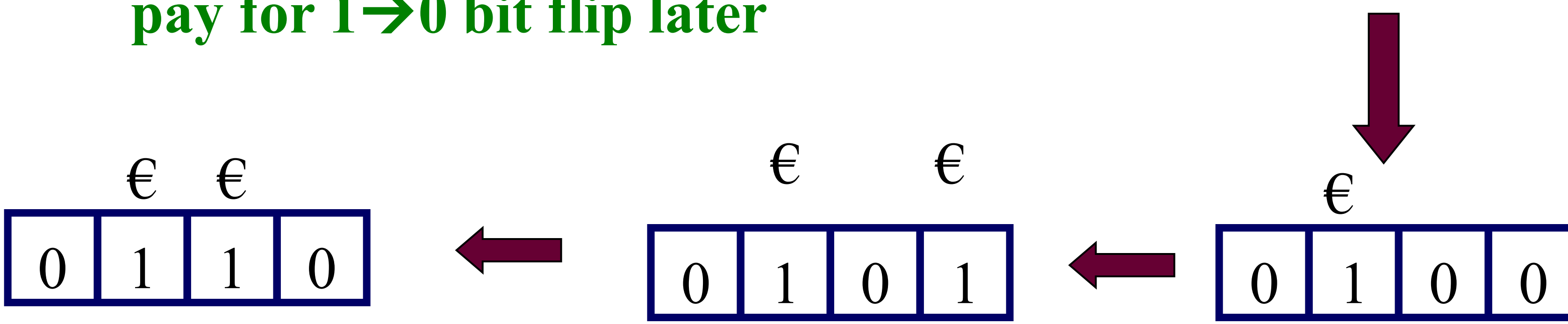
Binary Counter Example

Credit invariant



- **Charge 2€ for every 0→1 bit flip. 1€ pays for the actual operation**
- **Every 1 bit has 1 € deposited to pay for 1→0 bit flip later**





Splay Trees

- Splay Trees: Self-Adjusting (balanced) Binary Search Trees (Sleator and Tarjan, AT&T Bell 1984)

- They have $O(\log n)$ amortized run time for
  - SplayInsert()
  - SplaySearch()
  - SplayDelete()
  - Split()
    - splits in 2 trees around an element
  - Join()
    - joins two ordered trees

These are expensive operations for AVLs

Splay Trees

- A splay tree has the binary search tree property:

  $$\textit{left subtree} < \textit{parent} < \textit{right subtree}$$

- Operations are performed similar to BSTs. At the end we always do a *splay* operation

Splay Trees: Basic Operations

SplayInsert(x)

- insert *x* as in BST;
- *splay(x);*

SplaySearch(x)

- search for *x* as in BST;
- if you locate *x* *splay(x);*

SplayDelete(x)

- delete *x* as in BST; if successful then
- *splay()* at successor or predecessor of *x*;

Splay Trees: Basic Operations

- A splay operation moves an element to the root through a sequence of zig, zig-zig, and zig-zag rotation operations
  - rotations preserve BST order

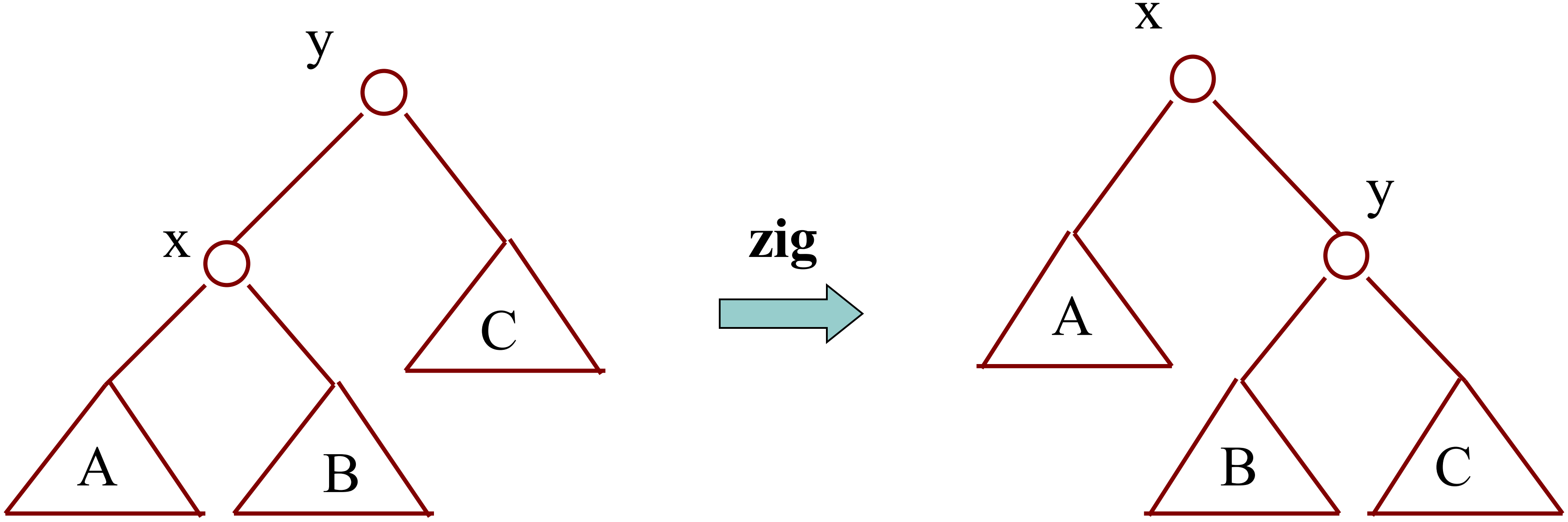
Splay(x)

- moves node *x* at the root of the tree

perform *zig, zig-zig, zig-zag* rotations until the element becomes the root of the tree

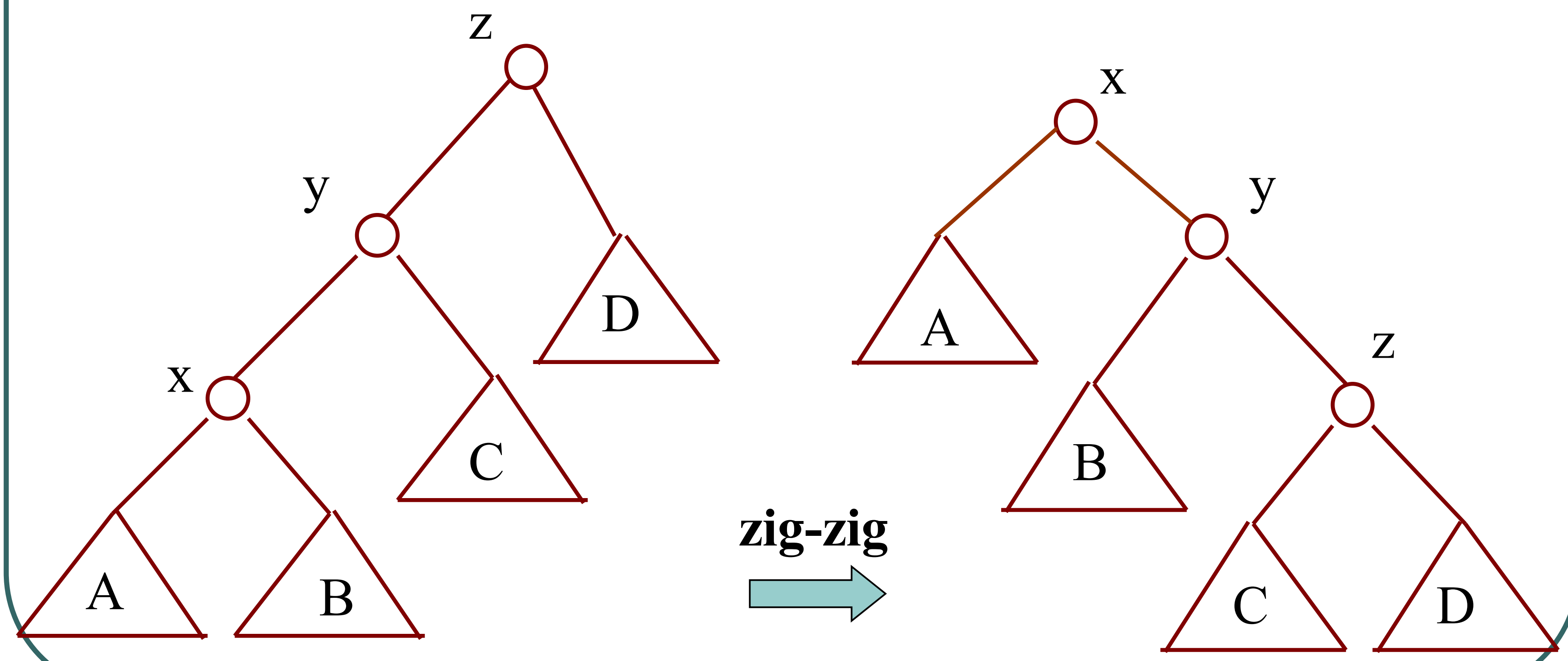
Splay Trees: Splay(x) Operation

ZIG (1 rotation):



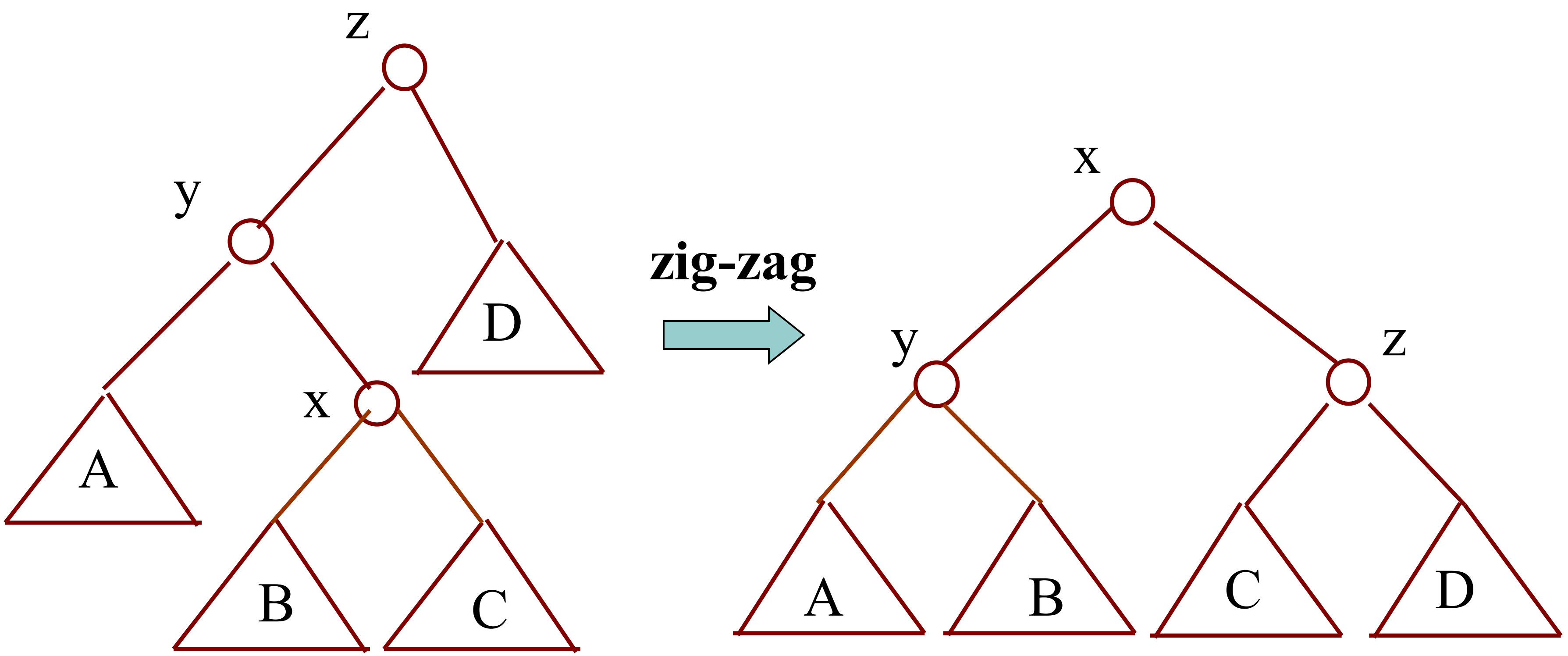
Splay Trees: Splay(x) Operation

ZIG-ZIG (2 rotations):



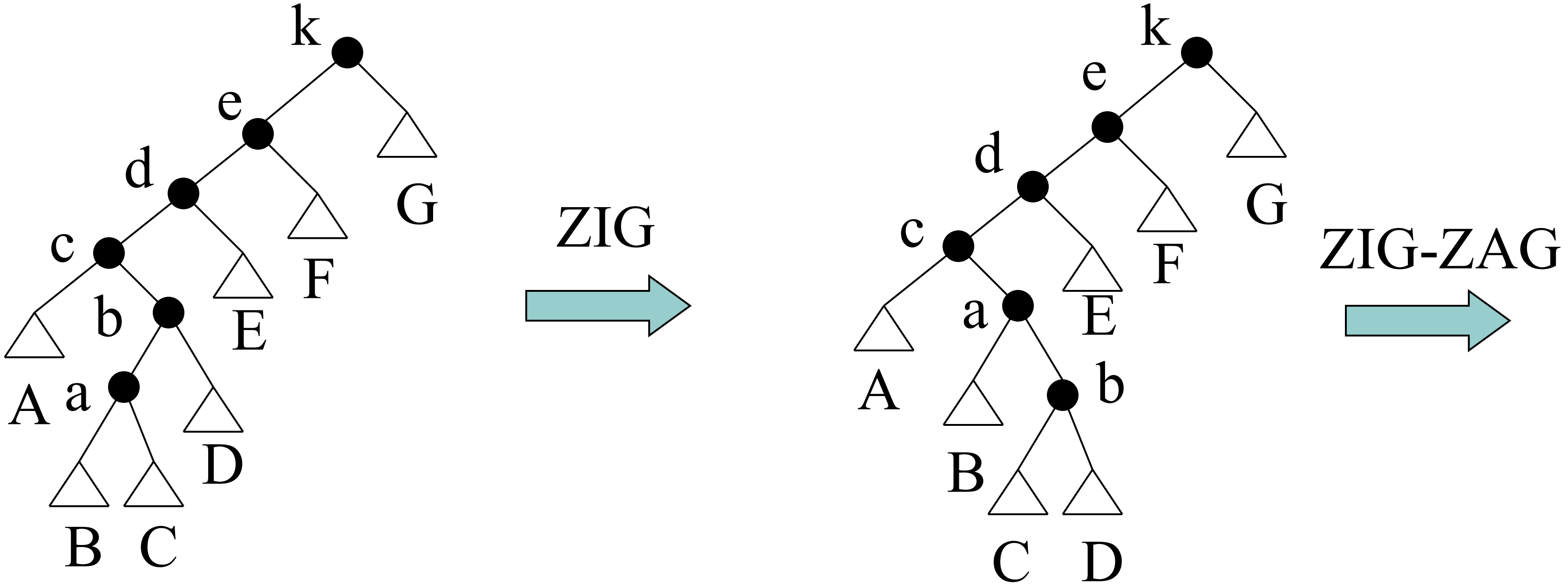
Splay Trees: Splay(x) Operation

ZIG-ZAG (2 rotations):

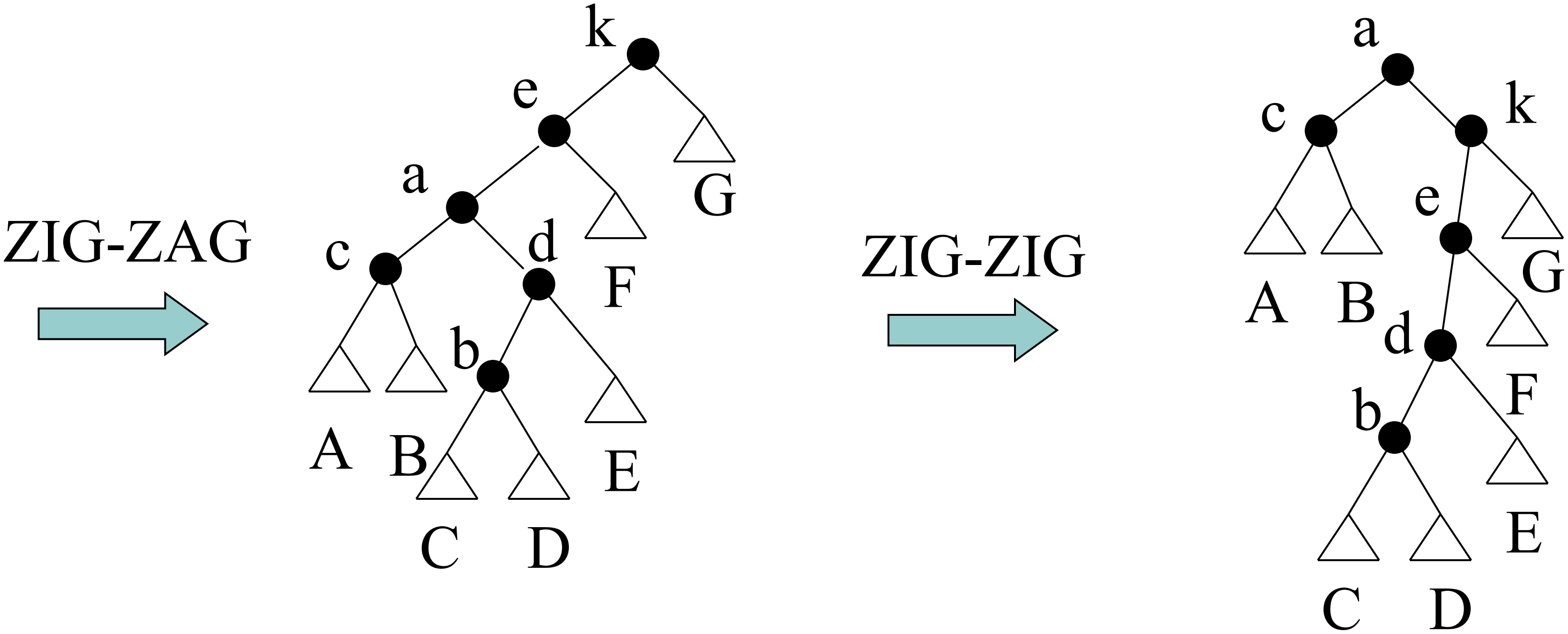


Splaying: Example

Splaying at a node **splay(a)**:



Splaying: Example (cont.)



Splaying

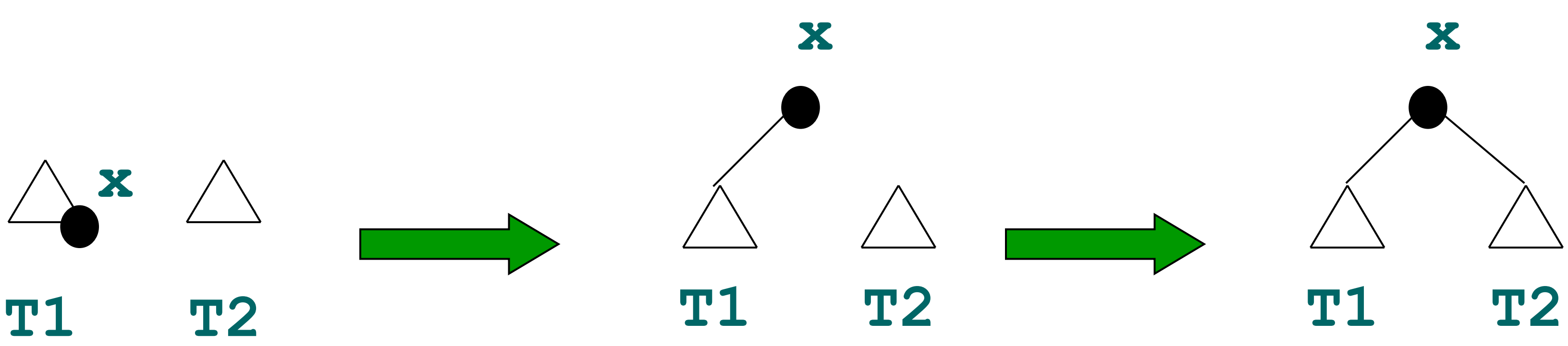
- We observe that the originally “unbalanced” splay trees become more “balanced” after splaying
- In general, one can prove that it costs $3\log n$ € to ***splay()*** at a node.
- Therefore, in an amortized sense, splay trees are balanced trees.
- Demos:
  - http://www.link.cs.cmu.edu/splay/
  - http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/SplayTree-Example.html

Splay Trees: Join(T1,T2,x)

```
Join(T1, T2, x)

    - every element in T1 is < T2
    - x largest element (rightmost element) of T1
    - it returns a tree containing x, T1 and T2

SplayMax(x);   /* this splays max to root */
Splay(x);
right(x) = T2;
```

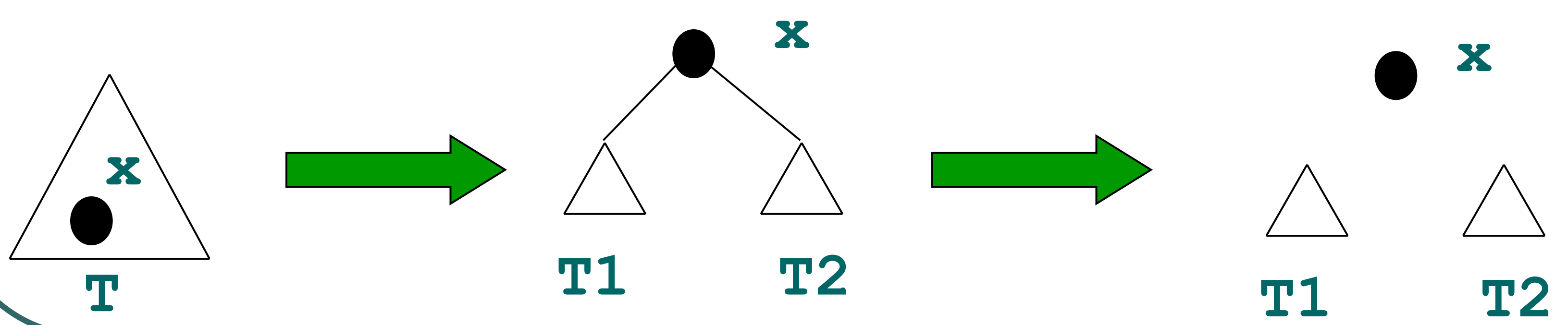


Splay Trees: Split()

Split(T,x)

- *it takes a single tree T and splits it into two trees T1 and T2*
- *T1 contains x and elements of T smaller than x*
- *T2 contains elements of T larger than x*

```
SplaySearch(x);
Splay(x);        /* this brings x to root */
return left(x), x, right(x);
```



Splay Trees: Complexity

- The amortized complexity of the following is $O(\log n)$:
  - SplayInsert()
  - SplaySearch()
  - SplayDelete()
- That is in a sequence of n insert/search/delete operations on a splay tree each operation takes $O(\log n)$ *amortized time*
- Therefore, `Split()` and `Join()` also take $O(\log n)$ *amortized time*,

`Split()` and `Join()` *CANNOT* be done in $O(\log n)$ time with other balanced tree structures such as AVL trees

Reading

- Sleator and Tarjan article available as http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf
- CLRS, chapter 17 section 2