Containers

Stanislav Protasov

Agenda

- Lists, Sorted list
  - Unrolled linked list
  - Skip list
- Set, Sorted set
  - Search trees
- Persistent data structures
  - Techniques
  - V-List

Abstract data types and their implementations

List

List

Countable number of ordered non-unique values. Finite sequence.

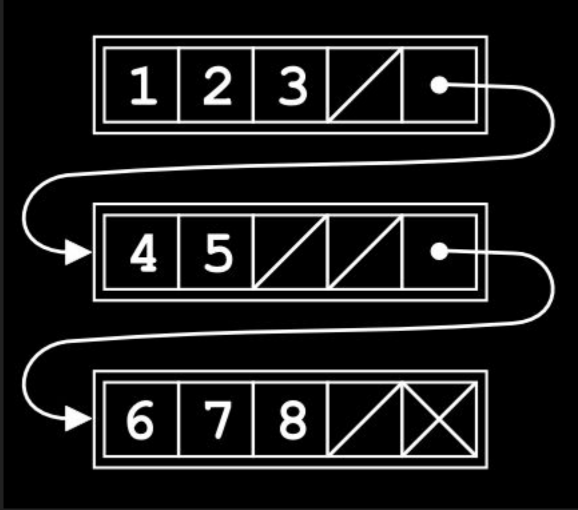
	Array List	Linked List
creating an empty list		
testing a list is empty		
prepending an entity		
appending an entity		
determining the "head" of a list		
accessing the element at a given index		

List

Countable number of ordered non-unique values. Finite sequence.

	Array List	Linked List
creating an empty list	O(1)	O(1)
testing a list is empty	O(1)	O(1)
prepending an entity / inserting at position	O(N)	O(1)
appending an entity	O(N), $O_A(1)$	O(1)
determining the "head" of a list	O(1)	O(1)
accessing the element at a given index	O(1)	O(N)

Unrolled ***linked list*** (1994)



Increases cache performance

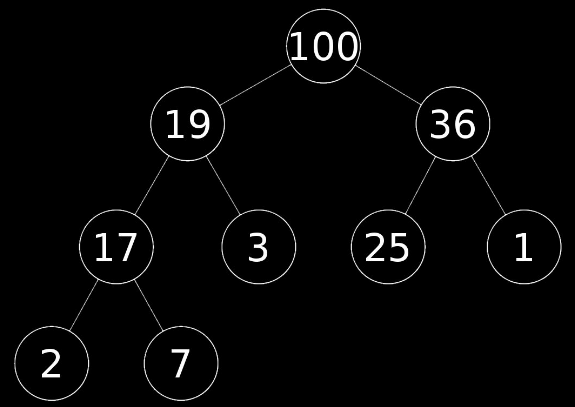
Decreases memory overhead

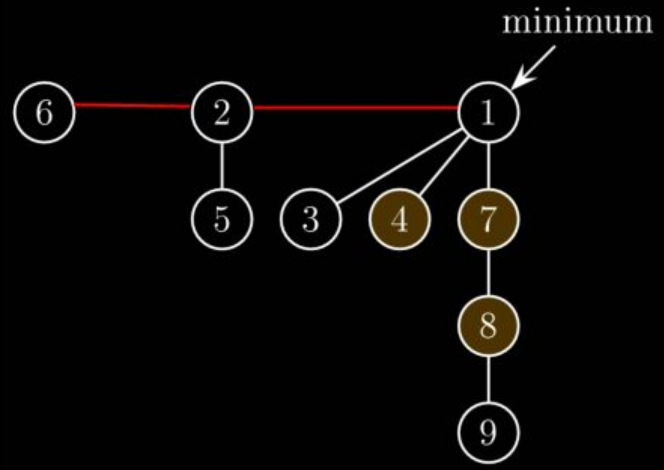
Ideas:

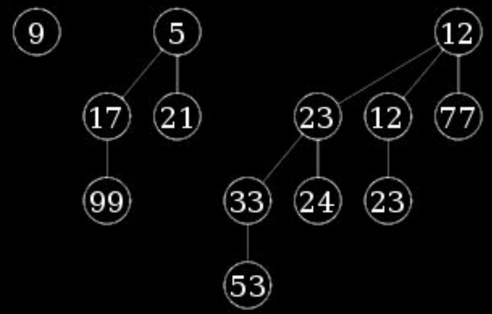
- Nodes capacity is constant ***k***. Thus, search is O(n/k).
- Each node stores it's size and ~ cache line of data.
- Nodes are preserved at worst case half-full.
  - On insert overflow: **split** a node into 2 of (k/2)+1 and (k/2) in +O(k).
    O(n/k + k) for insert
  - On delete: either **steal** from the next or **merge** is possible in +O(k).
    O(n/k + k) for insert

Priority queue and Sorted list

Heaps (priority queues)







Sorted list: new requirements for the list

1. Fast “**TOP N**” operation (including peak() and pop())
2. Sublinear `search`(x)
    a. Sublinear insert
    b. Sublinear delete

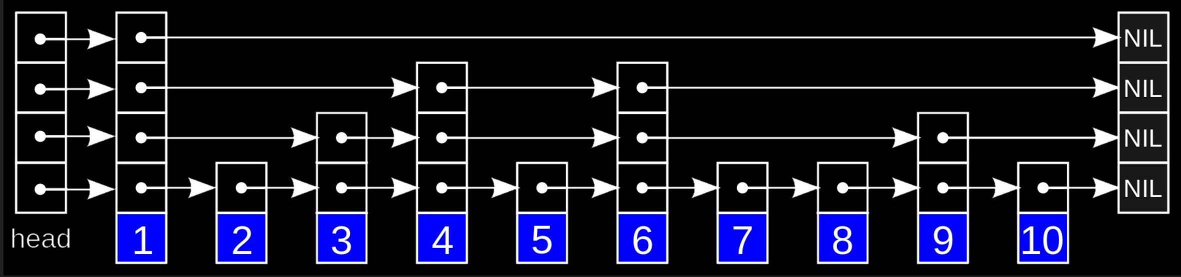
	Array List	**Linked List**
Search		
Insert		
Delete		

Sorted list: new requirements for the list

1. Fast “**TOP N**” operation (including peak() and pop())
2. Sublinear **`search`**(x)
    a. Sublinear insert
    b. Sublinear delete

	Array List	**Linked List**
Search	O(log(N))	O(N)
Insert	O(N)	O(N)
Delete	O(N)	O(N)

Skip List



1) Based on the linked list
   a) Instead of `Node* next` it has `Node*[LEVELS] next`;
2) Introduces idea of search “shortcuts”
3) **Probabilistic insertion algorithm**
   a) Insert into basic linked list
   b) Increment a level. If maximum -
   c) Toss a coin (***p*** = 0.5 or any other). **If “win” - goto (b)**
4) ***Expected*** search time for a list with n elements: $T_E(n) = \frac{1}{p} \log_{1/p} n$
   a) `if node.next[level] is null or node.next[max_level].value > x`
      i) `then: level-- (or return None on level 0)`
      ii) `else: node = node.next[level]`

Set, multiset, map and sorted

Set+

1) Basic operations
    a) Union, Intersection, Difference, IsSubset

	HashTable	Union-Find
Union (n, k)		
Intersection (n, k)		
Difference (n, k)		
IsSubsetOf (n, k)		

Set+

1) Basic operations
   a) Union, Intersection, Difference, IsSubset
2) IsElementOf
3) Iterate/Enumerate*, **
4) Add(x), Remove(x)

	HashTable	Union-Find
Union (n, k)	O(min(n+k))	O(1)
Intersection (n, k)	O(min(n, k))*	O(min(n,k))**
Difference (n, k)	O(n+k)*	O(n+k)**
IsSubsetOf (n, k)	O(n)*	O(n)**

	HashTable	Union-Find
IsElementOf		
Iterate		
Add		
Remove		

Set+

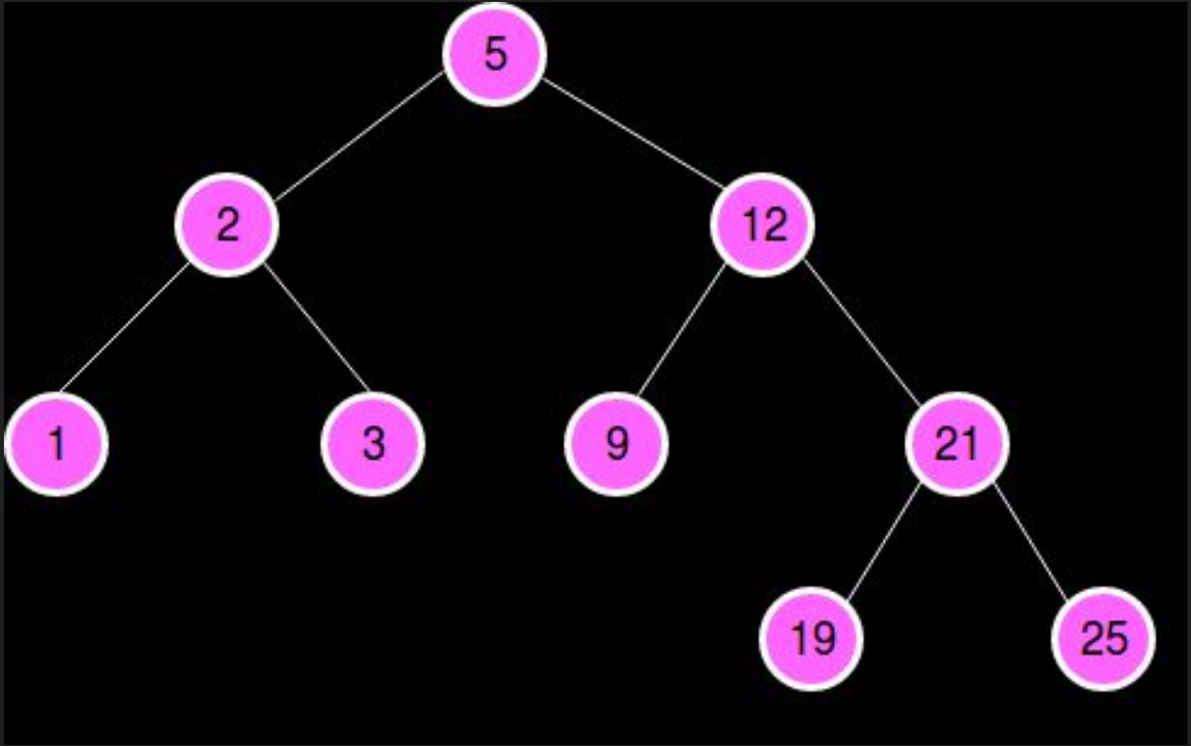
1) Basic operations
   a) Union, Intersection, Difference, IsSubset
2) IsElementOf
3) Iterate/Enumerate*, **
4) Add(x), Remove(x)

	HashTable	Union-Find
Union (n, k)	O(min(n+k))	O(1)
Intersection (n, k)	O(min(n, k))*	O(min(n,k))**
Difference (n, k)	O(n+k)*	O(n+k)**
IsSubsetOf (n, k)	O(n)*	O(n)**

	HashTable	Union-Find
IsElementOf	O(1)	O(1)
Iterate	O(n)*	--
Add	$O_A(1)$	O(1)
Remove	O(1)	--

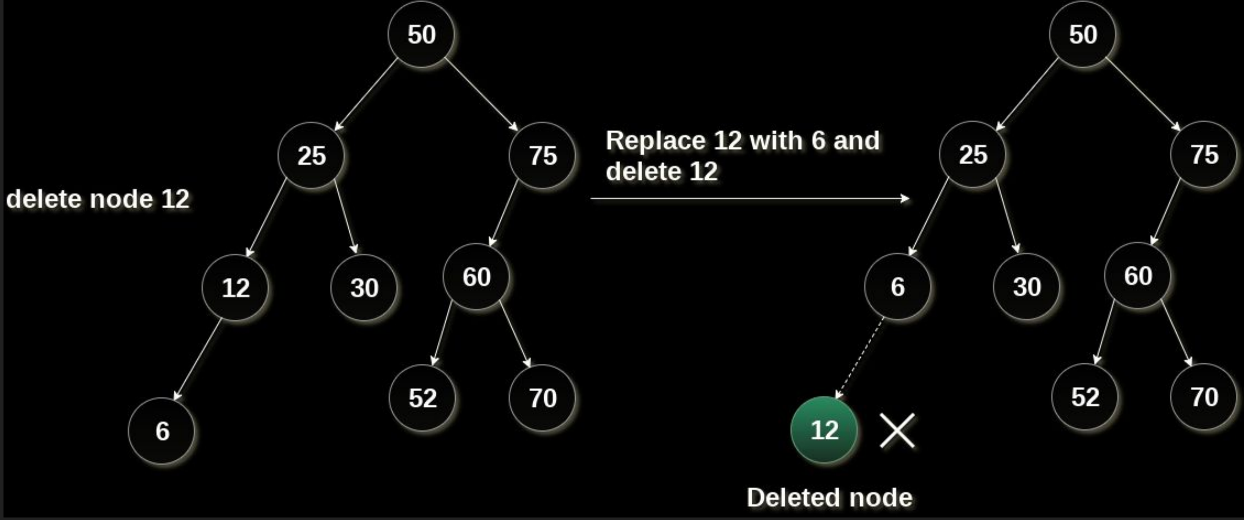
Sorted sets: [binary] search trees

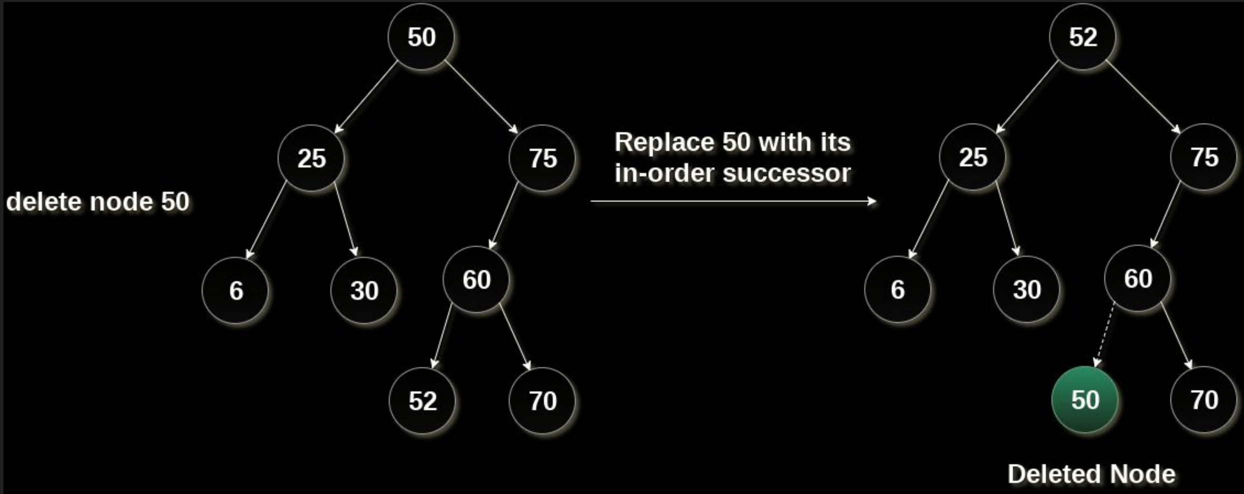
Idea



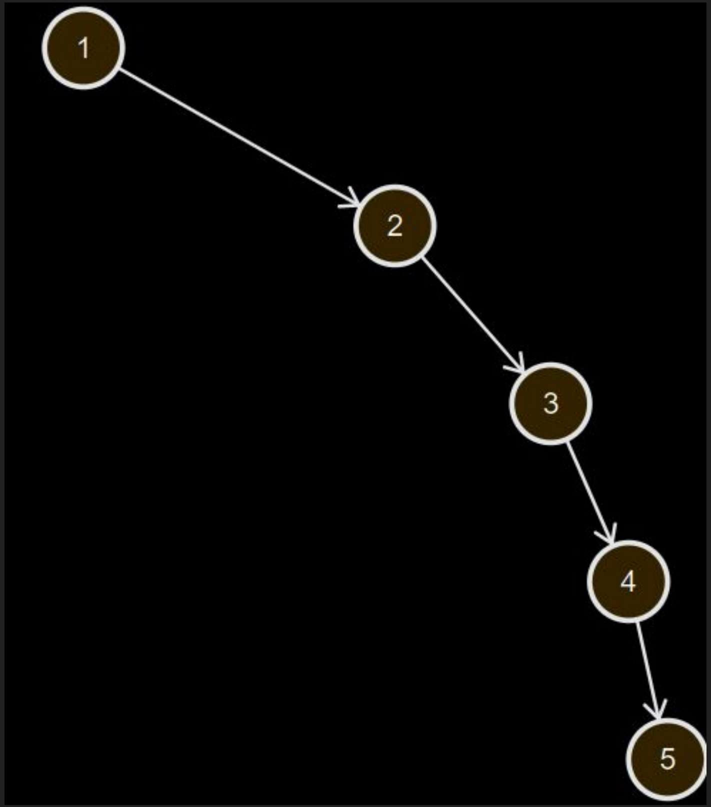
Non trivial delete

1) Find
2) If no ancestors - trivial
3) Single child - pull it up
4) 2 children
    a) Find pre/in-order ancestor and replace with it





Skewed binary tree



Balanced (self-balancing) binary search trees

AVL-tree (Adelson-Velskii, Landis, 1962)

Restriction: subtrees height differ by not more then 1.

```
|h(left) - h(right)| ≤ 1
```

Thus, worst case is:

$$n_0 = 0,$$

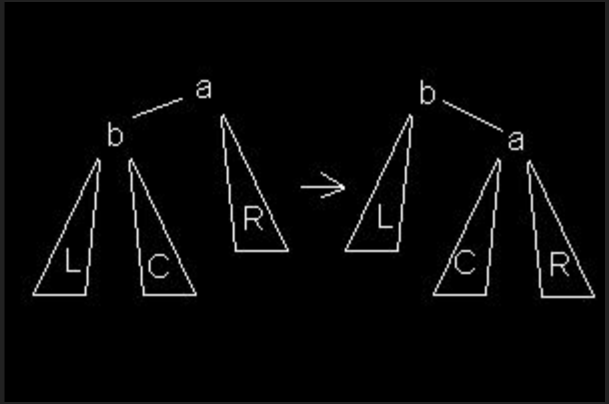
$$n_1 = 1,$$

$$n_h = n_{h-2} + n_{h-1} + 1.$$

$$N_h = \Phi_{h+2} - 1$$

$$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} \right\rceil$$

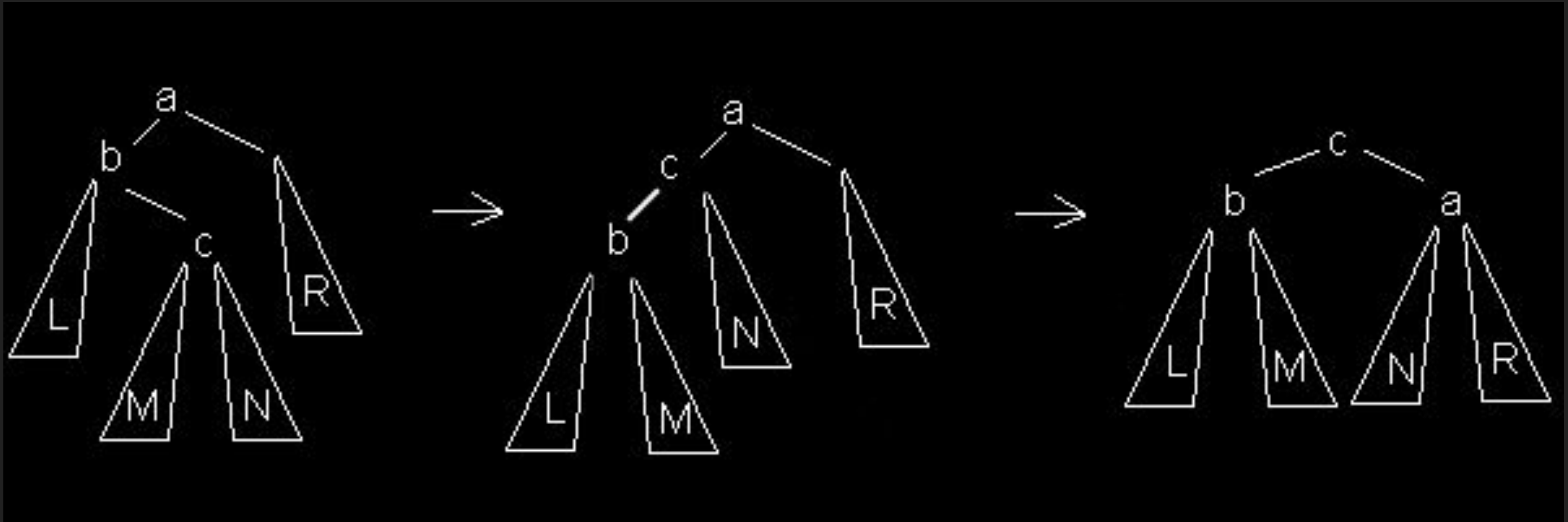
AVL Operations



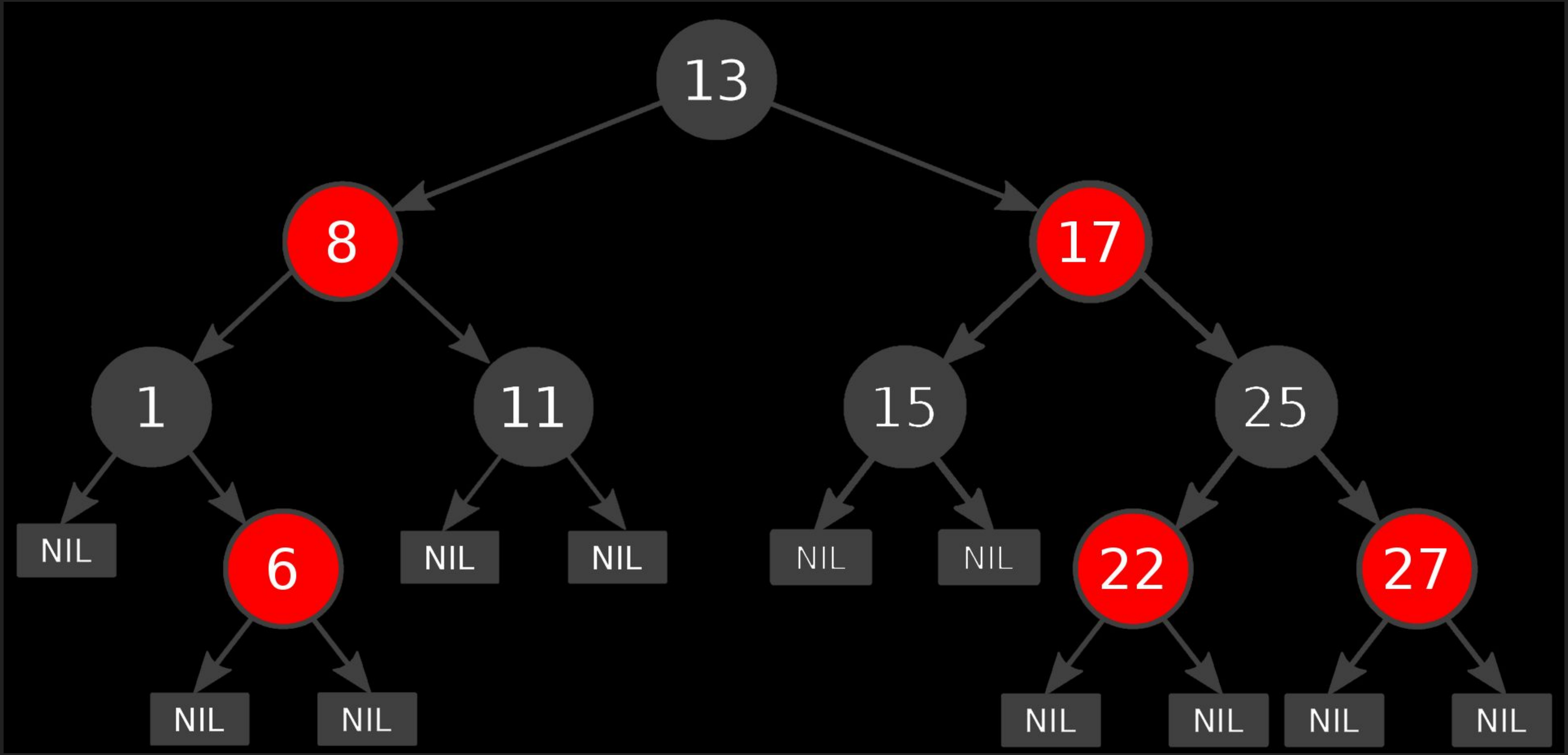
Search - trivial BT search

Removal, Insertion:

Left and right ***rotations and "big rotations"***



Red-Black Trees (Bayer, 1972)



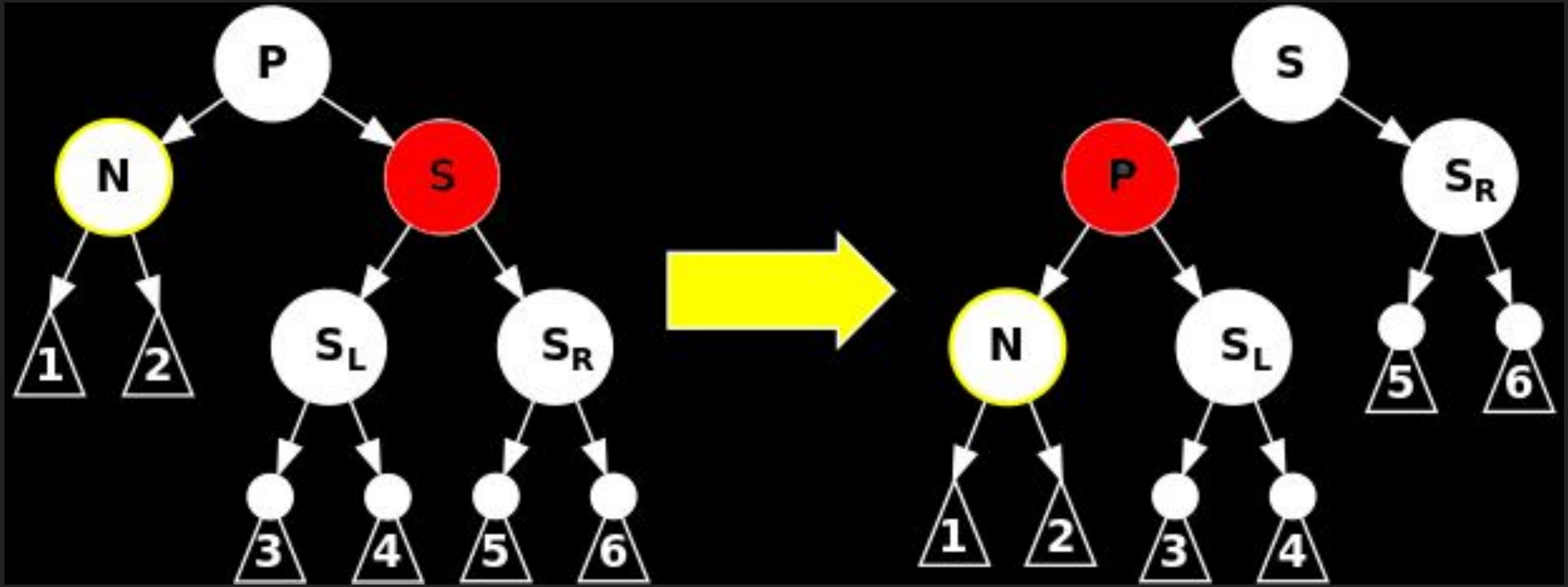
RB-tree restrictions

1. Each node is either **red** or black.
2. The **root** is black. This rule is sometimes omitted. Since the root can always be changed from red to black
3. All **leaves** (NIL) are black.
4. If a node is **red**, then both its **children** are black.
5. Every **path** from a given node to any of its descendant NIL nodes contains the **same number of black nodes**.

$$n \geq 2^{\frac{h(\mathrm{root})}{2}} - 1 \leftrightarrow \log_2(n+1) \geq \frac{h(\mathrm{root})}{2} \leftrightarrow h(\mathrm{root}) \leq 2\log_2(n+1).$$

RB-trees: techniques

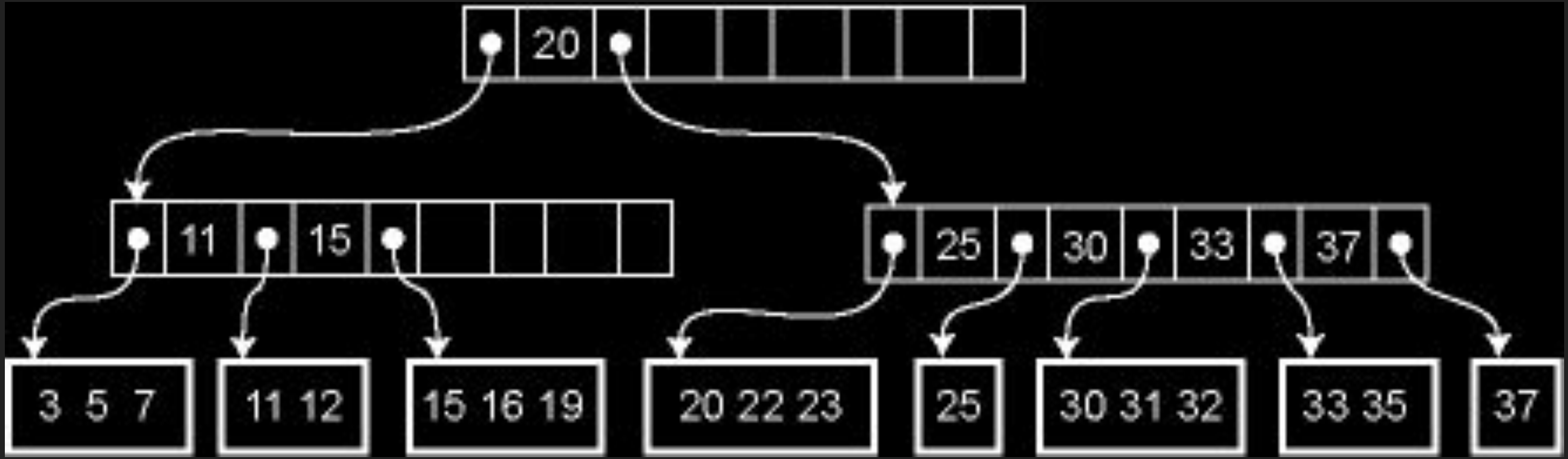
Restoring RB-properties (rotations, restructuring) requires **O(log(N))** or **O_A(1)**



Non-binary search tree: B-trees

Takes best from Unrolled linked lists and search trees:

1. Cache and HDD friendly
2. Minimizes restructuring (SPLIT, MERGE, BORROW)
3. Modifications are used in file systems and databases



Persistent lists

Techniques

Copy on Write

Fat node

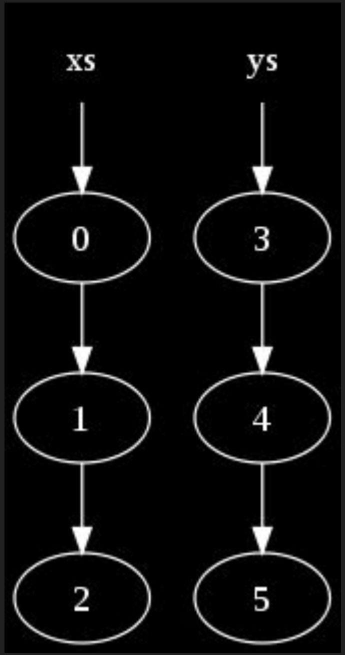
Path copying

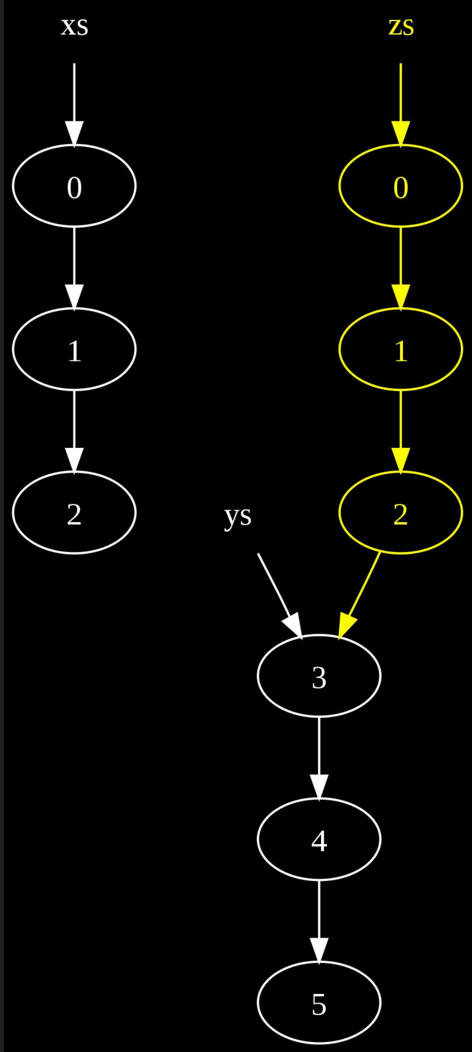
A combination fat node and path copying

Persistent singly-linked list

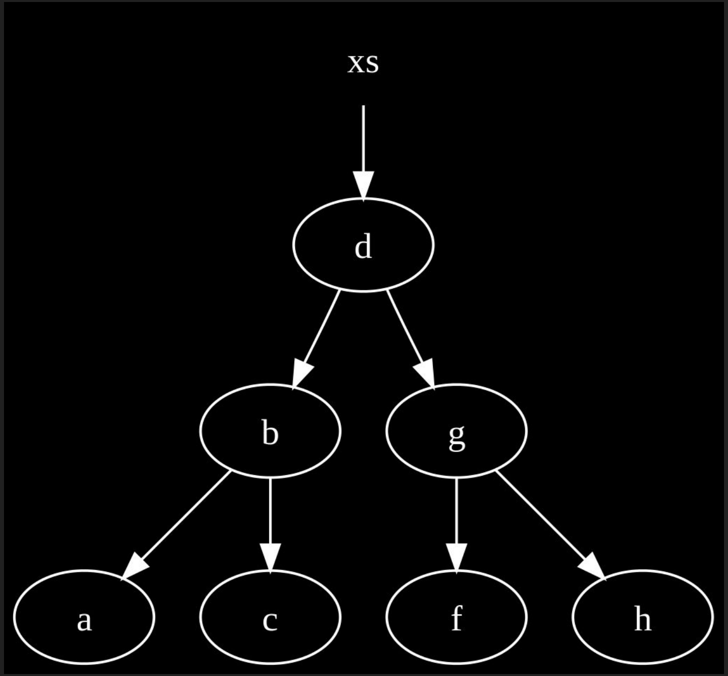
If we produce an operation, we need to preserve original structure unchanged. If we cannot, we need to copy. Singly-linked Lists can be constructed and operated in persistent way with cons, car and cdr operations.

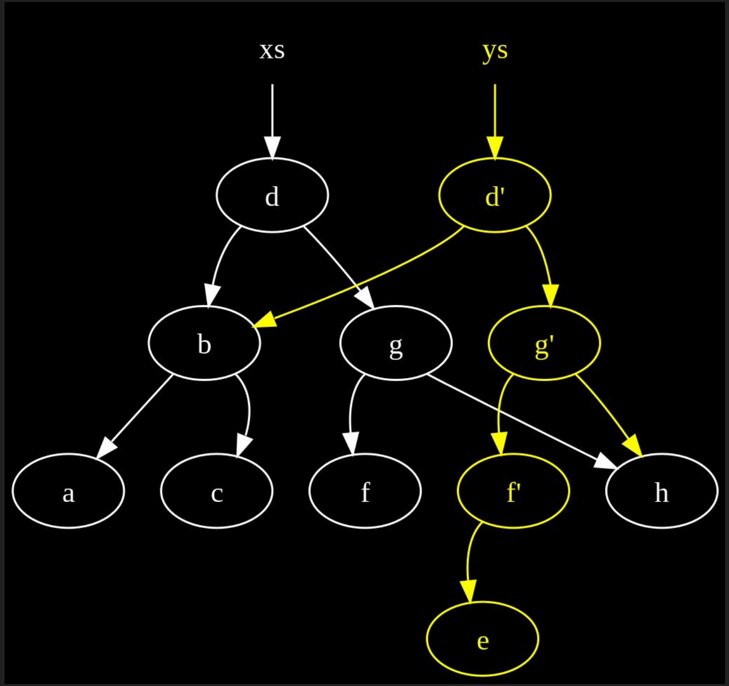
- A[k] = O(k)
- A[1:k] = O(1)
- Prepend = O(1)
- Append, insert before k = O(k)





Persistent search trees

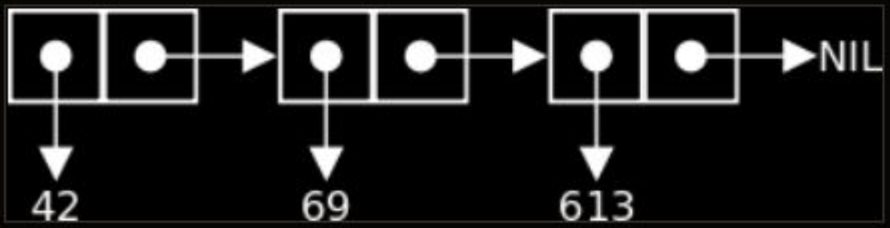




VList (Bagwell, 2002)

cons

CONS: **cons**tructs memory objects which hold two values




```
(cons 42 (cons 69 (cons 613 nil)))
```

and written with `list`:

```
(list 42 69 613)
```

```
(cons (cons 1 2) (cons 3 4))
```

```
    *
   / \
  *   *
 / \ / \
1  2 3  4
```

car and cdr

car extracts first element of the pair, created by cons, cdr extracts the second

```
(cadr '(1 2 3)) = (car (cdr '(1 2 3))) = 2

(caar '((1 2) (3 4))) = (car (car '((1 2) (3 4)))) = 1
```

When cons cells are used to implement **singly linked lists** (rather than trees and other more complicated structures), the car operation returns the first element of the list

VList

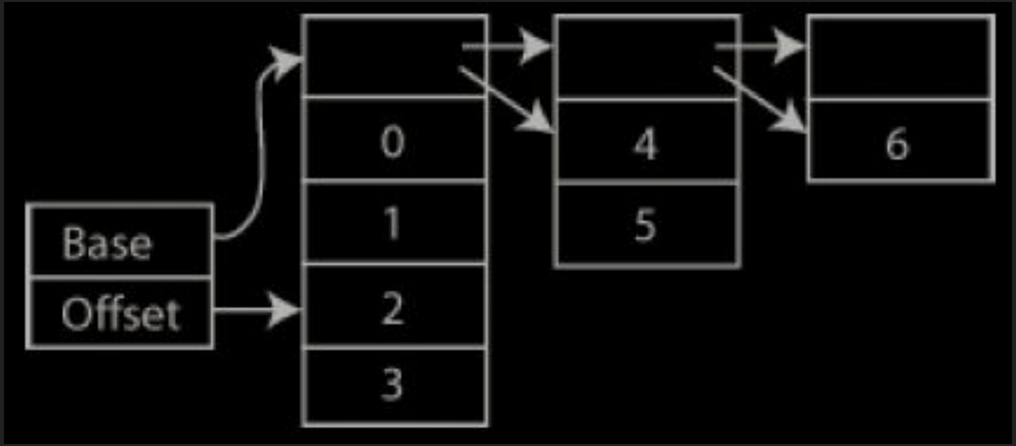
- **`A[k]`** - $O_A(1)$ average, O(log n)
- **Prepend** - $O_A(1)$ average
- `A[1:k]` (**cdr**) - O(1)
- `len(A)` - O(log n)

- While immutability is a benefit, it is also a drawback, making it inefficient to **modify elements in the middle of the array**
- **`A[-1]` Access near the end** of the list can be as expensive as O(log n)
- **Wasted space** in the first block is proportional to n. This is similar to linked lists

VList

structure of a VList can be seen as a singly-linked list of arrays whose sizes decrease geometrically.

A[k] timing comes from sum of geometric series



Any particular reference to a VList is actually a <base, offset> pair indicating the position of its first element

VList operations

A[0] (car) = trivial lookup O(1)

A[k] (car+cdr) = iterative process with O(log(N))

remove (cdr) = increase the offset | increase the base, offset = 0 - O(1)

prepend (cons) = insert in front array or add new *2*r* level $O_A(1)$

insert before ***k*** (cons in the middle) = see linked list example. We create new front array, place there an element, pointing to ***k***'s base+offset - O(malloc(n - k)) for allocation

8.11 - midterm exam

Please, don’t be late