Premaster Course Algorithms 1

Chapter 3: Elementary Data Structures

Christian Scheideler

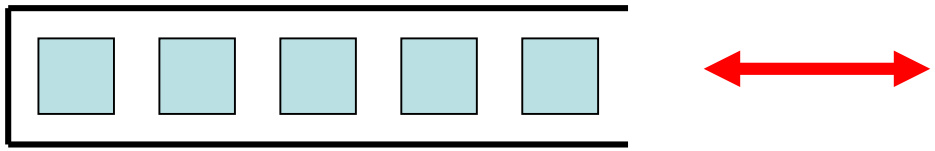
SS 2019

Overview

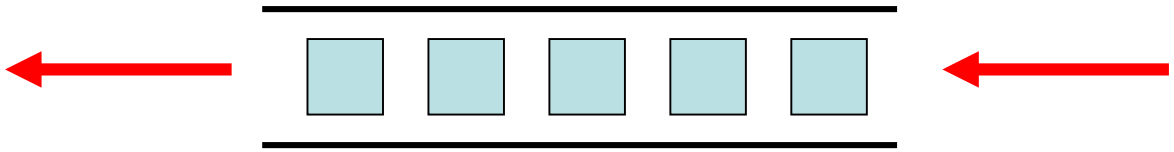
- Basic data structures
- Search structures (successor searching)
- Dictionaries (exact searching)

Stacks and Queues

Stack: implements the LIFO (last-in-first-out) rule, i.e., the youngest element will be removed first



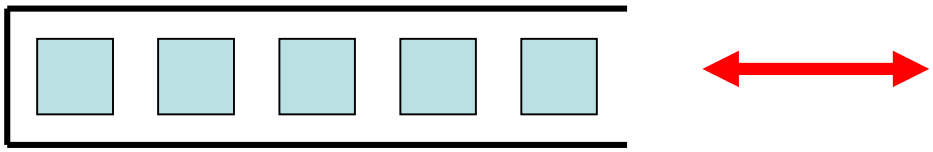
Queue: implements the FIFO (first-in-first-out) rule, i.e., the oldest element will be removed first



Stack

Stack operations:

- Push(S,x): insert element x into stack S
- Pop(S): remove youngest element from stack
- Empty(S): checks if stack is empty

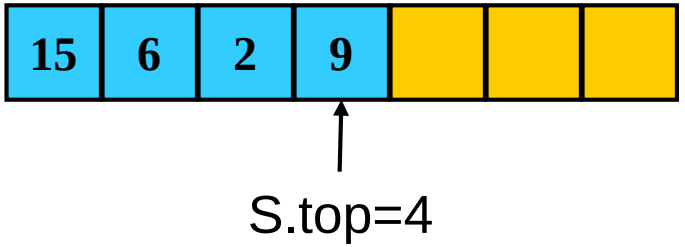


Simplest implementation:

- Array S[1..N] (#elements in S never more than N)
- S.top: number of elements currently in S[1..N]

Stack

Example:



Implementation:

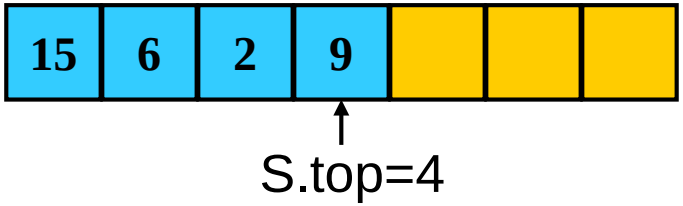
Push(S,x):
1. if S.top=N then error „overflow“
2. else
3. S.top:=S.top+1
4. S[S.top]:=x

Pop(S):
1. if Empty(S) then error „underflow“
2. else
3. S.top:=S.top-1
4. return S[S.top+1]

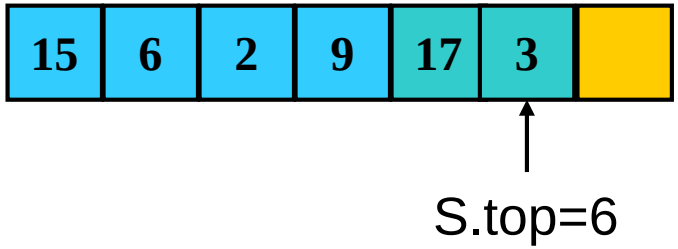
Empty(S):
1. if S.top=0 then
2. return true
3. else
4. return false

Stack

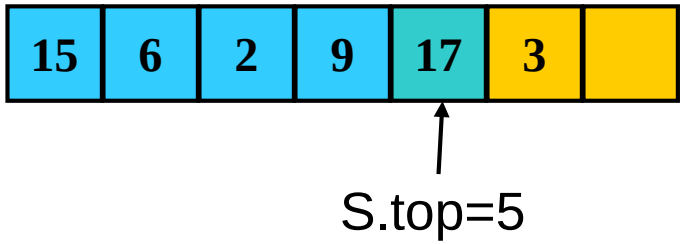
Starting point:



After Push(S,17), Push(S,3):



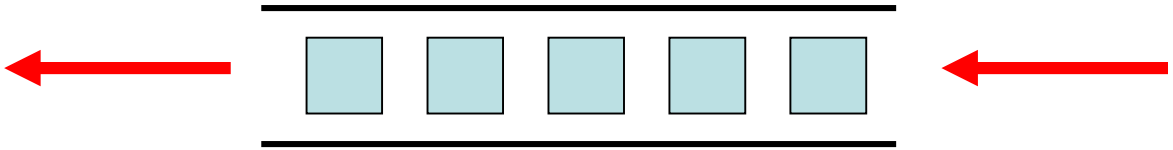
After Pop(S):



Queue

Queue operations:

- Enqueue(Q,x): insert element x into queue Q
- Dequeue(Q): remove oldest element from queue
- Empty(Q): checks if queue is empty

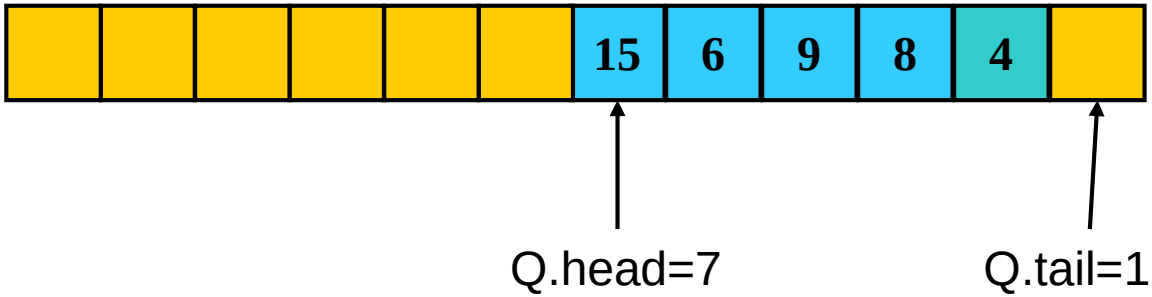


Simplest implementation:

- Array Q[1..N] (#elements in Q at most N)
- Q.head: starting point in Q[1..N]
- Q.tail: end point (next free position) in Q[1..N]
- Initially, Q.head=Q.tail=1

Queue

Example:



Implementation:

Empty(Q):

1. if Q.head=Q.tail then
2. return true
3. else
4. return false

Enqueue(Q,x):

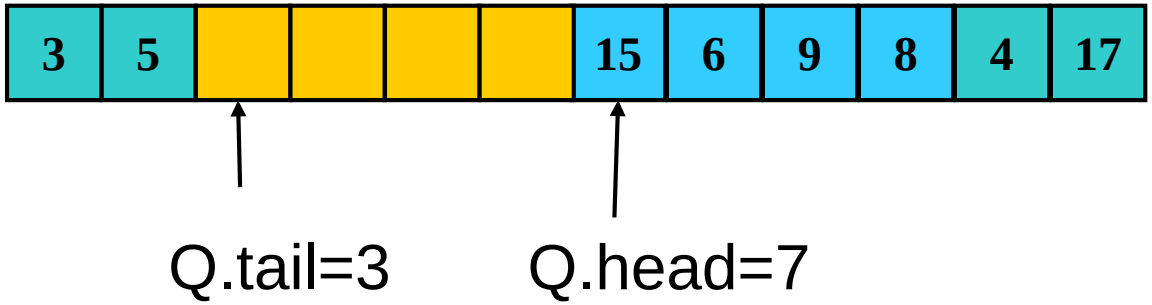
1. if Q.tail=Q.head-1 or (Q.head=1 and Q.tail=N) then error „overflow“
2. else
3. Q[Q.tail]:=x
4. if Q.tail=Q.length then
5. Q.tail:=1
6. else Q.tail:=Q.tail+1

Dequeue(Q):

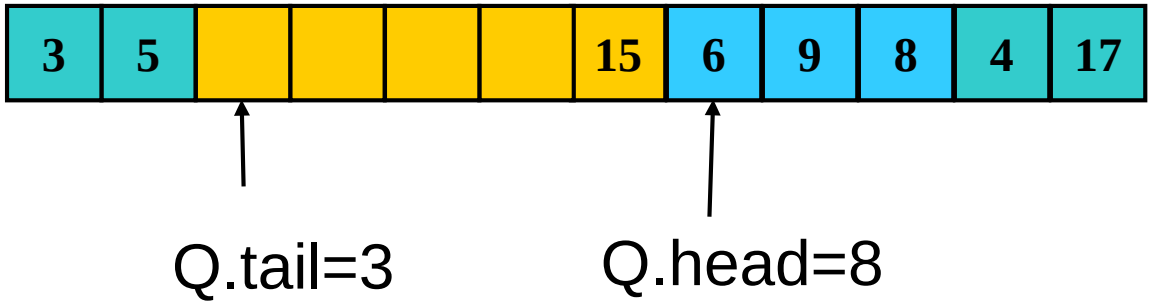
1. if Empty(Q) then error „underflow“
2. else
3. x:=Q[Q.head]
4. if Q.head=Q.length then
5. Q.head:=1
6. else Q.head:=Q.head+1
7. return x

Queue

Example:



After Dequeue:



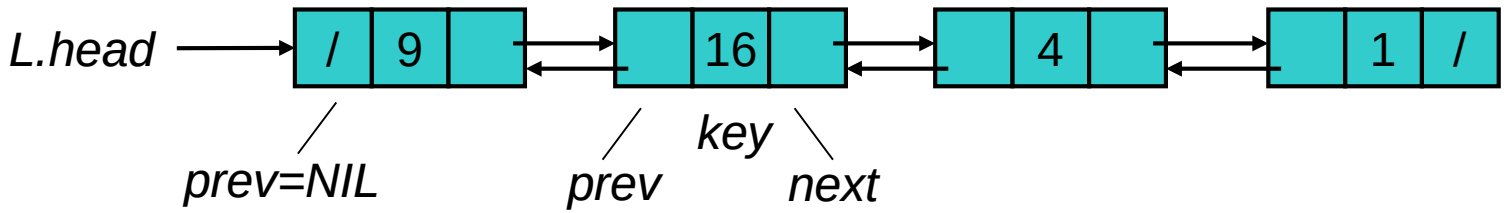
Linked Lists

List operations:

- List-Insert(L,x): insert element x into list
- List-Delete(L,x): remove element x from list
- List-Search(L,k): search for key k in list

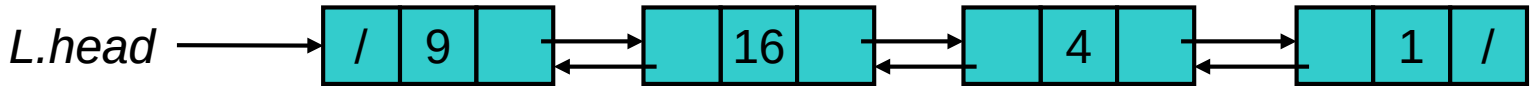
Implementation as doubly linked list:

- L.head: refers to first element of list L
- x: element of list, which contains x.prev, X.key, and x.next
- x.prev: refers to predecessor of x (NIL: no predecessor)
- x.next: refers to successor of x (NIL: no successor)
- x.key: stores key of x

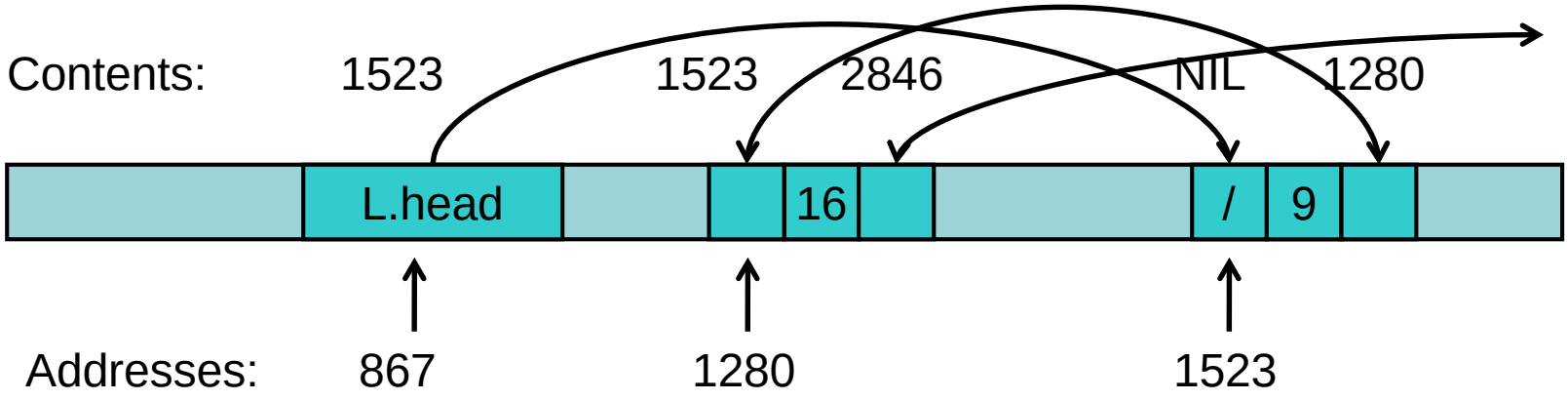


Linked Lists

Abstract view: linked list



Internal representation: linear addressable memory

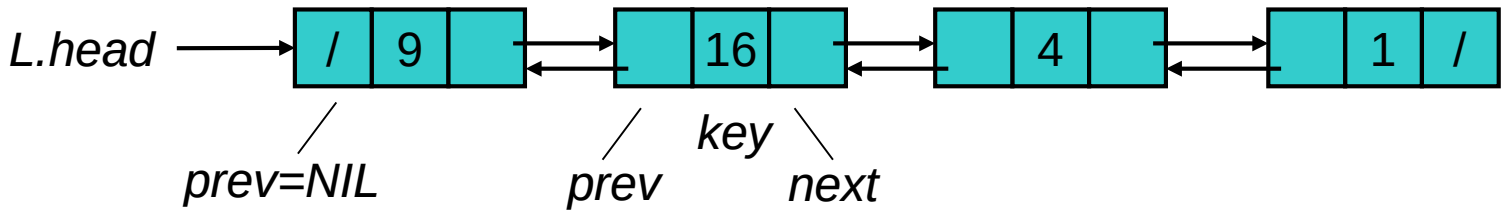


Linked Lists

List-Search(L,k):

1. x:=L.head
2. while x≠NIL and x.key≠k do
3. x:=x.next
4. return x

Example: List-Search(L,1)

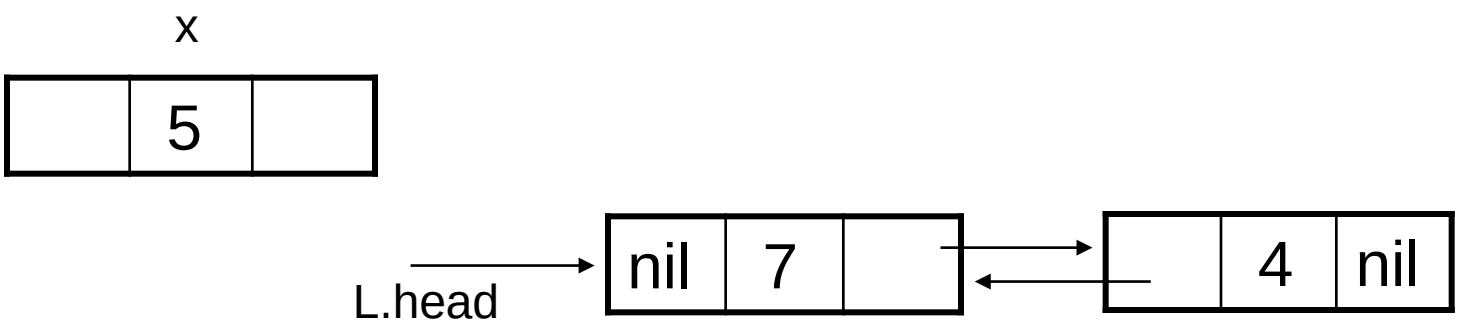


Linked Lists

List-Insert(L,x):

1. x.next:=L.head
2. if L.head≠NIL then
3. L.head.prev:=x
4. L.head:=x
5. x.prev:=NIL

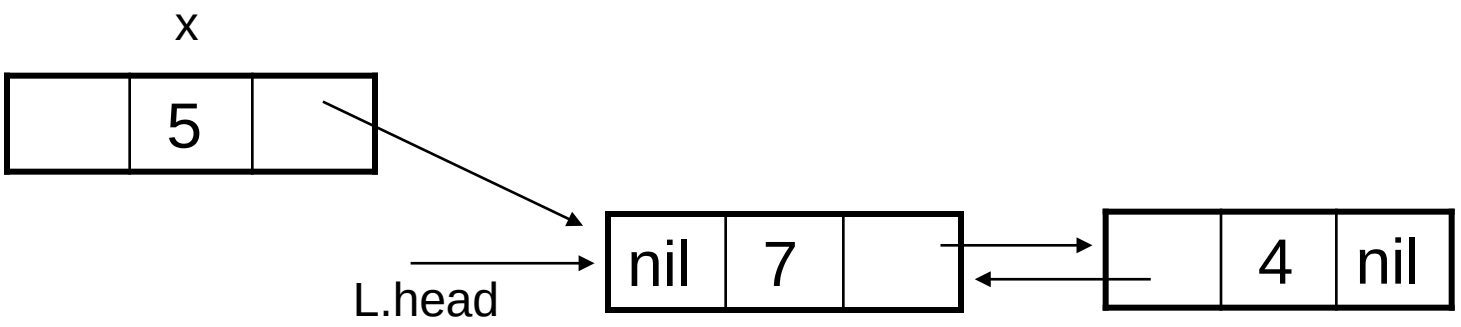
x: pointer that points to address of element that needs to be inserted



Linked Lists

List-Insert(L,x):

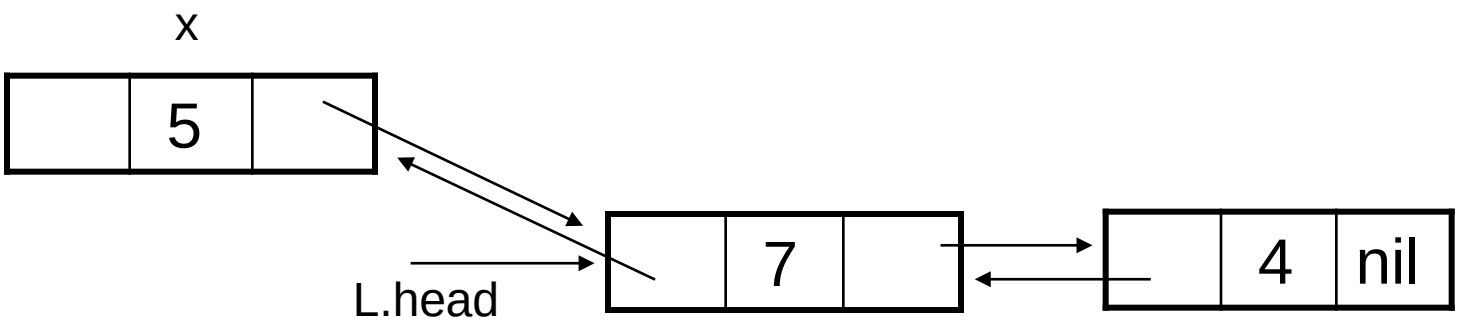
1. x.next:=L.head
2. if L.head≠NIL then
3. L.head.prev:=x
4. L.head:=x
5. x.prev:=NIL



Linked Lists

List-Insert(L,x):

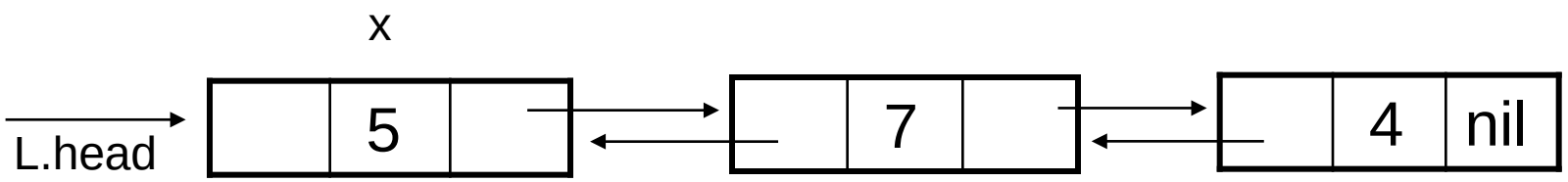
1. x.next:=L.head
2. if L.head≠NIL then
3. L.head.prev:=x
4. L.head:=x
5. x.prev:=NIL



Linked Lists

List-Insert(L,x):

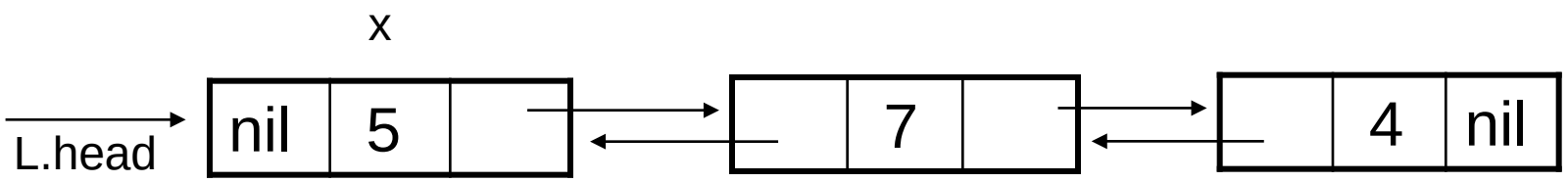
1. x.next:=L.head
2. if L.head≠NIL then
3. L.head.prev:=x
4. L.head:=x
5. x.prev:=NIL



Linked Lists

List-Insert(L,x):

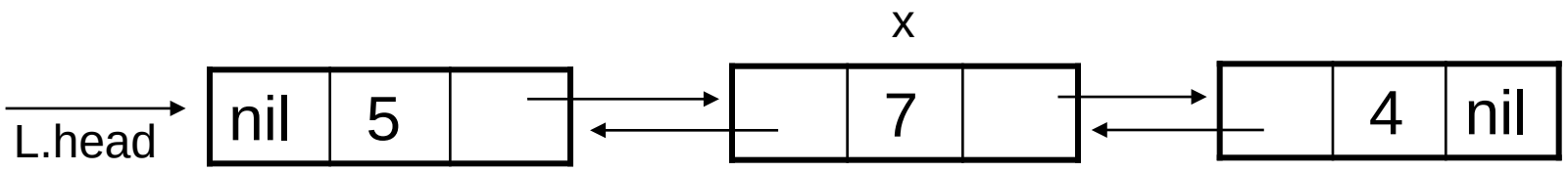
1. x.next:=L.head
2. if L.head≠NIL then
3. L.head.prev:=x
4. L.head:=x
5. x.prev:=NIL



Linked Lists

List-Delete(L,x):

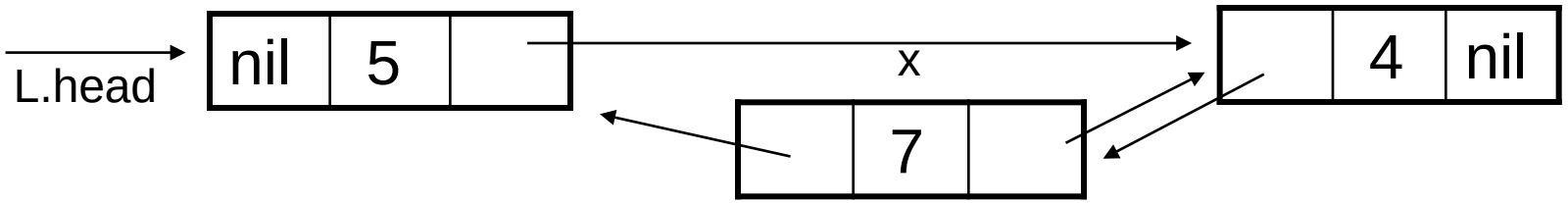
1. if x.prev≠NIL then
2. x.prev.next:=x.next
3. else L.head:=x.next
4. if x.next≠NIL then
5. x.next.prev:=x.prev



Linked Lists

List-Delete(L,x):

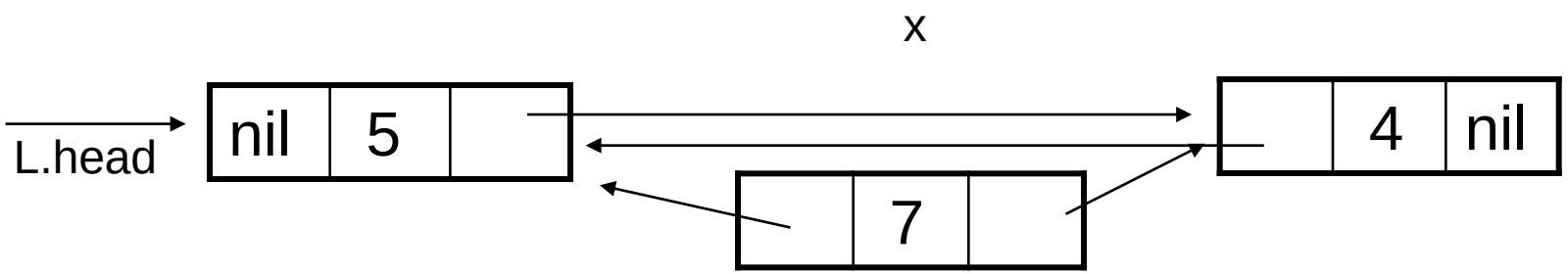
1. if x.prev≠NIL then
2. x.prev.next:=x.next
3. else L.head:=x.next
4. if x.next≠NIL then
5. x.next.prev:=x.prev



Linked Lists

List-Delete(L,x):

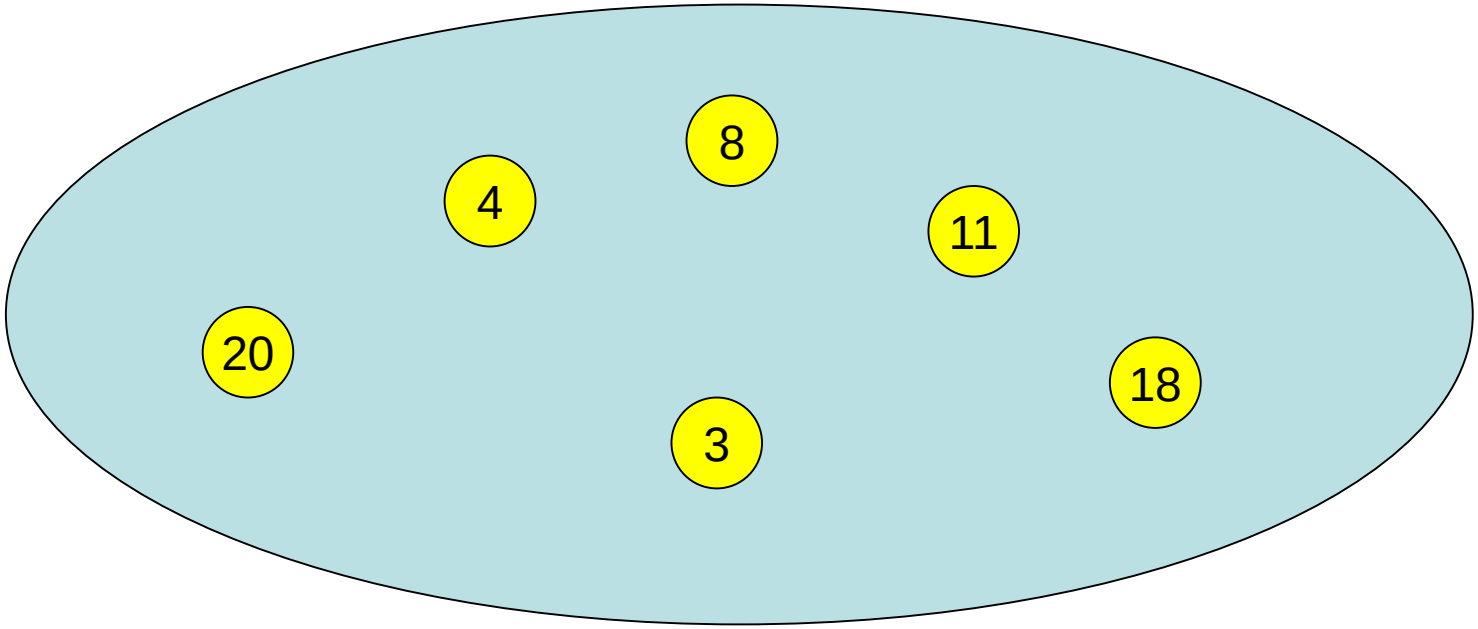
1. if x.prev≠NIL then
2. x.prev.next:=x.next
3. else L.head:=x.next
4. if x.next≠NIL then
5. x.next.prev:=x.prev



Overview

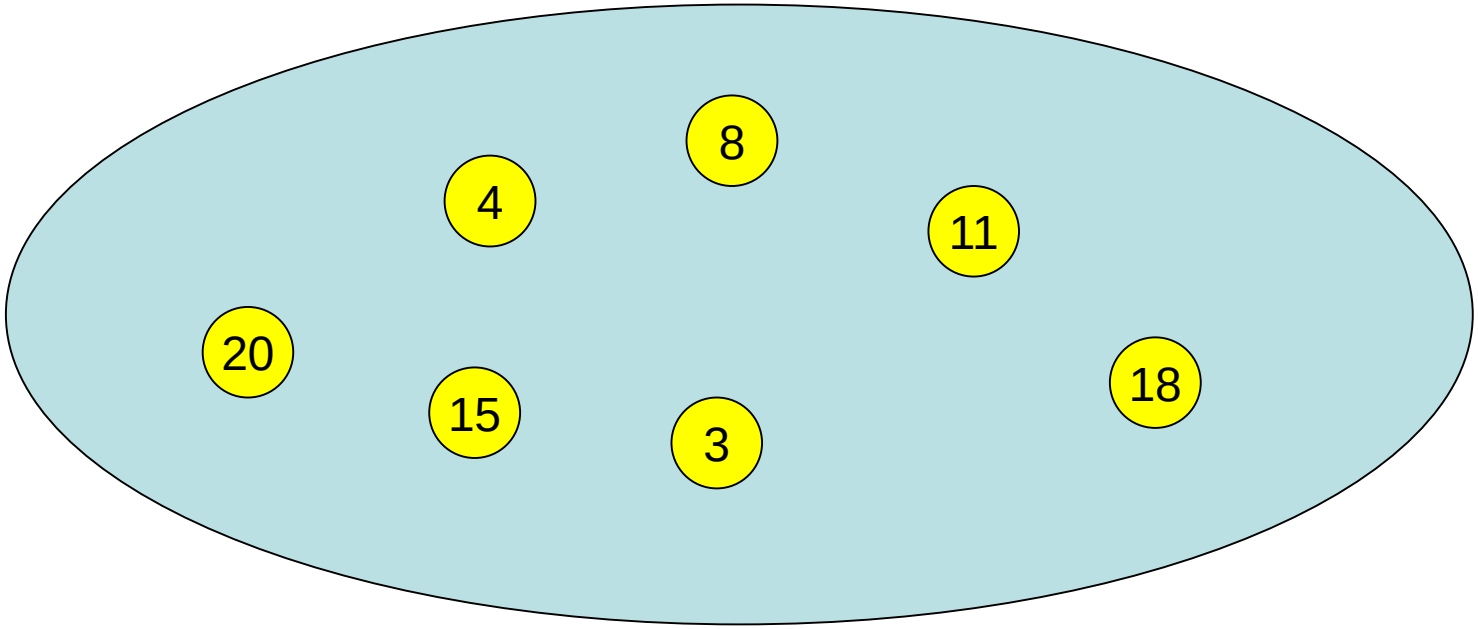
- Basic data structures
- Search structures (successor searching)
- Dictionaries (exact searching)

Search Structure



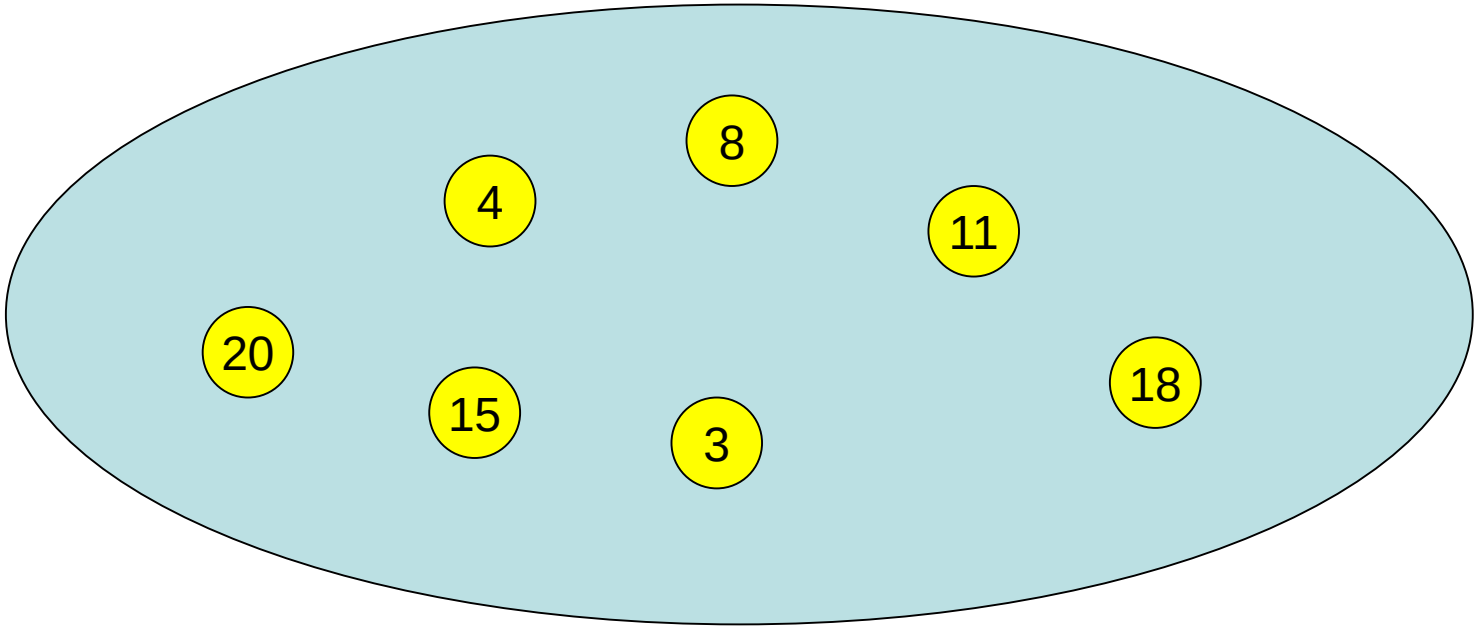
Search Structure

insert(15)



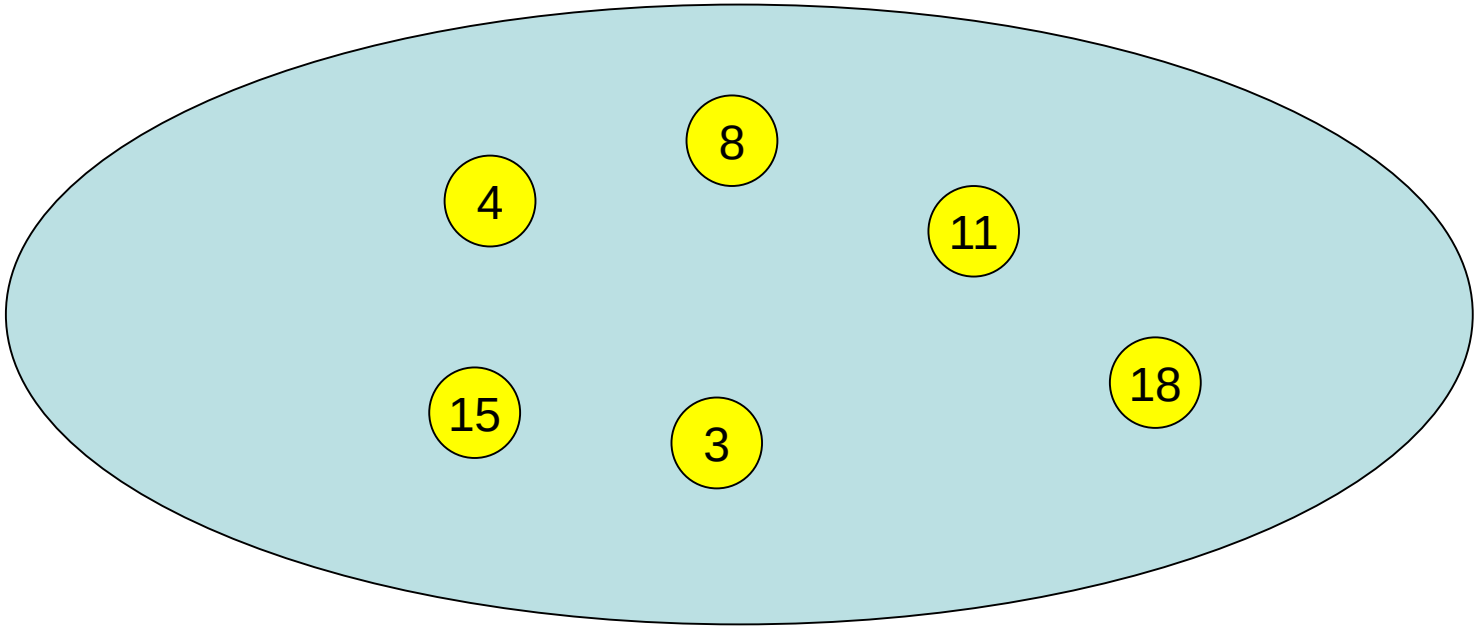
Search Structure

delete(20)



Search Structure

search(7) gives 8 (closest successor)



Search Structure

S: set of elements

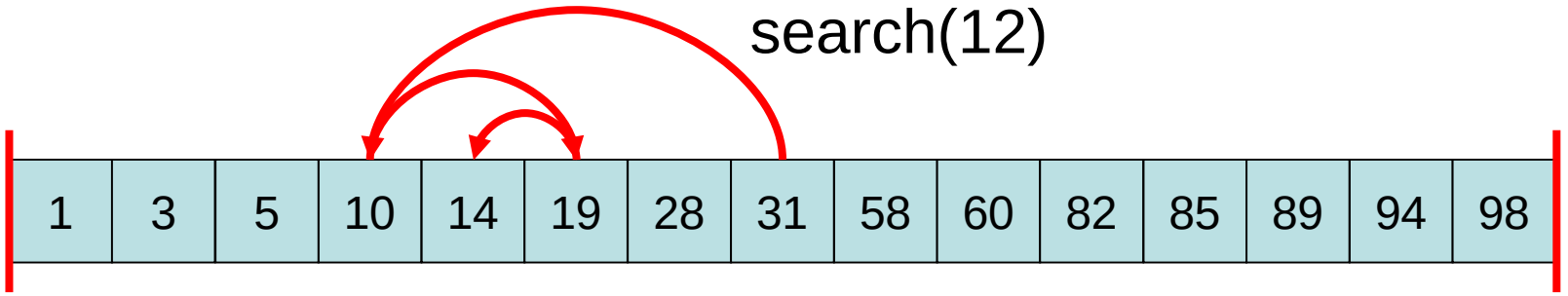
Every element e identified by key(e).

Operations:

- S.insert(e: Element): $S:=S\cup\{e\}$
- S.delete(k: Key): $S:=S\setminus\{e\}$, where e is the element with key(e)=k
- S.search(k: Key): outputs $e\in S$ with minimal key(e) so that $key(e)\geq k$

Static Search Structure

1. Store elements in sorted array.



search: via binary search (in O(log n) time)

Binary Search

Input: number x and sorted array A[1],…,A[n]

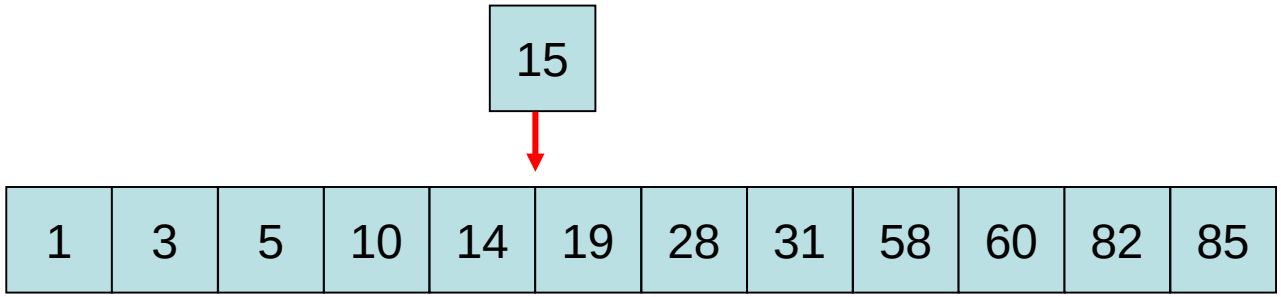
Algorithm BinarySearch:

```
l:=1; r:=n
while l < r do
    m:=(r+l) div 2
    if A[m] = x then return m
    if A[m] < x then l:=m+1
                else r:=m
return l
```

Dynamic Search Structure

insert und delete Operations:

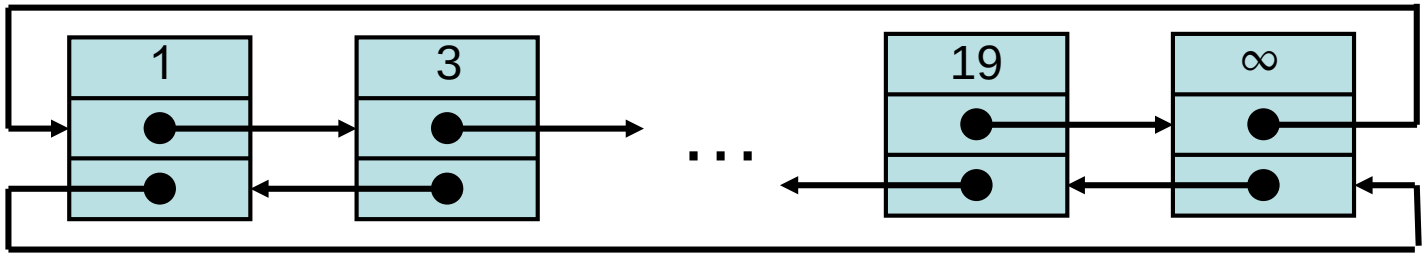
Sorted array difficult to update!



Worst case: $\Theta(n)$ time

Search Structure

2. Sorted List (with an ∞-Element)

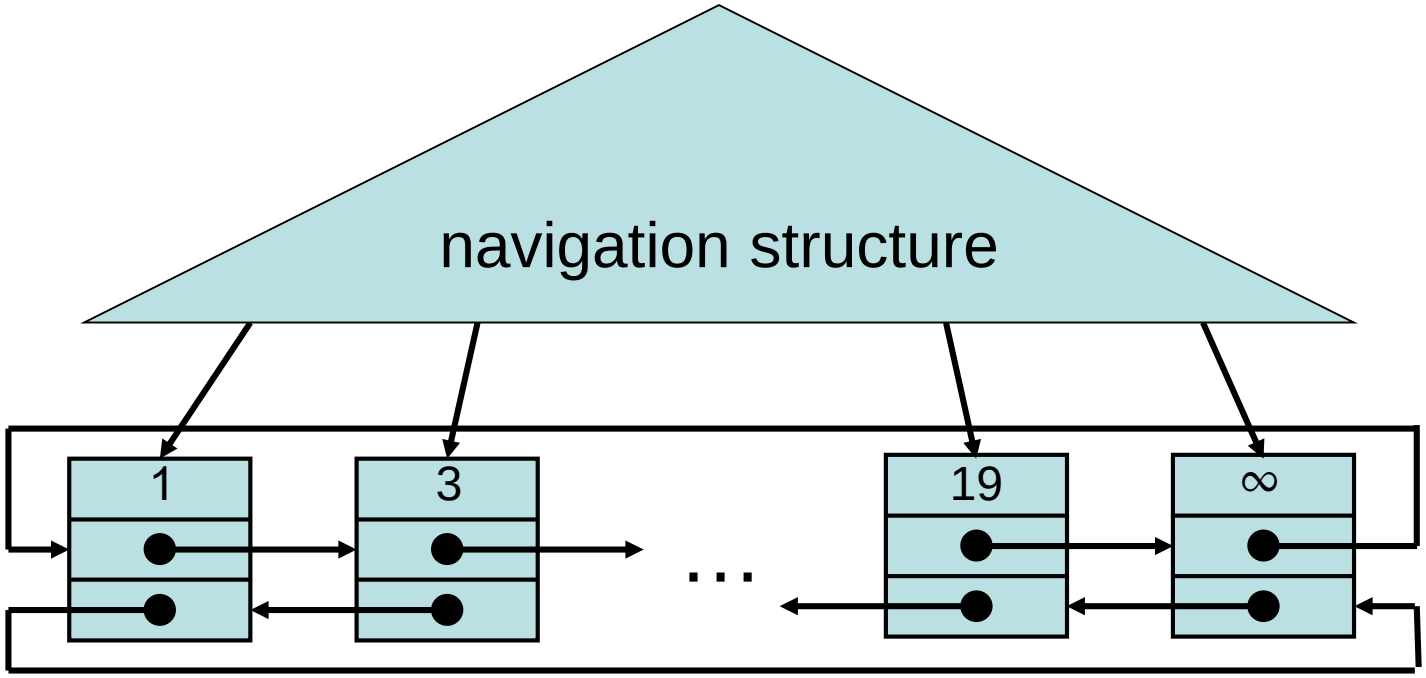


Problem: insert, delete and search take $\Theta(n)$ time in the worst case

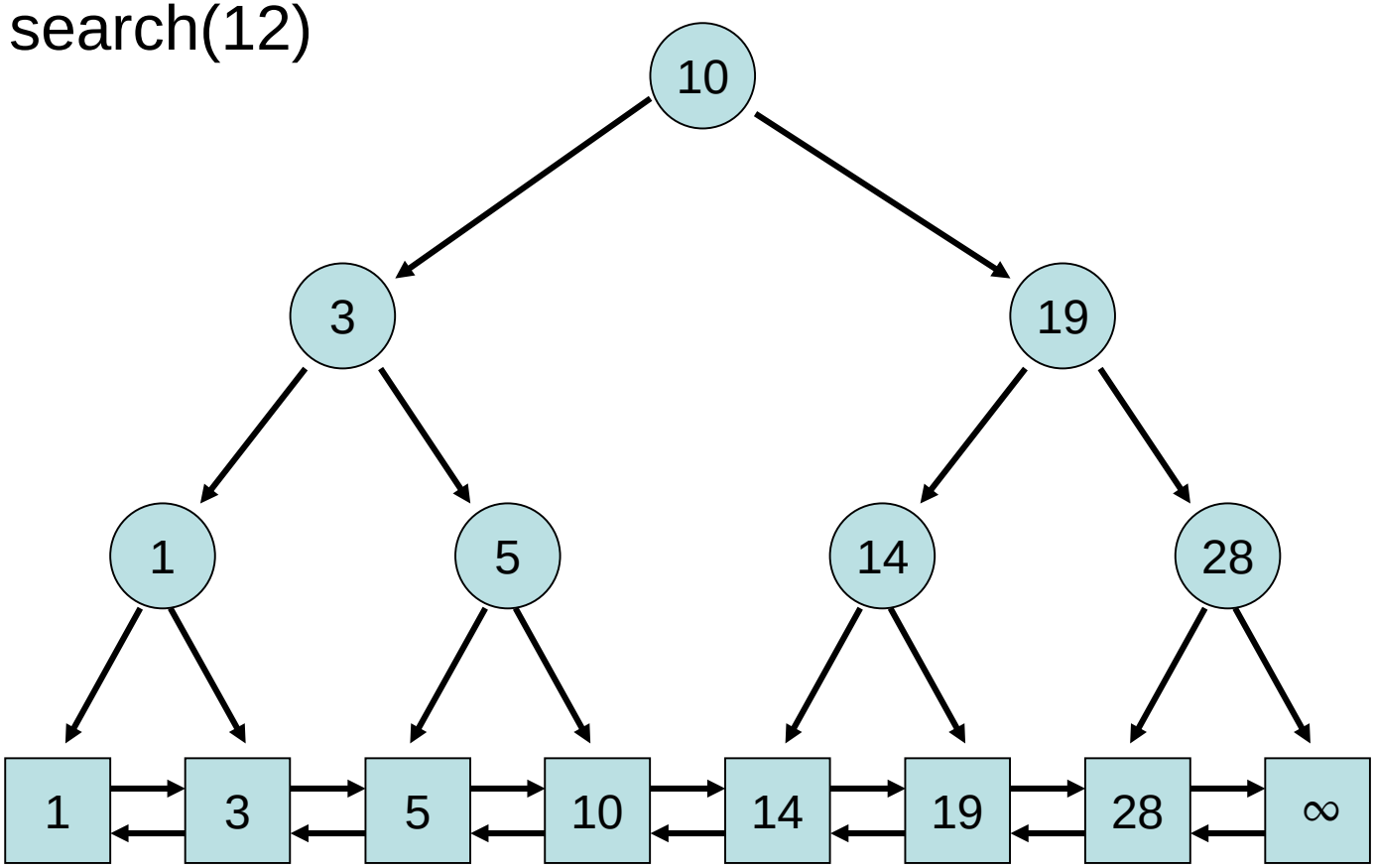
Observation: If search could be implemented efficiently, then also all other operations

Search Structure

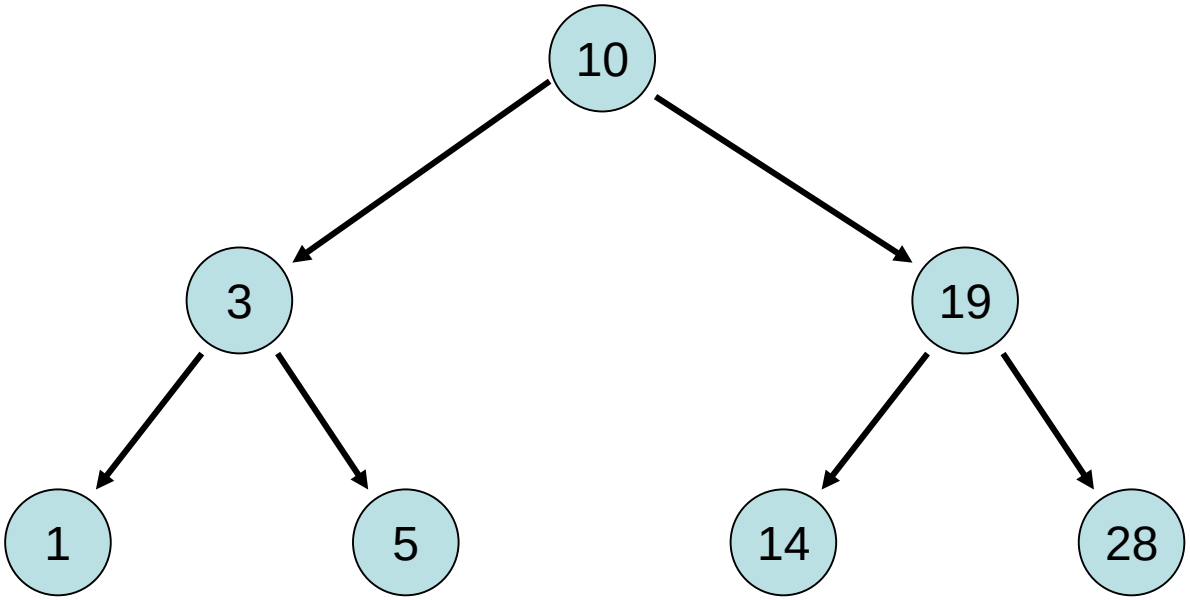
Idee: add navigation structure that allows search to run efficiently



Binary Search Tree



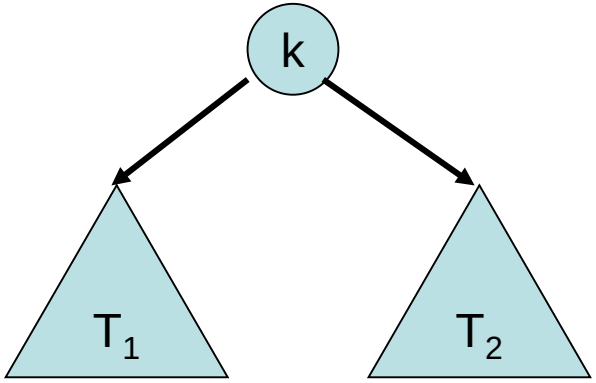
Binary Search Tree



Usual setup: without list (full elements and not just keys stored in tree). Updates easier to understand if list is added.

Binary Search Tree

Search tree invariant:



For all keys k' in T_1 and k'' in T_2: $k' \leq k < k''$

Binary Search Tree

Formally: for every tree node v let

- key(v) be the key stored at v
- d(v) the number of children of v

- Search tree invariant: (as above)

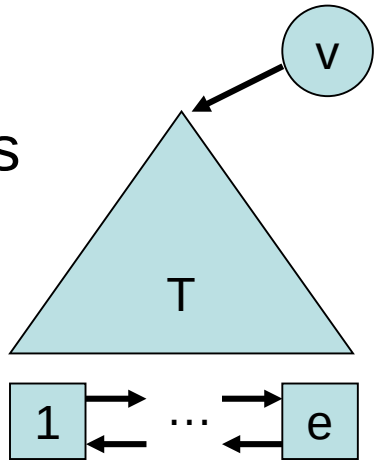
- Degree invariant:
  All tree nodes have exactly two children
  (as long as the number of elements in the list is >1)

- Key invariant:
  For every element e in the list there is exactly one tree node v with key(v)=key(e).

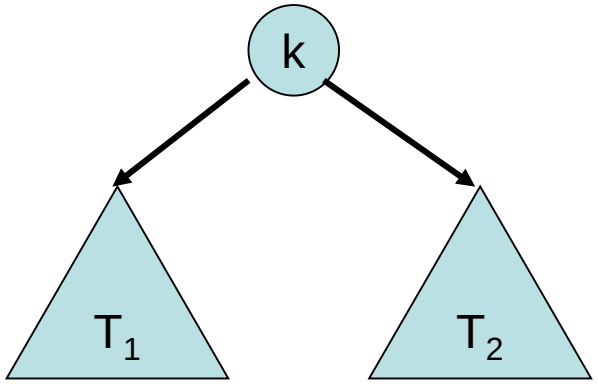
Binary Search Tree

- Search tree invariant: (as before)
- Degree invariant:
  All tree nodes have exactly two children
  (as long as the number of elements is >1)
- Key invariant:
  For every element e in the list there is exactly one tree node v with key(v)=key(e).

From the search tree and key invariants it follows that for every left subtree T of a node v, the rightmost list element e under T satisfies key(v)=key(e).



search(x) Operation

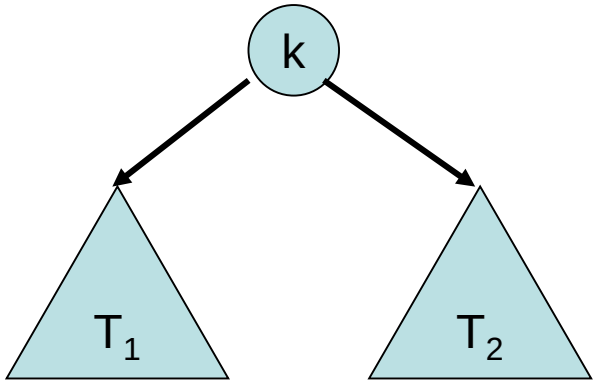


For all keys k' in T_1 and k'' in T_2: $k' \leq k < k''$

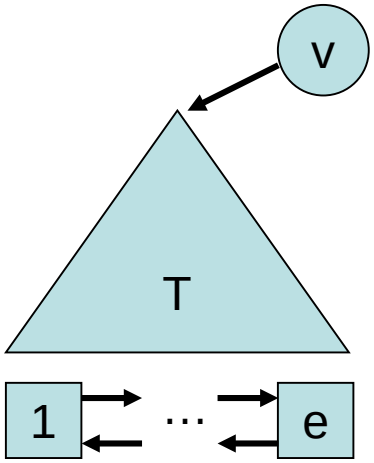
Search strategy:

- Start at the root, v, of the search tree
- while v is a tree node:
  - if $x \leq key(v)$ then let v be the left child of v,
    otherwise let v be the right child of v
- Output (list node) v

search(x) Operation



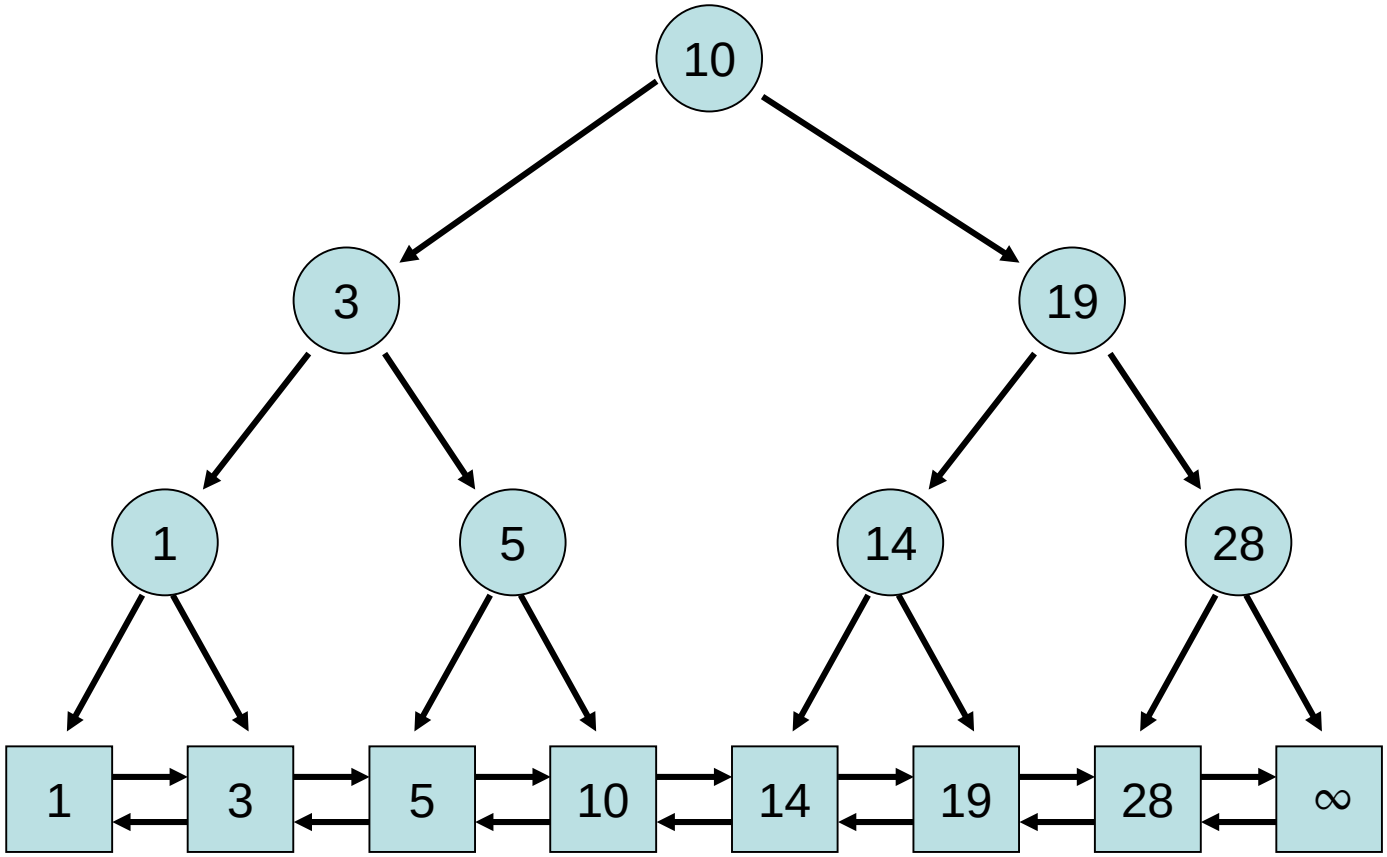
For all keys k' in T_1 and k'' in T_2: $k' \leq k < k''$



Correctness of search strategy:

- For every left subtree T of a node v, the rightmost list element e under T satisfies key(v)=key(e).
- So whenever search(x) enters T, there is an element e in the list below T with key(e)≥x.

Search(9)



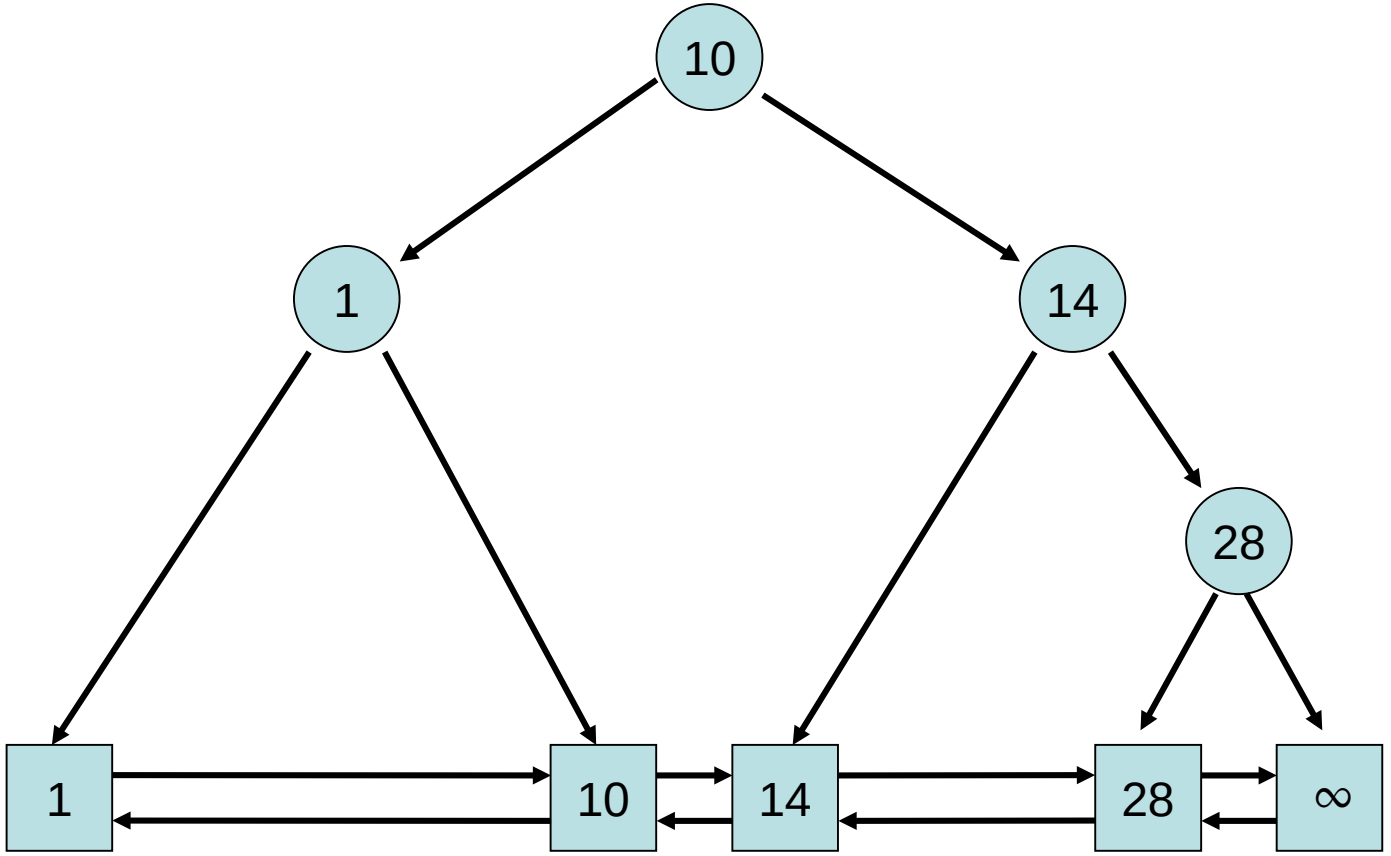
Insert and Delete Operations

Strategy:

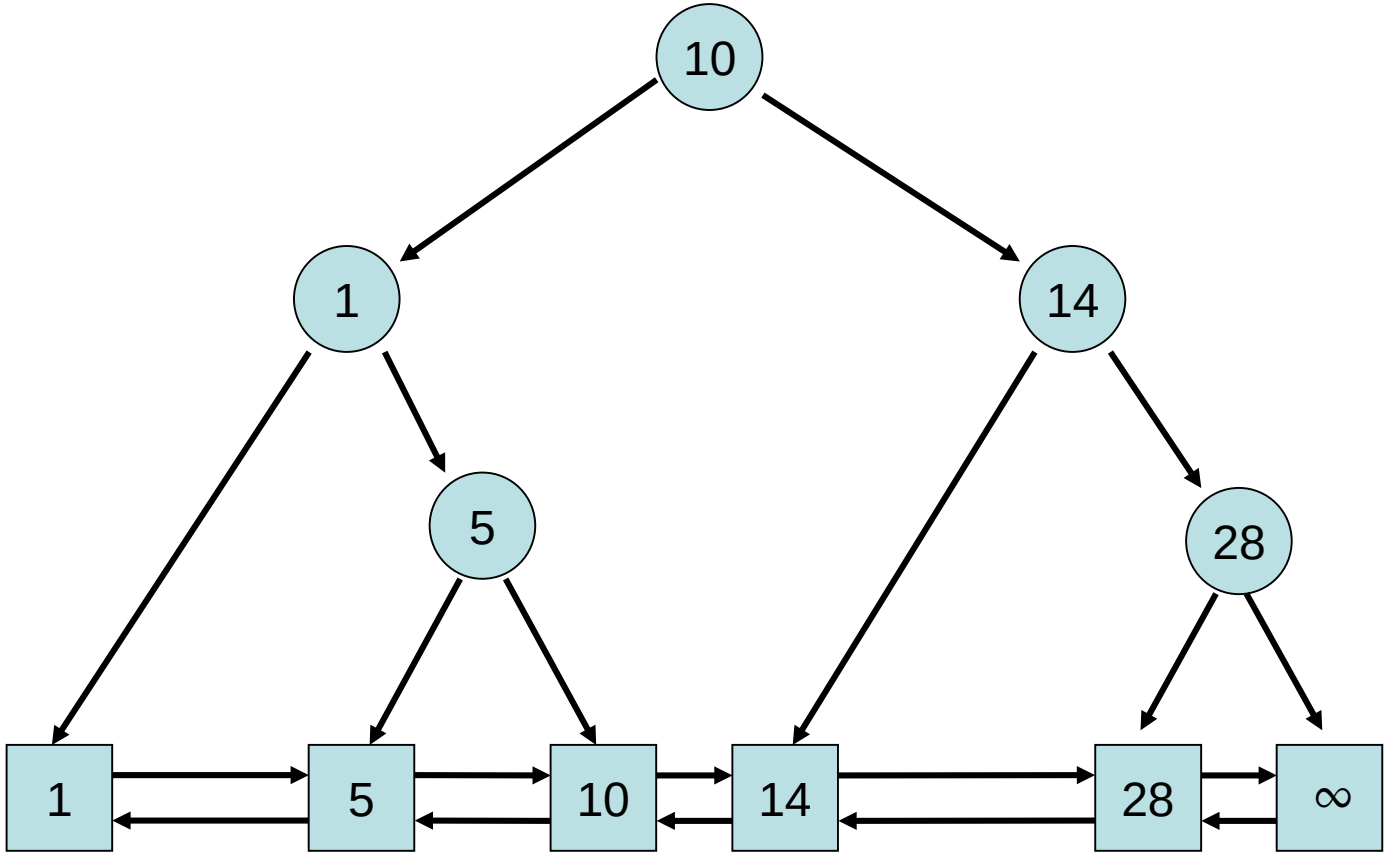
- insert(e):
  First, execute search(key(e)) to obtain a list element e´. If key(e)=key(e´), replace e´ by e, otherwise insert e between e´ and its predecessor in the list and add a new search tree leaf for e and e´ with key key(e) whose parent is the old parent of e´.

- delete(k):
  First, execute search(k) to obtain a list element e. If key(e)=k, then delete e from the list and the parent v of e from the search tree, and set in the tree node w with key(w)=k: key(w):=key(v). The new parent of the remaining child of v is set to the parent of v. (Thus, a delete can be seen as a reversal of an insert operation.)

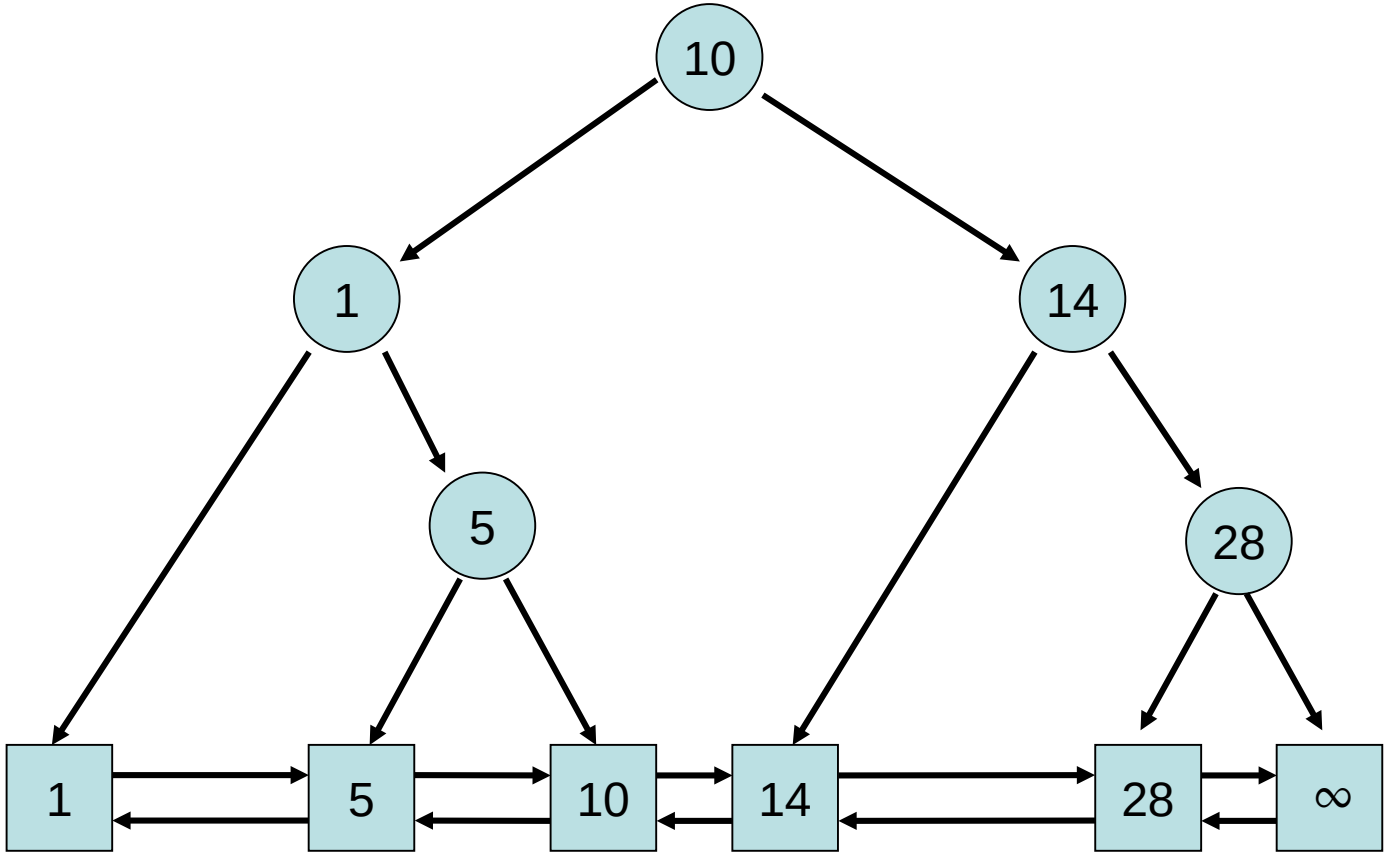
Insert(5)



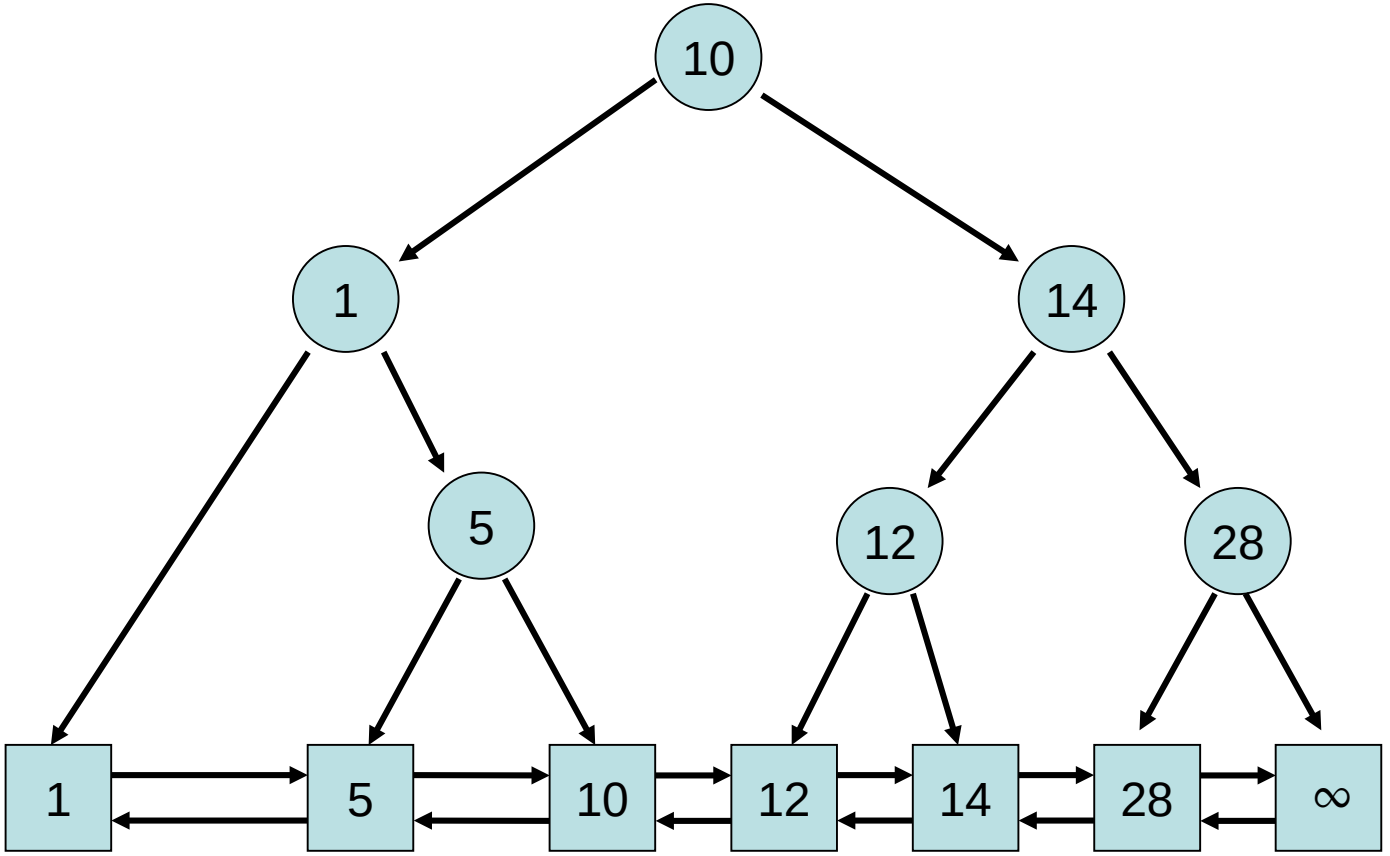
Insert(5)



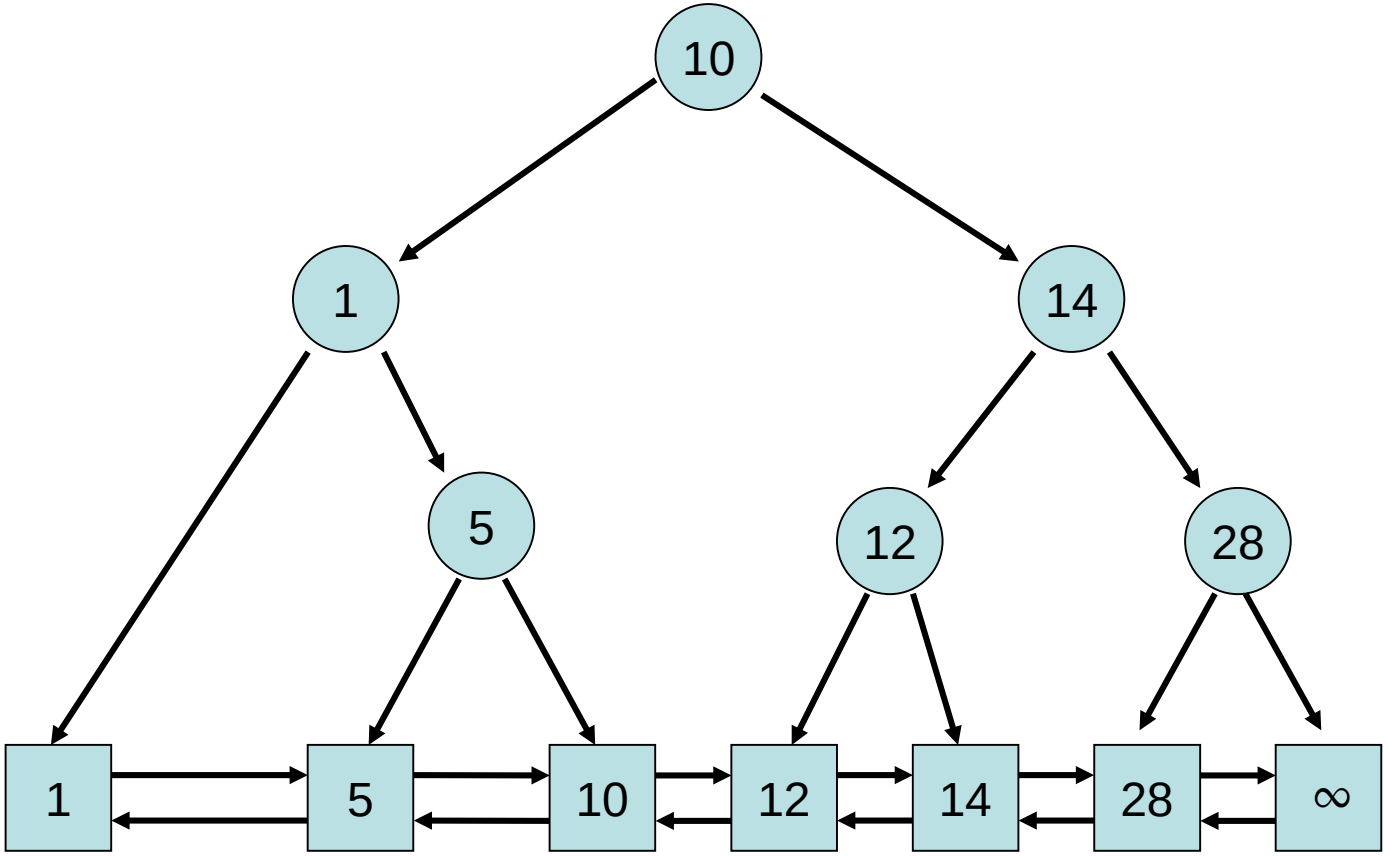
Insert(12)



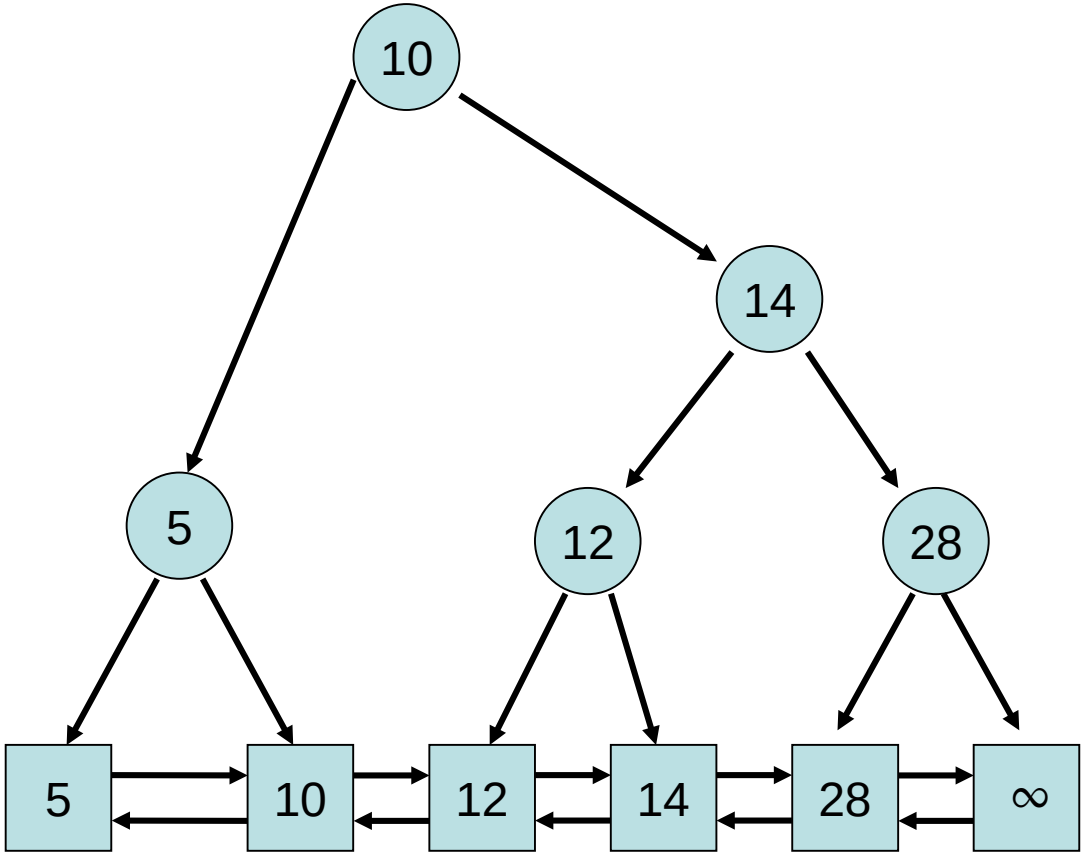
Insert(12)



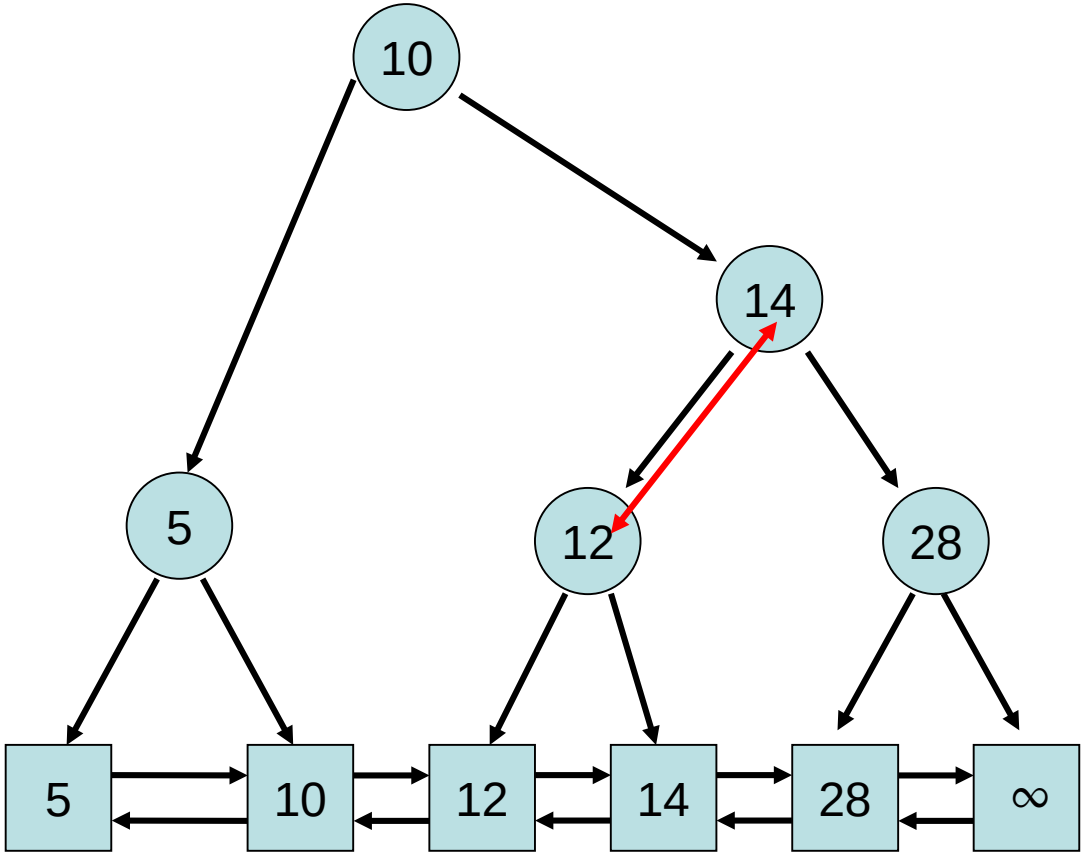
Delete(1)



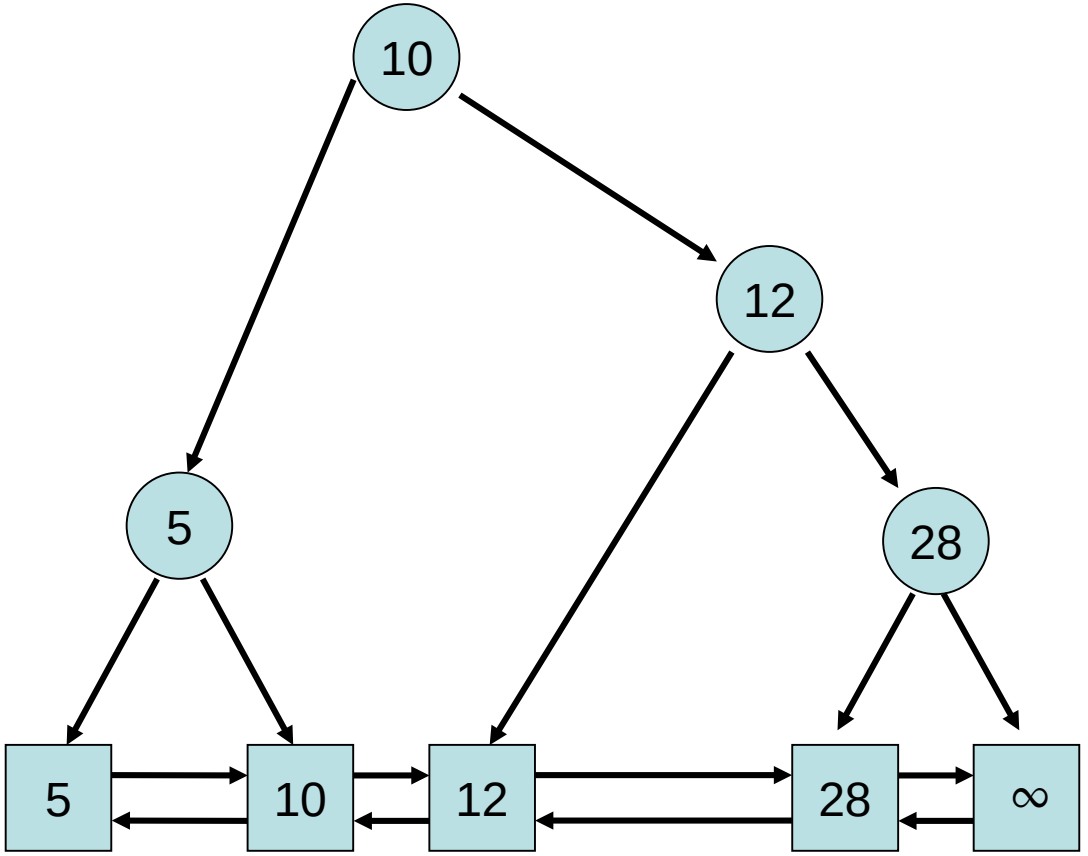
Delete(1)



Delete(14)



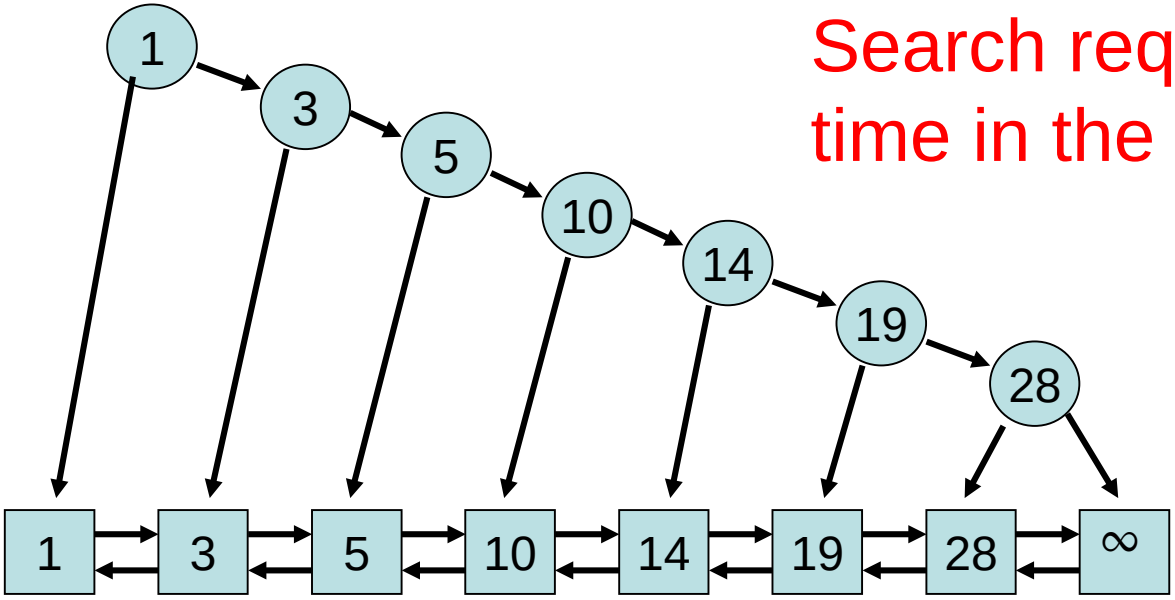
Delete(14)



Binary Search Tree

Problem: binary tree can degenerate!

Example: numbers are inserted in sorted order

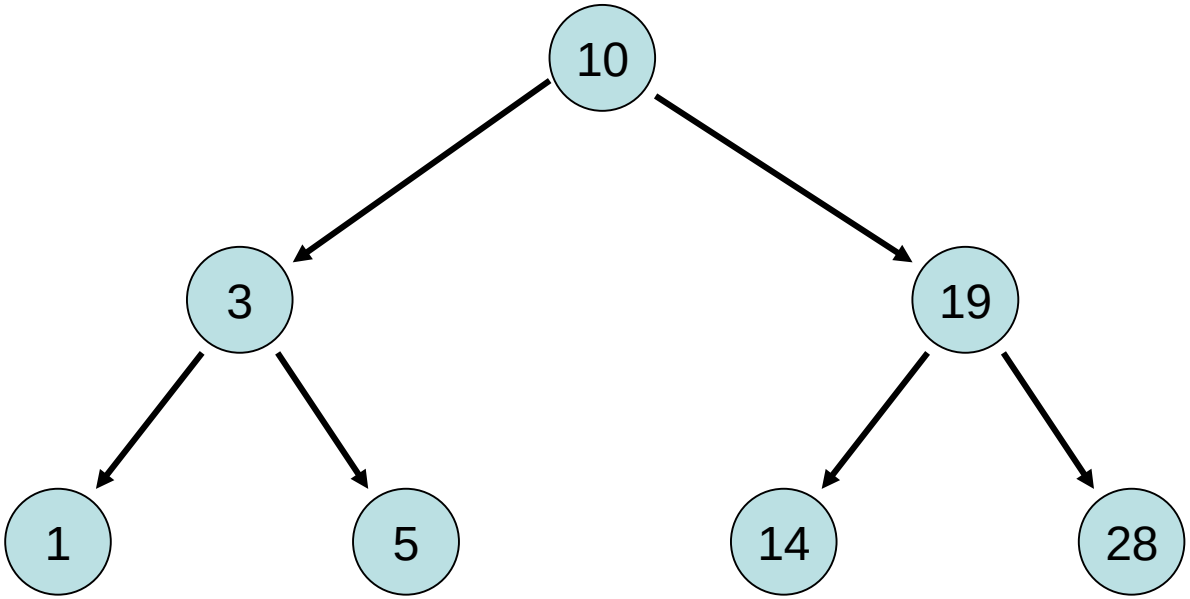


Search requires $\Theta(n)$ time in the worst case

Binary Search Tree

Lowest possible depth for n elements: log n.

Goal: maintain a depth of O(log n). Such a search tree is called balanced.



Search Trees

Problem: binary tree can degenerate!

Trees that stay balanced:

- (a,b)-tree (presented here)
- red-black tree
- AVL-tree
- …

(a,b)-Trees

Problem: how to maintain a balanced search tree

Idea:

- All nodes v (except for the root) have degree $d(v)$ with $a \leq d(v) \leq b$, where $a \geq 2$ and $b \geq 2a-1$ (otherwise this cannot be enforced)
- All leaves have the same depth (which implies that the tree stays balanced)

$\rightarrow$ (a,b)-trees, which are a general form of B-trees (see Chapter 18).

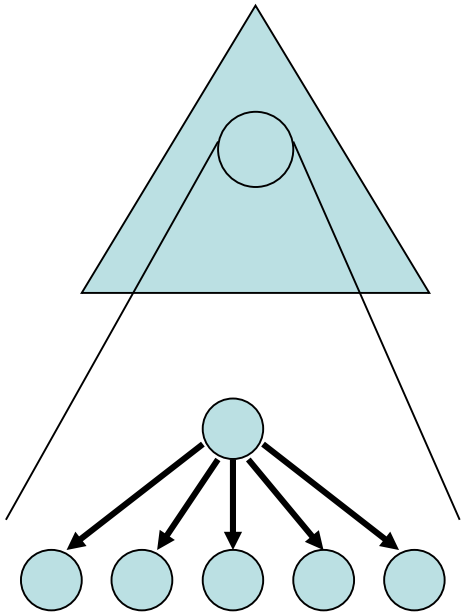
(a,b)-Trees

Formally: for a tree node v let

- $d(v)$ be the number of children of v
- $t(v)$ be the depth of v (root has depth 0)

- Form Invariant:
  For all leaves v,w: $t(v)=t(w)$

- Degree Invariant:
  For all inner nodes v
  except for root: $d(v) \in [a,b]$,
  for root r: $d(r) \in [2,b]$
  (as long as #elements >1)



(a,b)-Trees

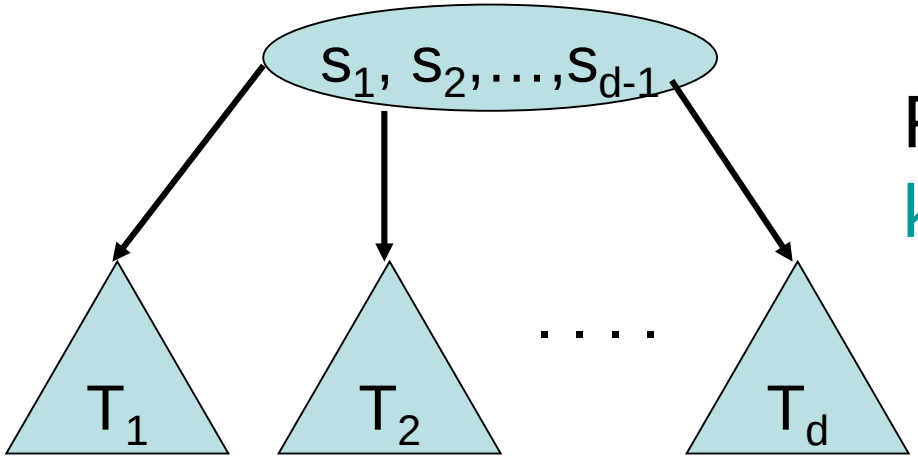
Lemma 3.10: An (a,b)-tree with n elements has depth at most $1+\lfloor \log_a (n+1)/2 \rfloor$

Proof:

- The root has degree ≥ 2 and every other inner node has degree $\geq a$.
- At depth t there are at least $2a^{t-1}$ nodes
- $n+1 \geq 2a^{t-1} \Leftrightarrow t \leq 1+\lfloor \log_a (n+1)/2 \rfloor$

(a,b)-Trees

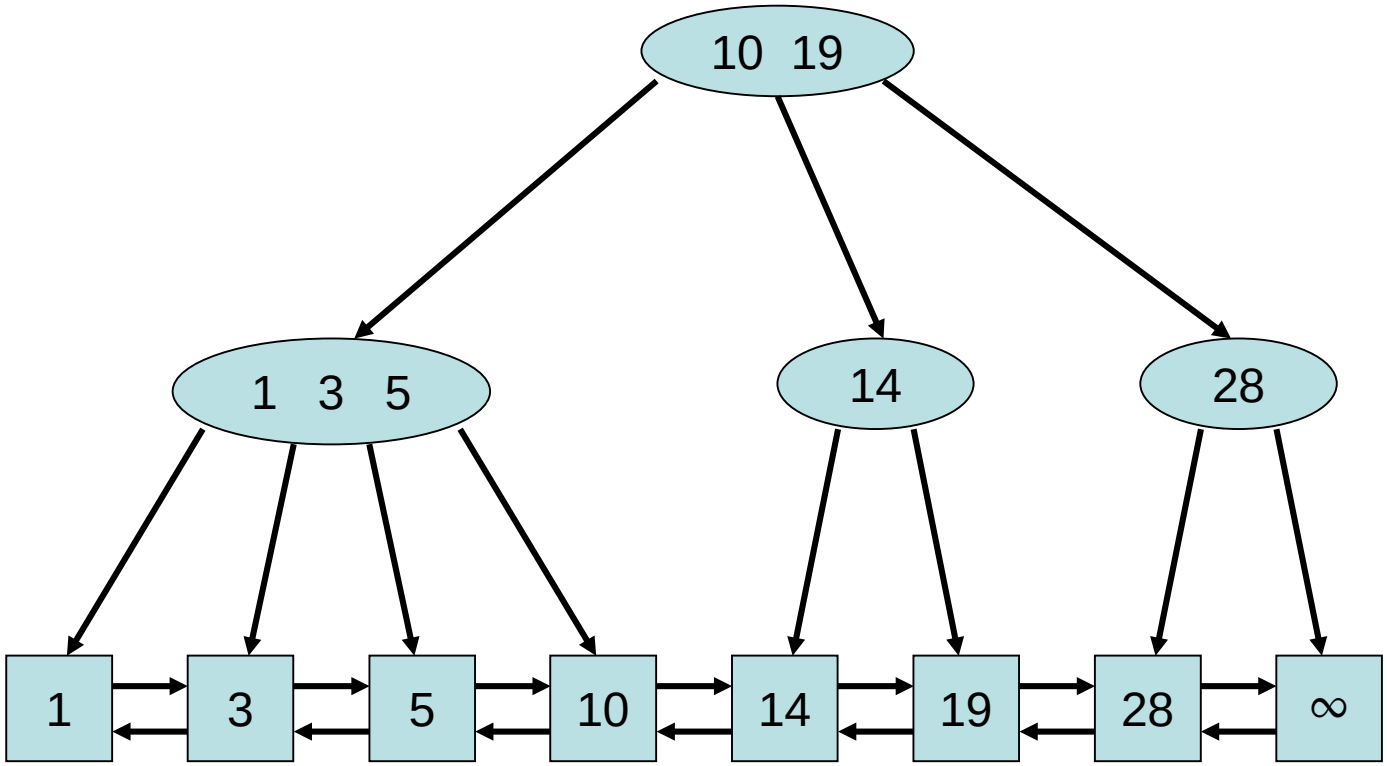
(a,b)-Tree-Rule:



For all keys k in T_i and k' in T_{i+1}: $k \leq s_i < k'$

Then search operation easy to implement.

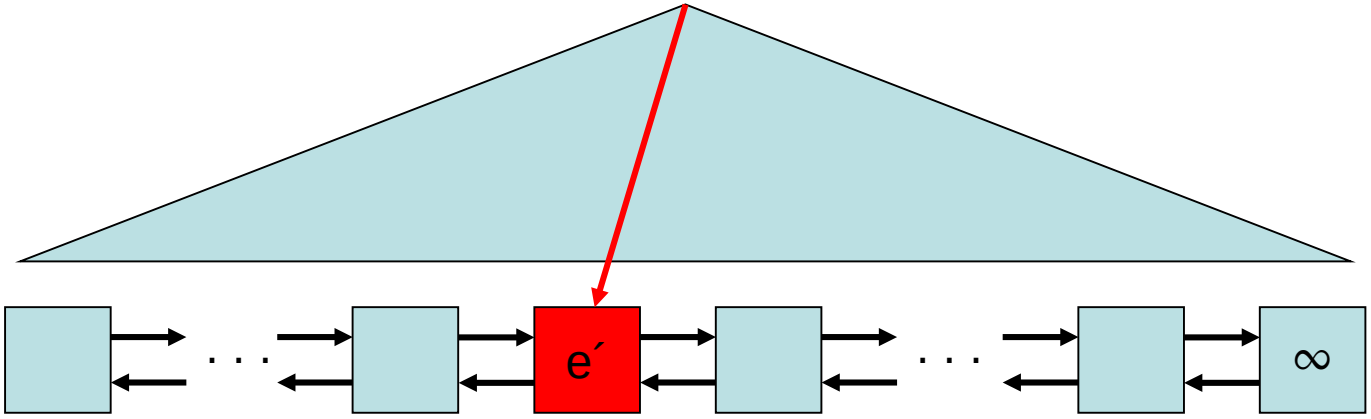
Search(9)



Insert(e) Operation

Strategy:

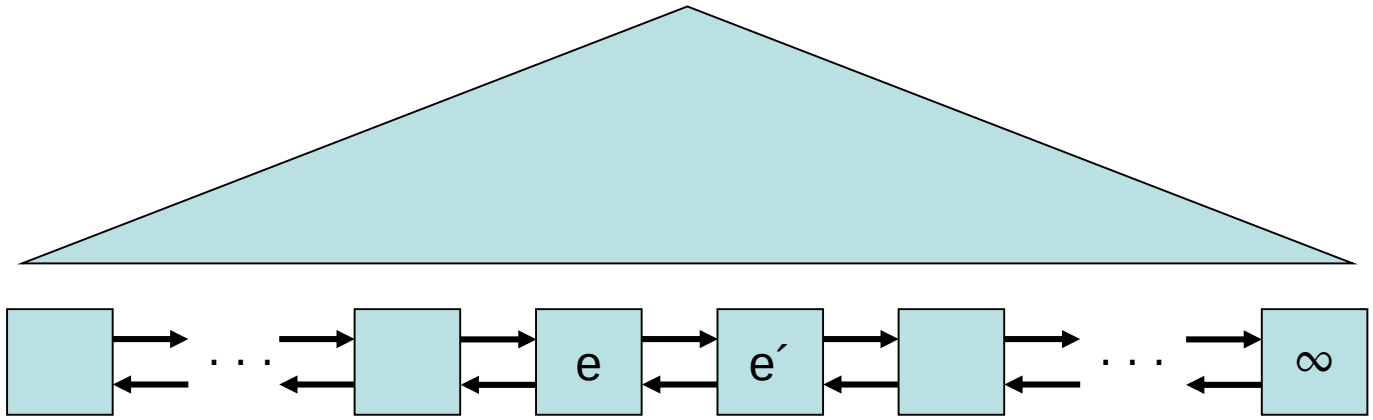
- First search(key(e)) until some e´ found in the list. If key(e´)>key(e), insert e in front of e´, otherwise replace e´ by e.



Insert(e) Operation

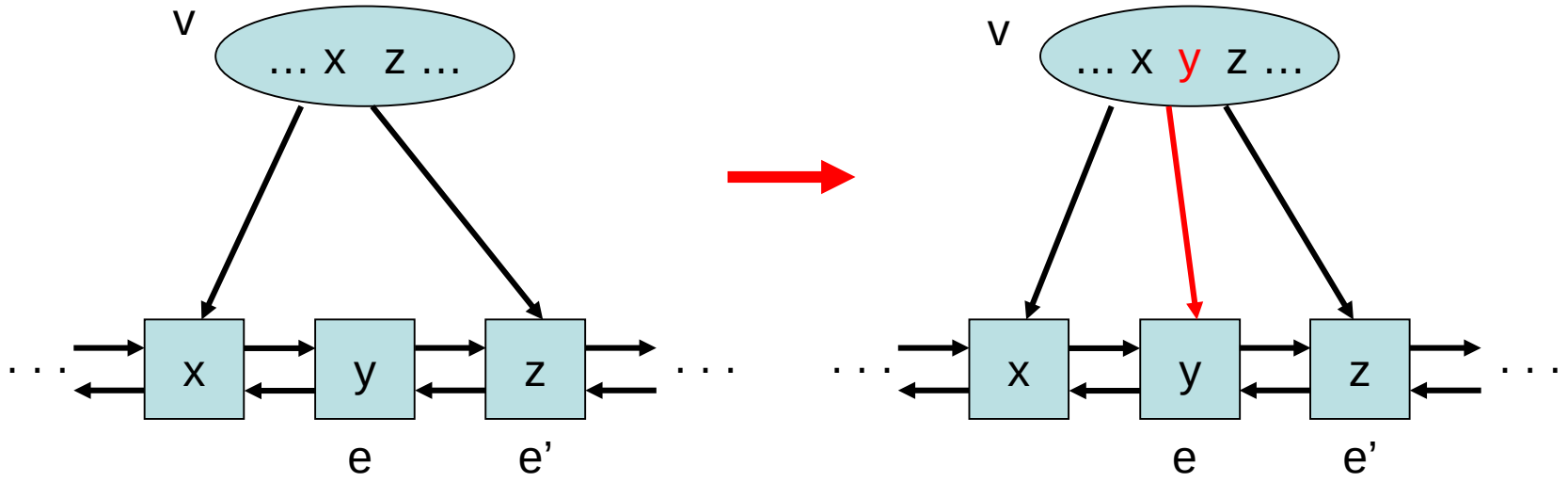
Strategy:

- First search(key(e)) until some e´ found in the list. If key(e´)>key(e), insert e in front of e´, otherwise replace e´ by e.



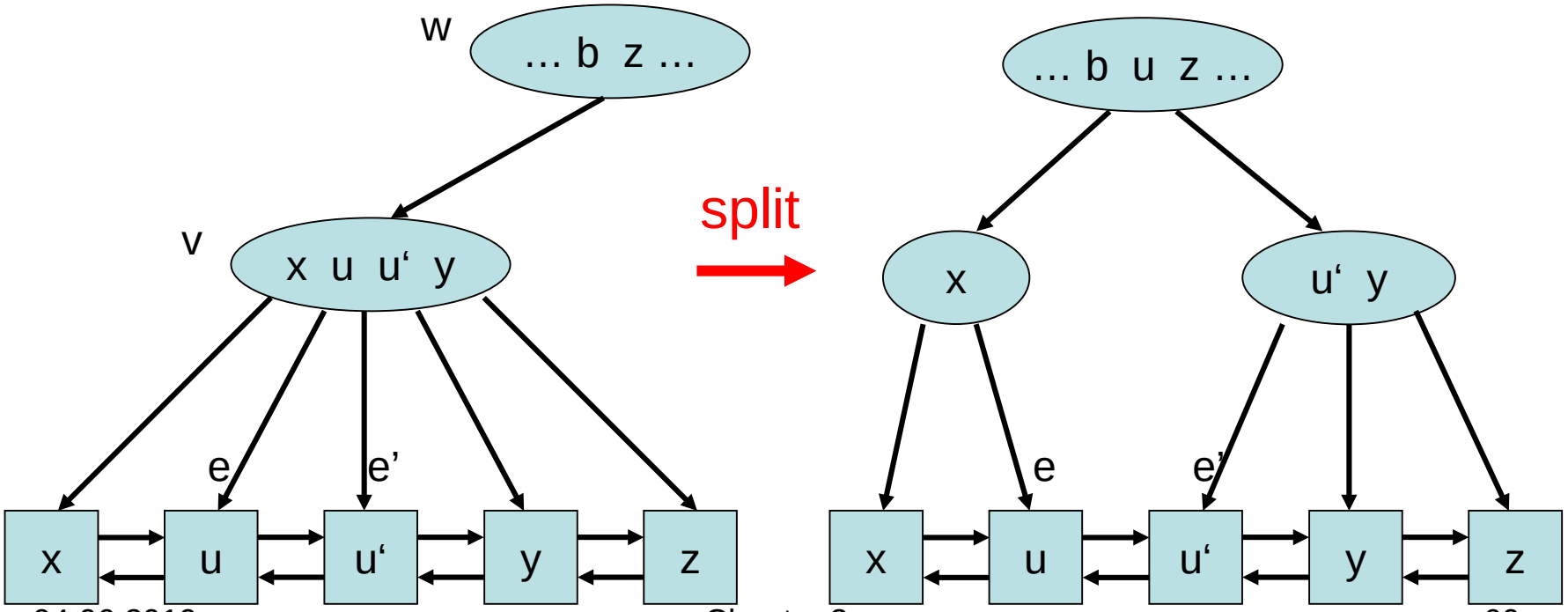
Insert(e) Operation

- Add key(e) and pointer to e in tree node v above e´. If we still have d(v)∈[a,b] afterwards, then we are done.



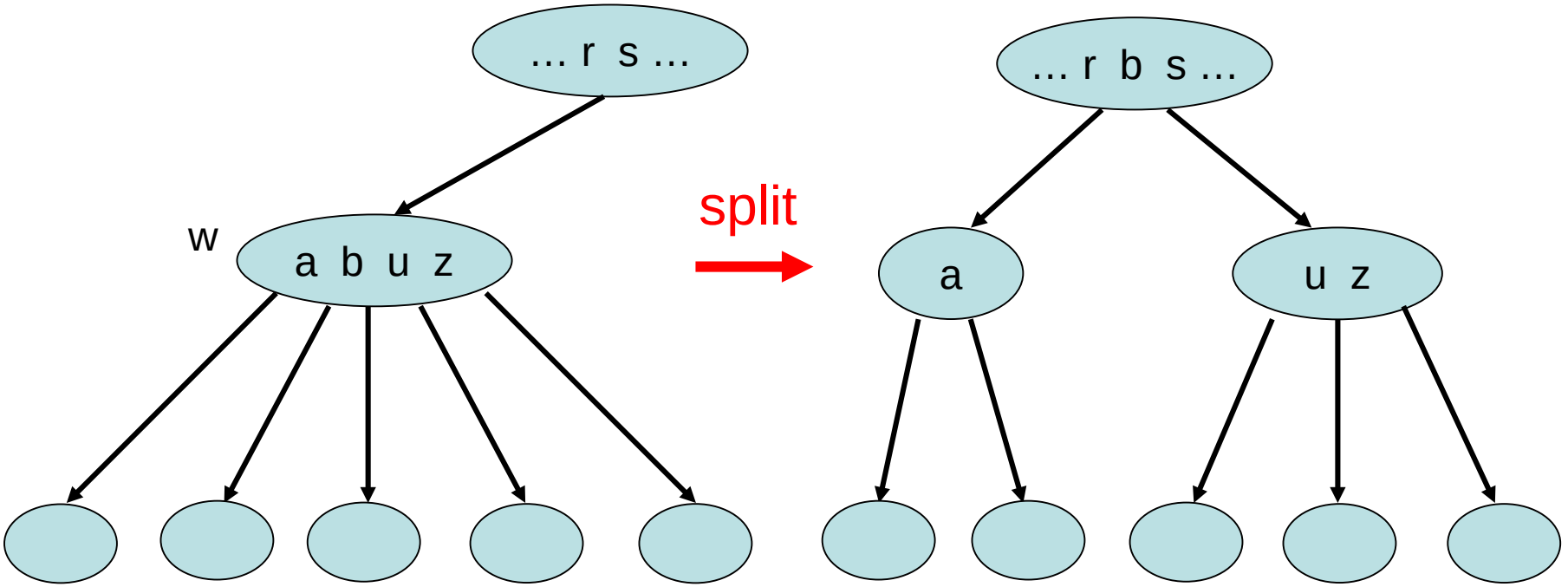
Insert(e) Operation

- If d(v)>b, then cut v into two nodes. (Example: a=2, b=4)



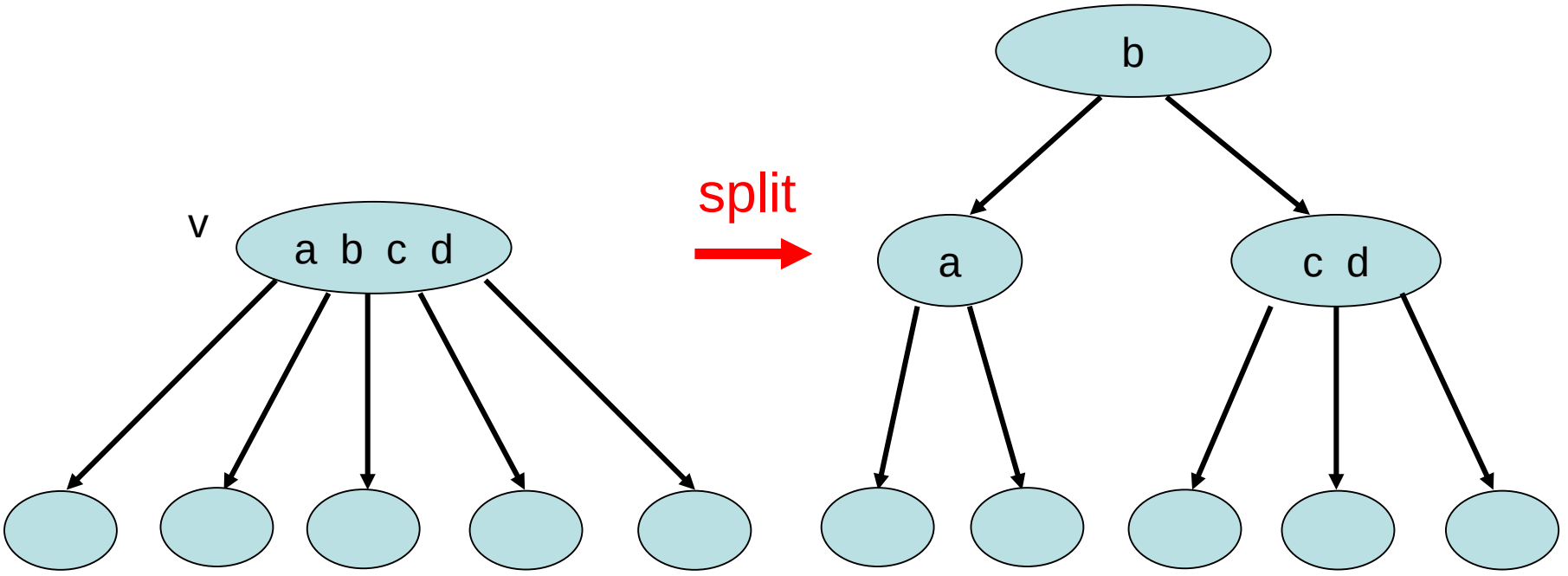
Insert(e) Operation

- If after splitting v, d(w)>b, then cut w into two nodes (and so on, until all nodes have degree ≤b or we reached the root)



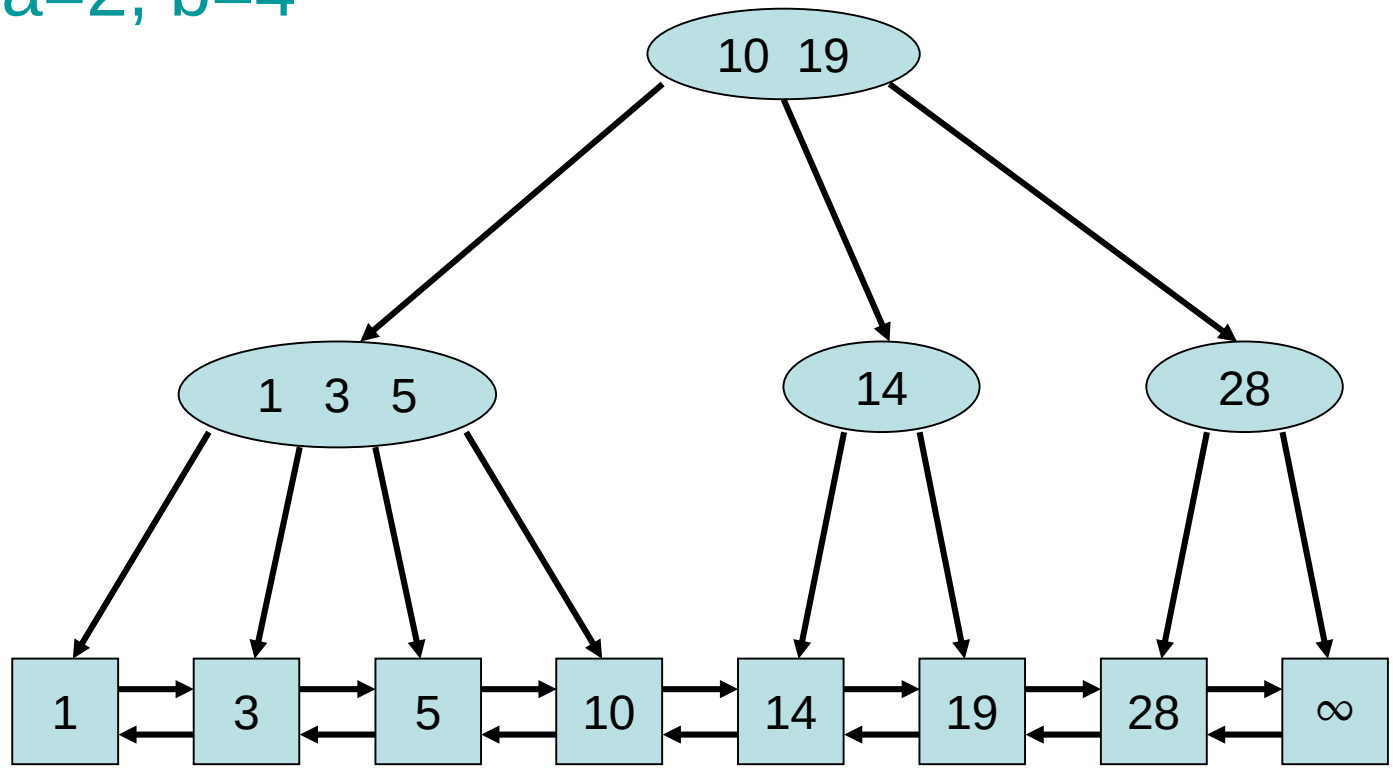
Insert(e) Operation

- If for the root v of T, d(v)>b, then cut v into two nodes and create a new root node.



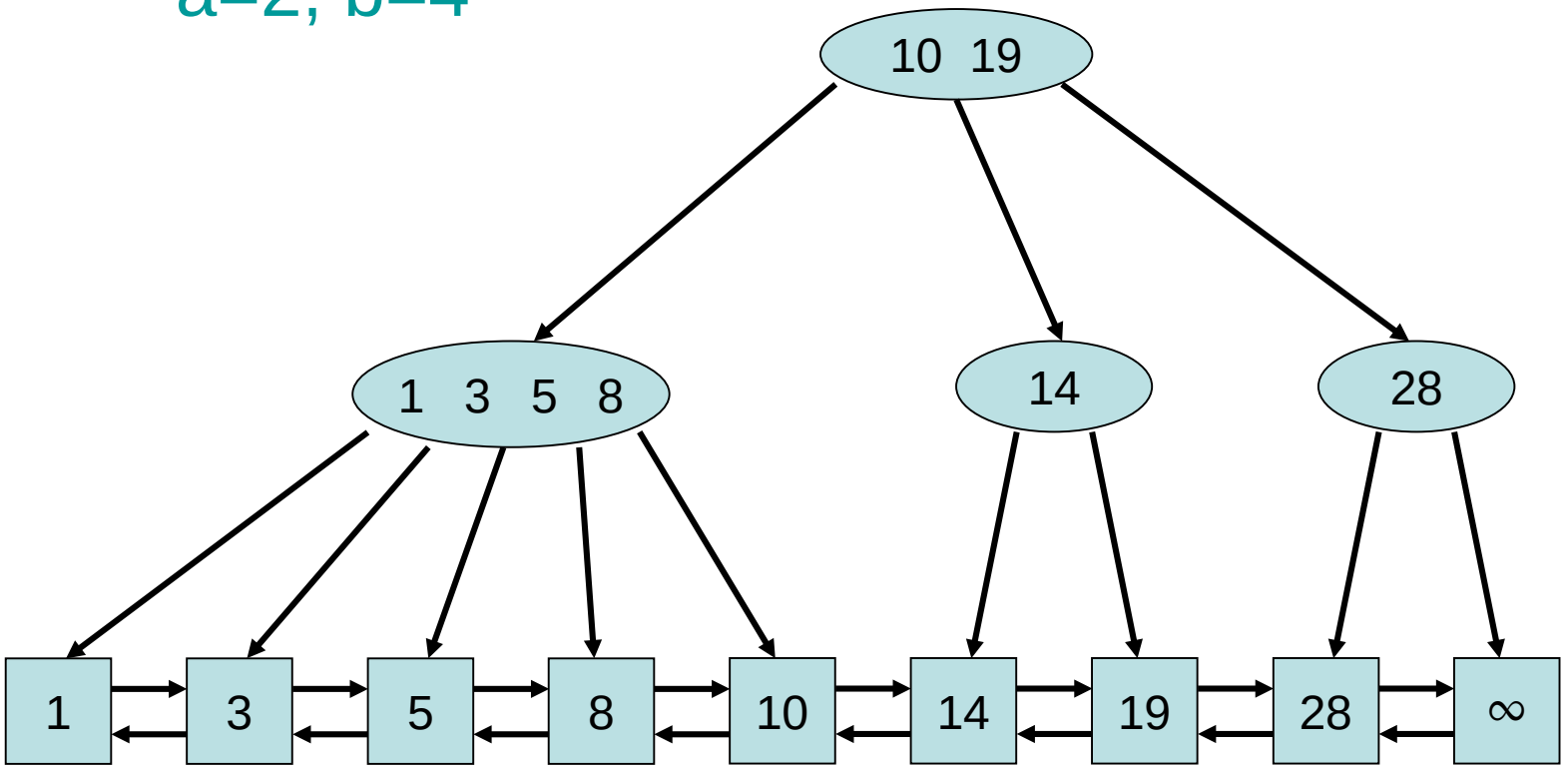
Insert(8)

a=2, b=4



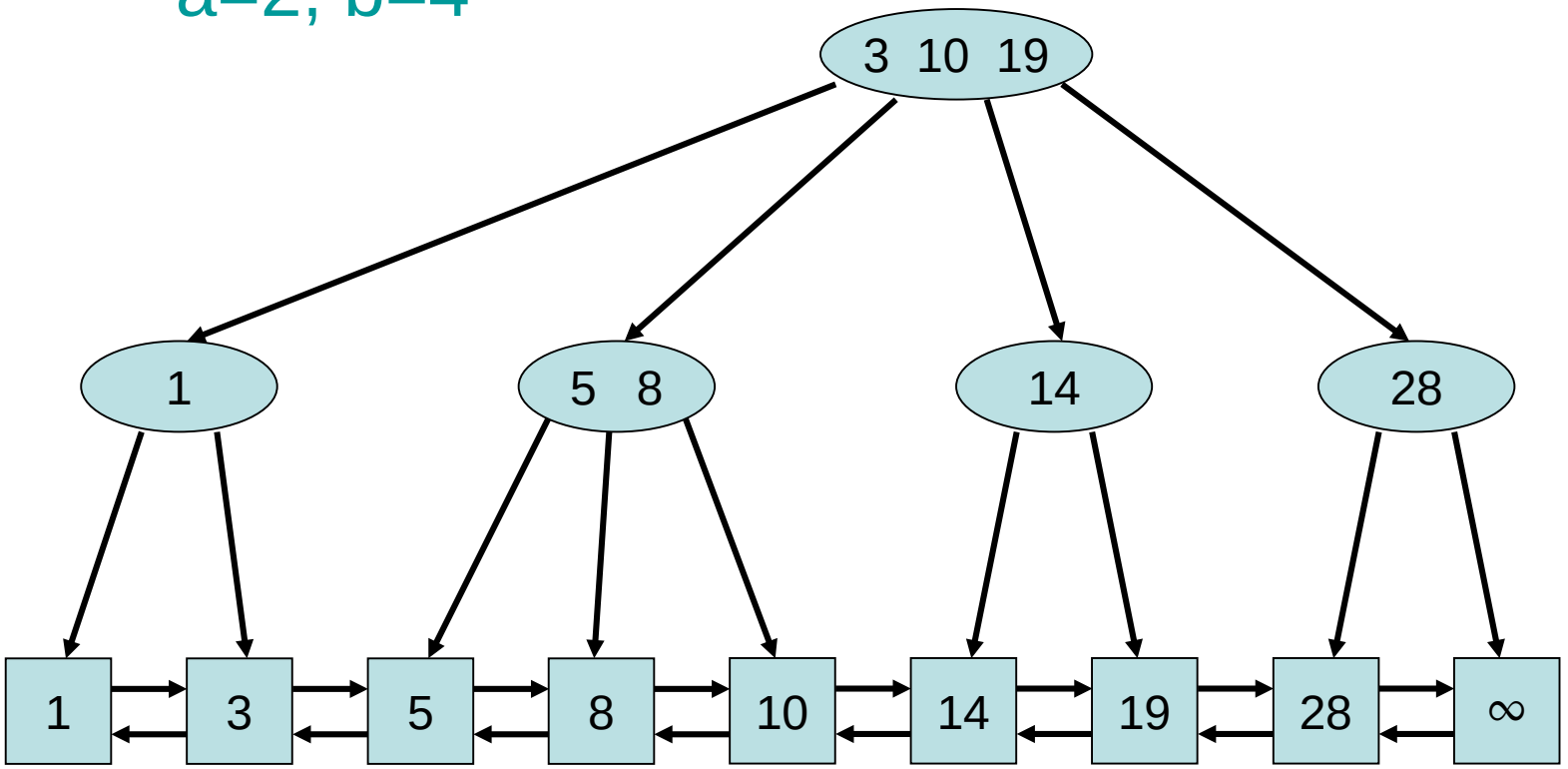
Insert(8)

a=2, b=4



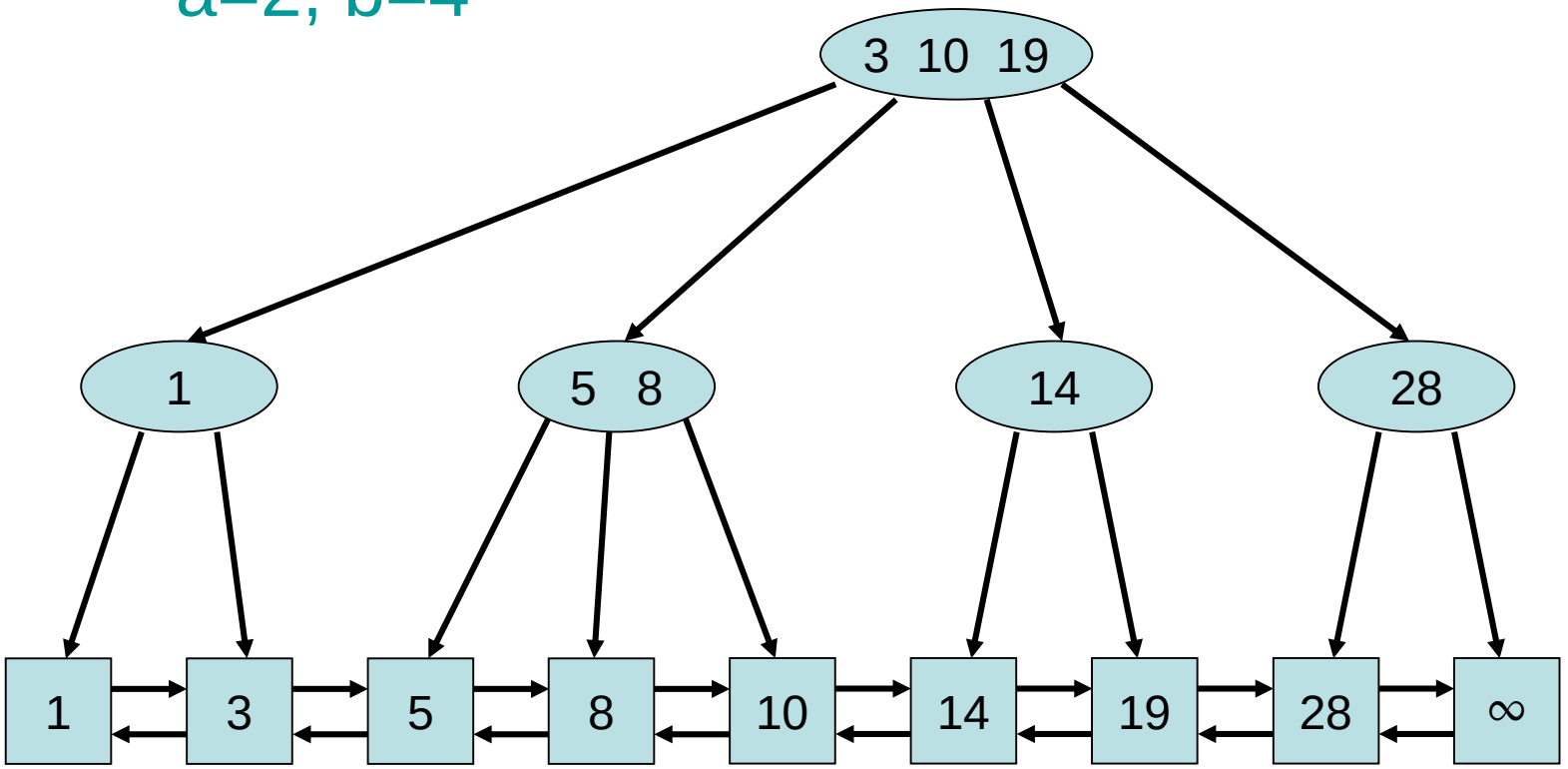
Insert(8)

a=2, b=4



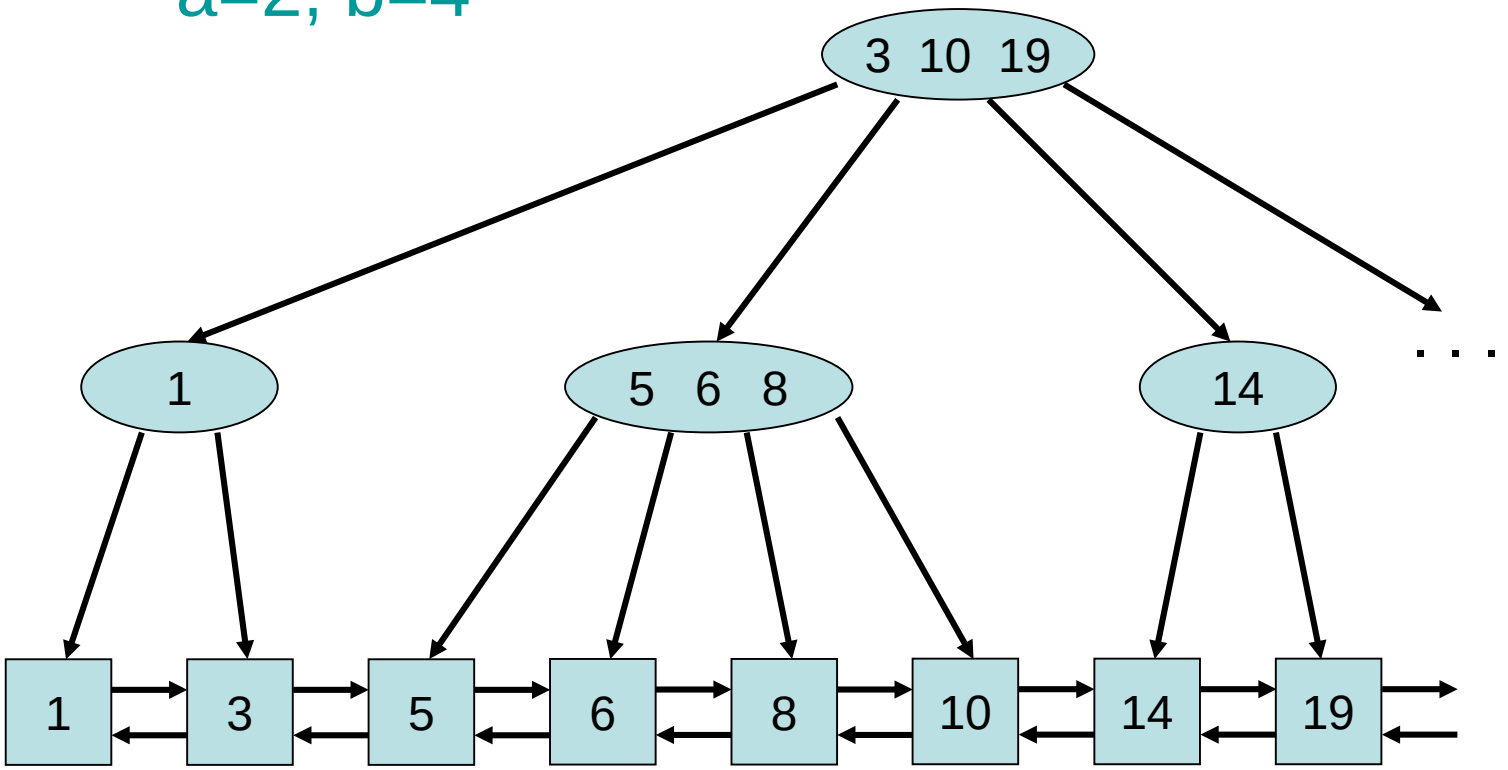
Insert(6)

a=2, b=4



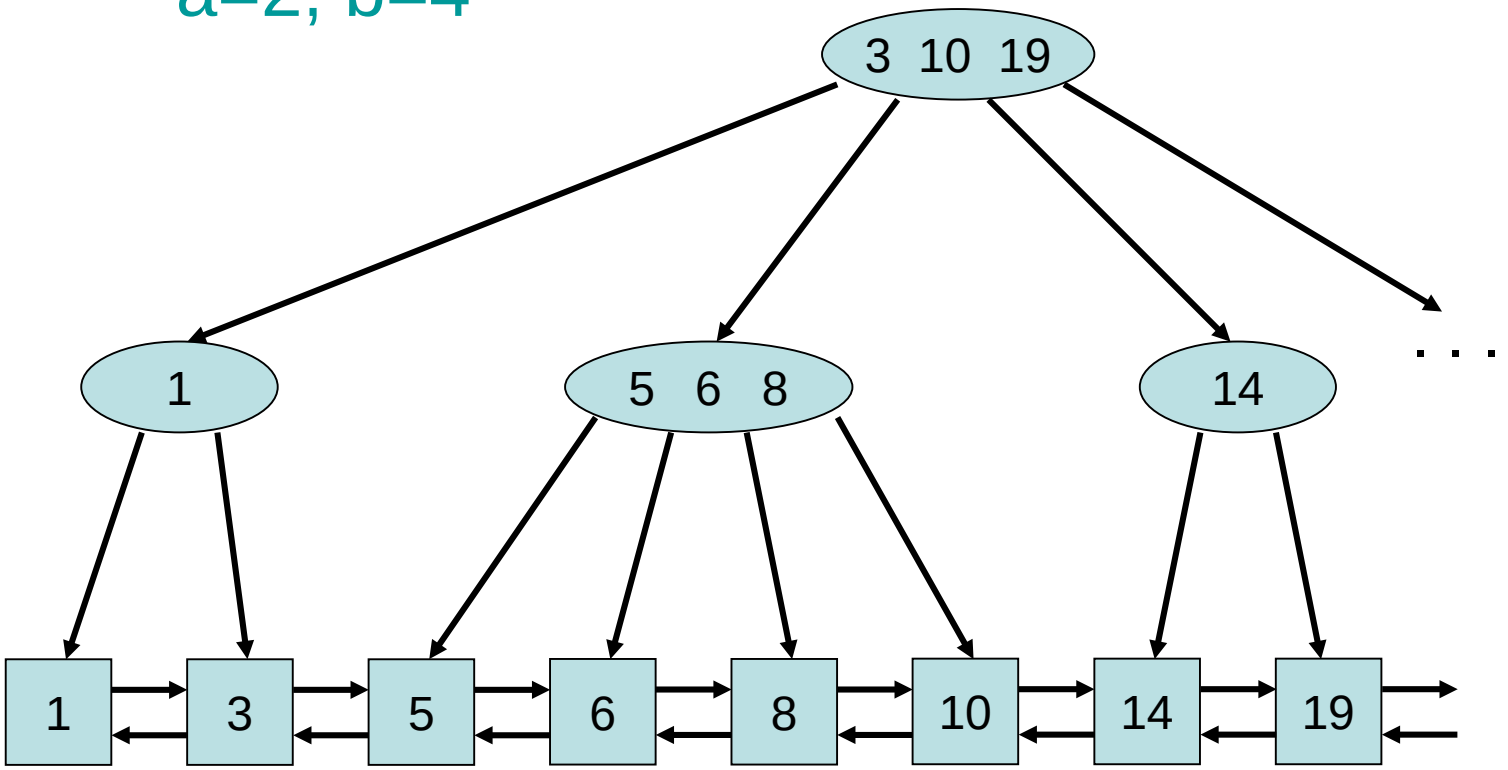
Insert(6)

a=2, b=4



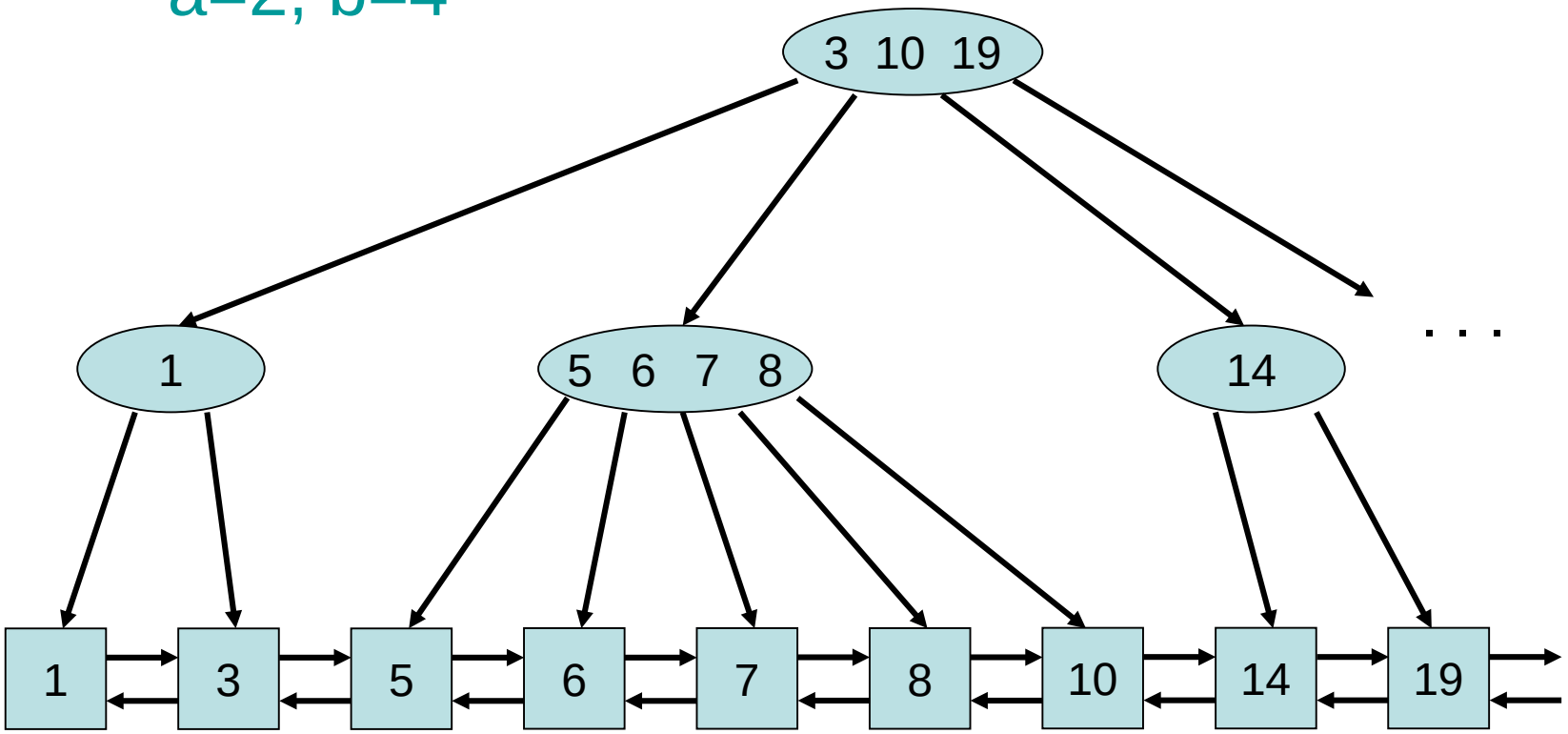
Insert(7)

a=2, b=4



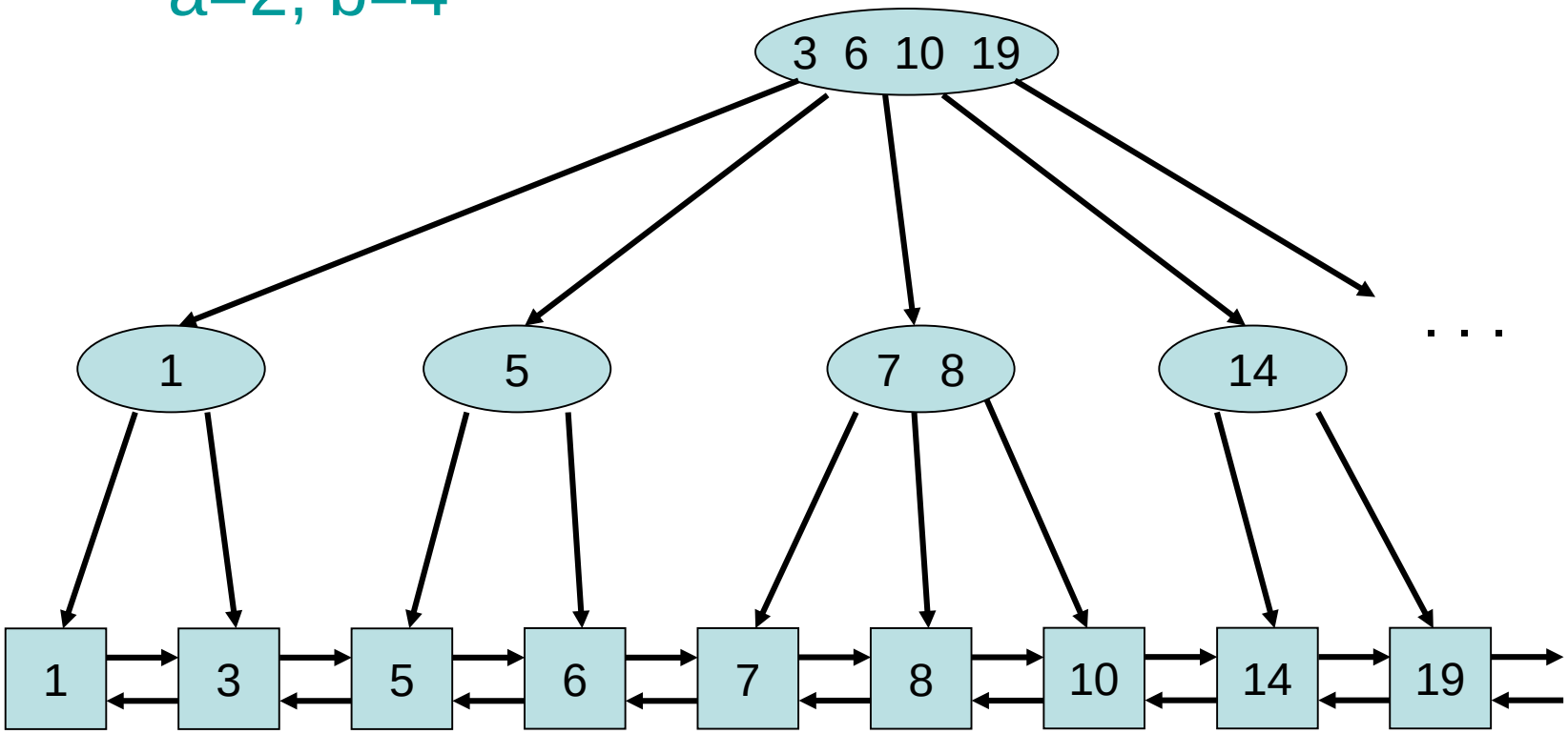
Insert(7)

a=2, b=4

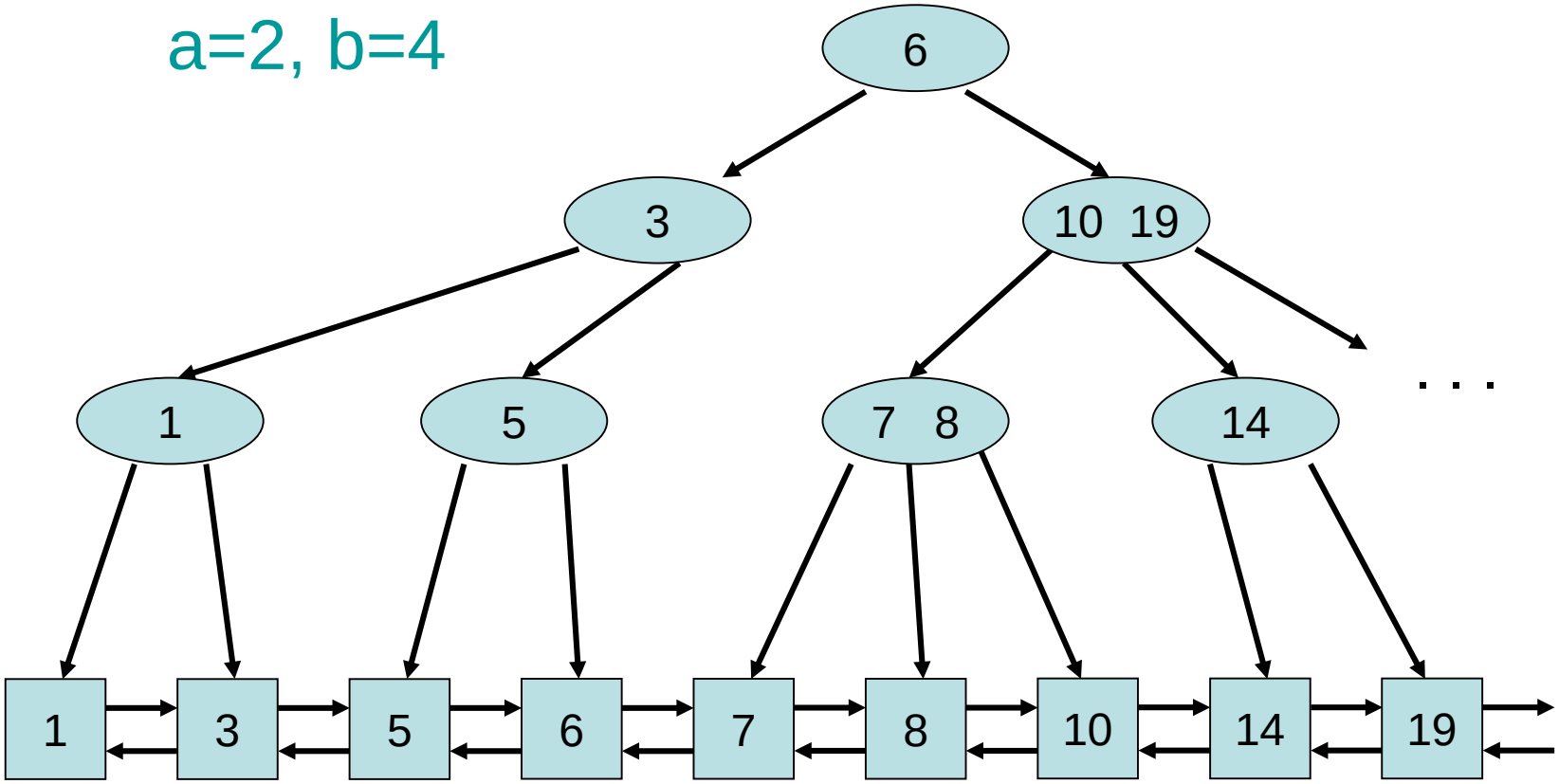


Insert(7)

a=2, b=4



Insert(7)



Insert Operation

- Form Invariant:
  For all leaves v,w: $t(v)=t(w)$
  Satisfied by Insert!

- Degree Invariant:
  For all inner nodes v except for the root:
  $d(v)\in[a,b]$, for root r: $d(r)\in[2,b]$

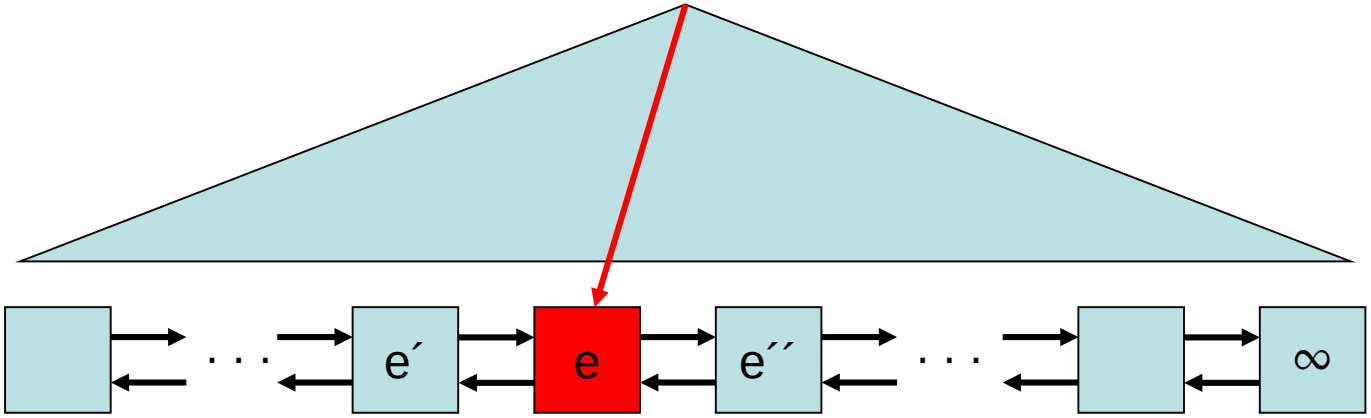
  1) Insert splits nodes of degree $b+1$ into nodes of degree $\lfloor(b+1)/2\rfloor$ and $\lceil(b+1)/2\rceil$. If $b\geq 2a-1$, then both values are at least a.

  2) If root has reached degree $b+1$, then a new root of degree 2 is created.

Delete(k) Operation

Strategy:

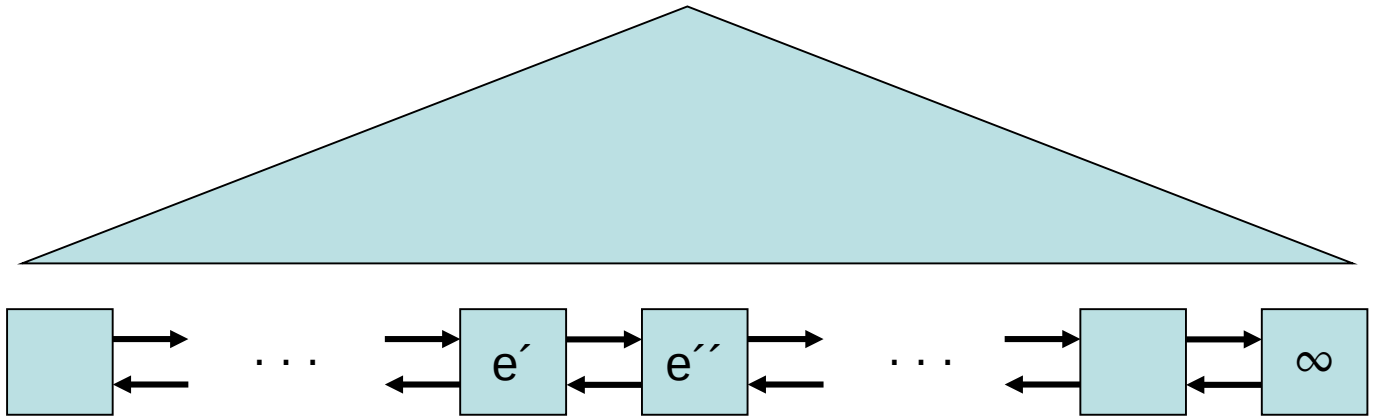
- First search(k) until some element e is reached in the list. If key(e)=k, remove e from the list, otherwise we are done.



Delete(k) Operation

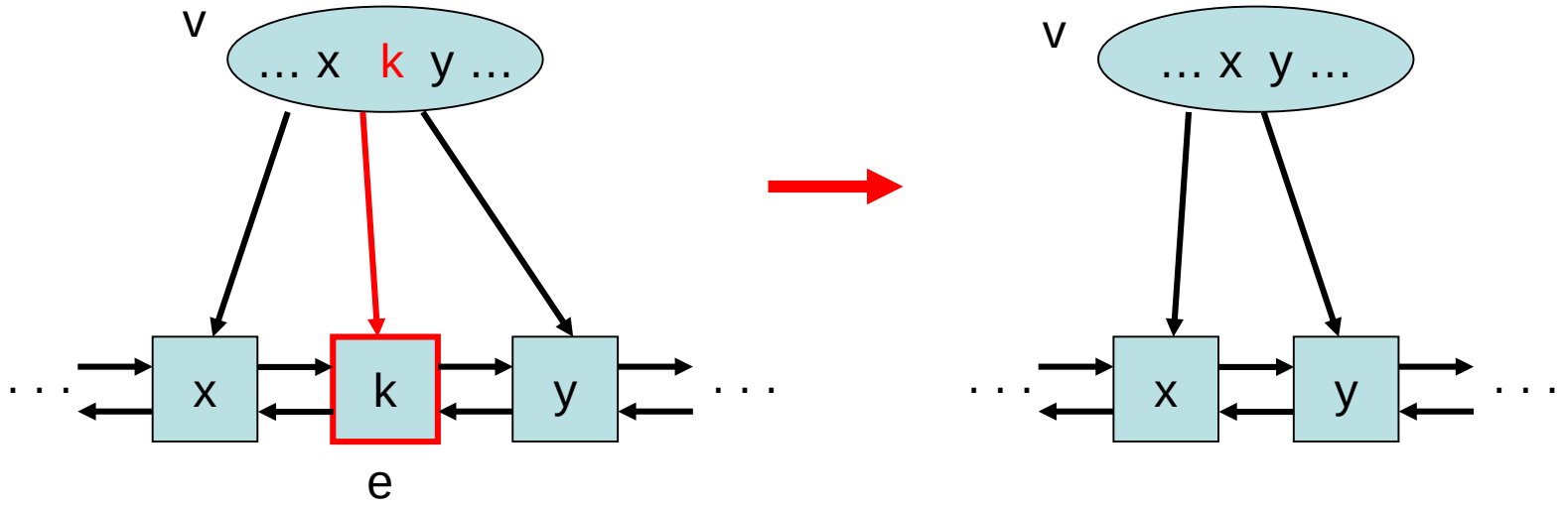
Strategy:

- First search(k) until some element e is reached in the list. If key(e)=k, remove e from the list, otherwise we are done.



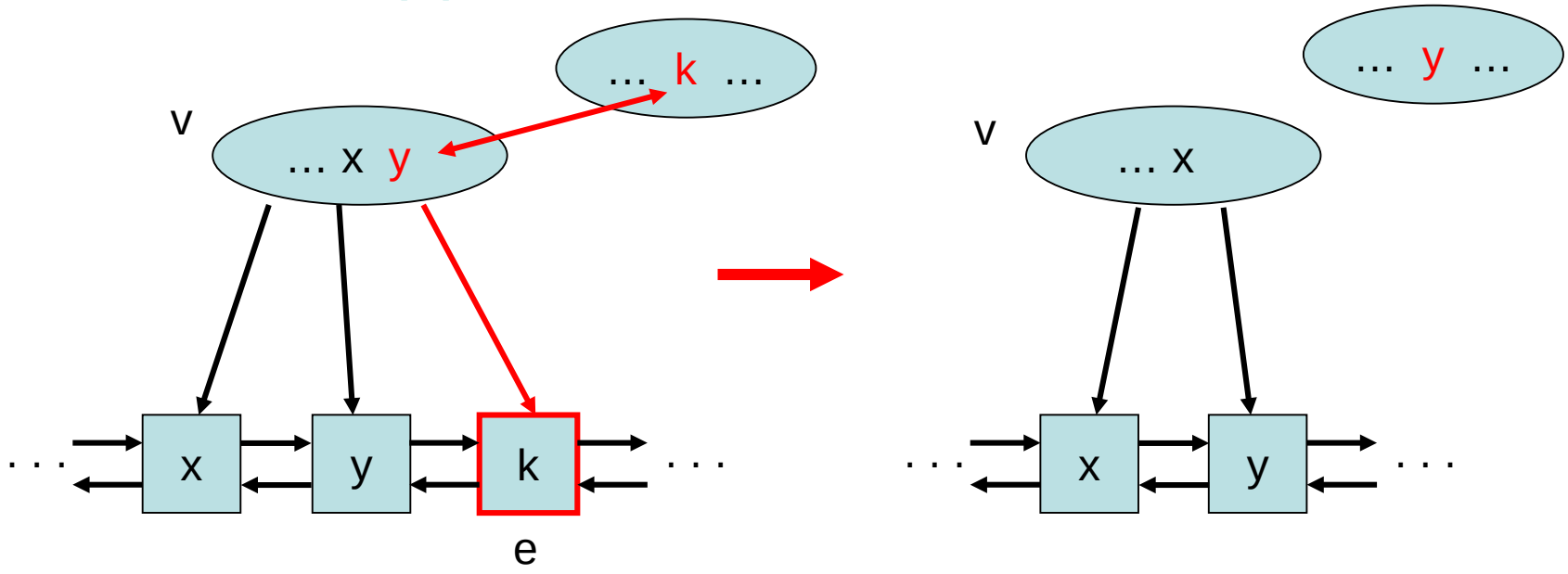
Delete(k) Operation

- Remove pointer to e and key k from the leaf node v above e. (e rightmost child: perform key exchange like in binary tree!) If afterwards we still have $d(v) \geq a$, we are done.



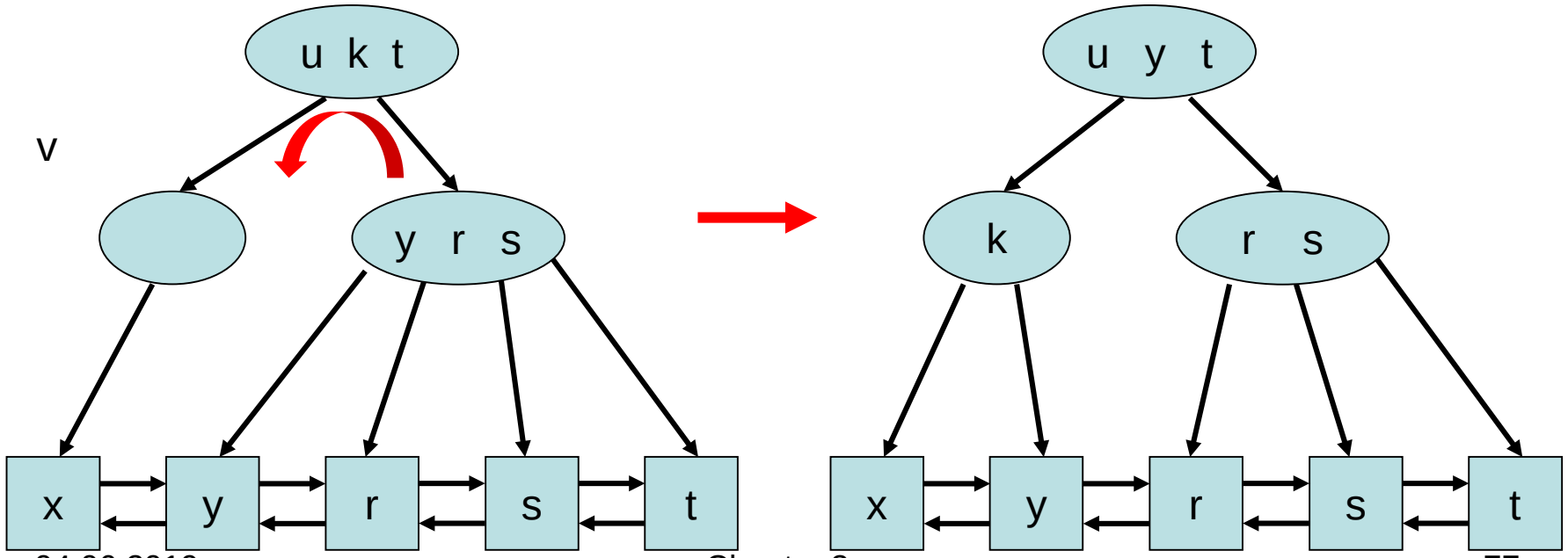
Delete(k) Operation

- Remove pointer to e and key k from the leaf node v above e. (e rightmost child: perform key exchange like in binary tree!) If afterwards we still have d(v)≥a, we are done.



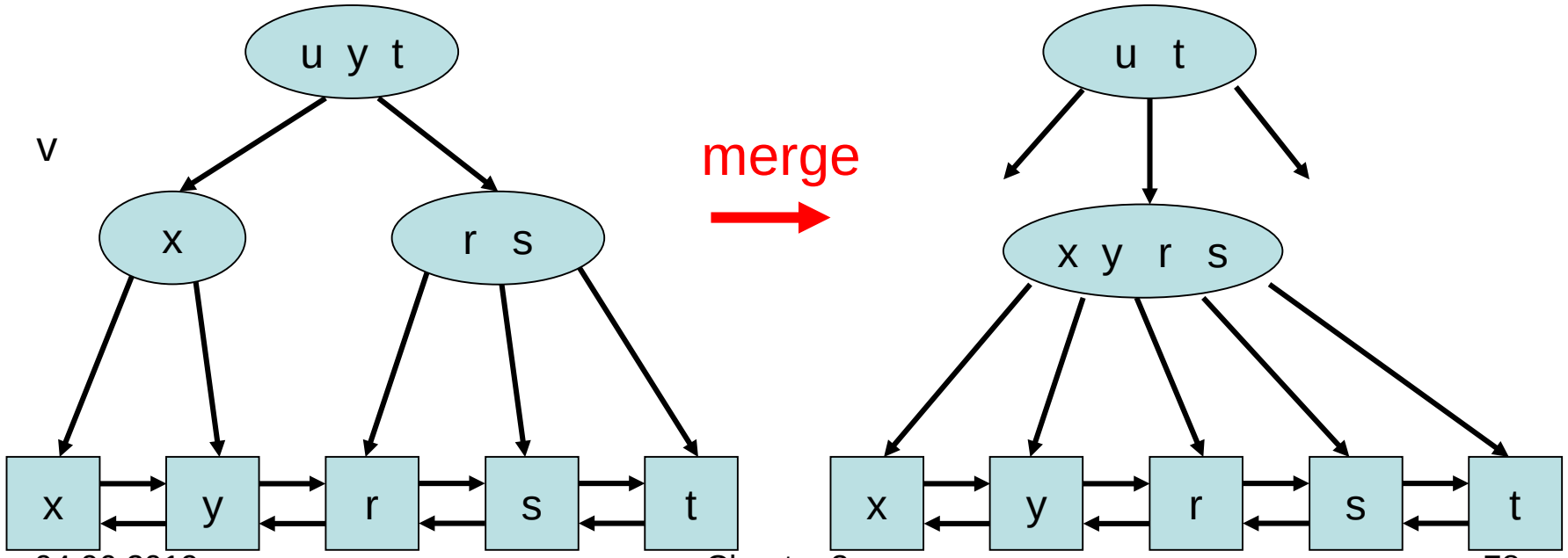
Delete(k) Operation

- If d(v)<a and the preceding or succeeding sibling of v has degree >a, steal an edge from that sibling. (Example: a=2, b=4)



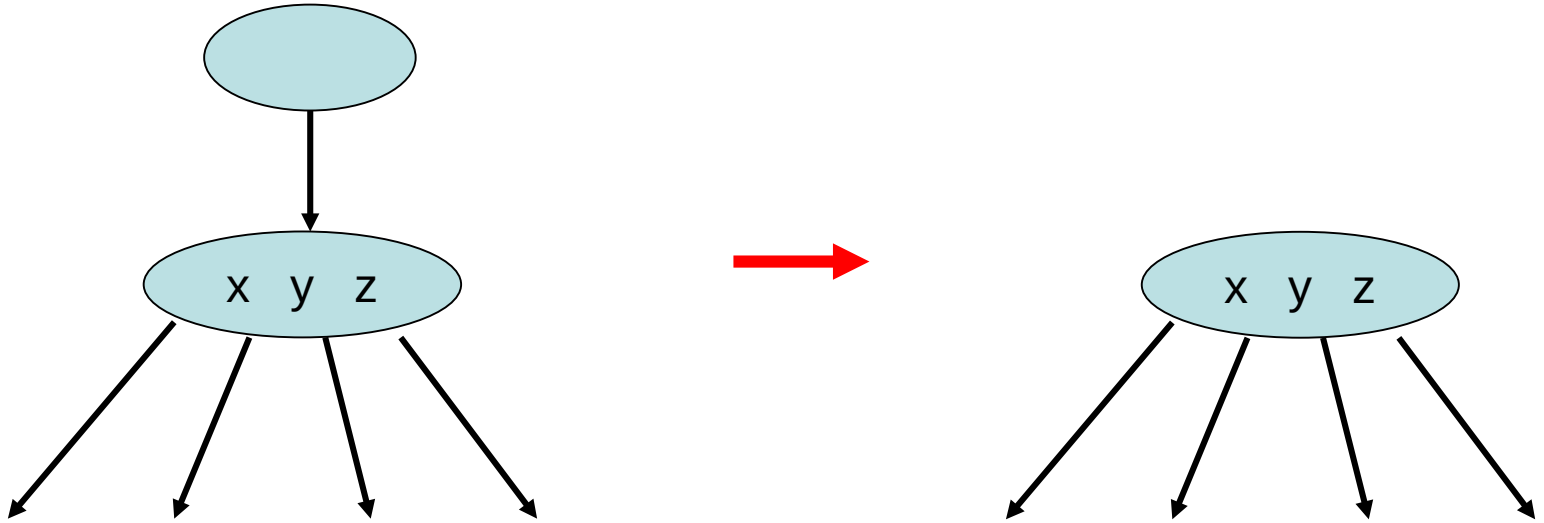
Delete(k) Operation

- If d(v)<a and the preceeding and succeeding siblings of v have degree a, merge v with one of these. (Example: a=3, b=5)



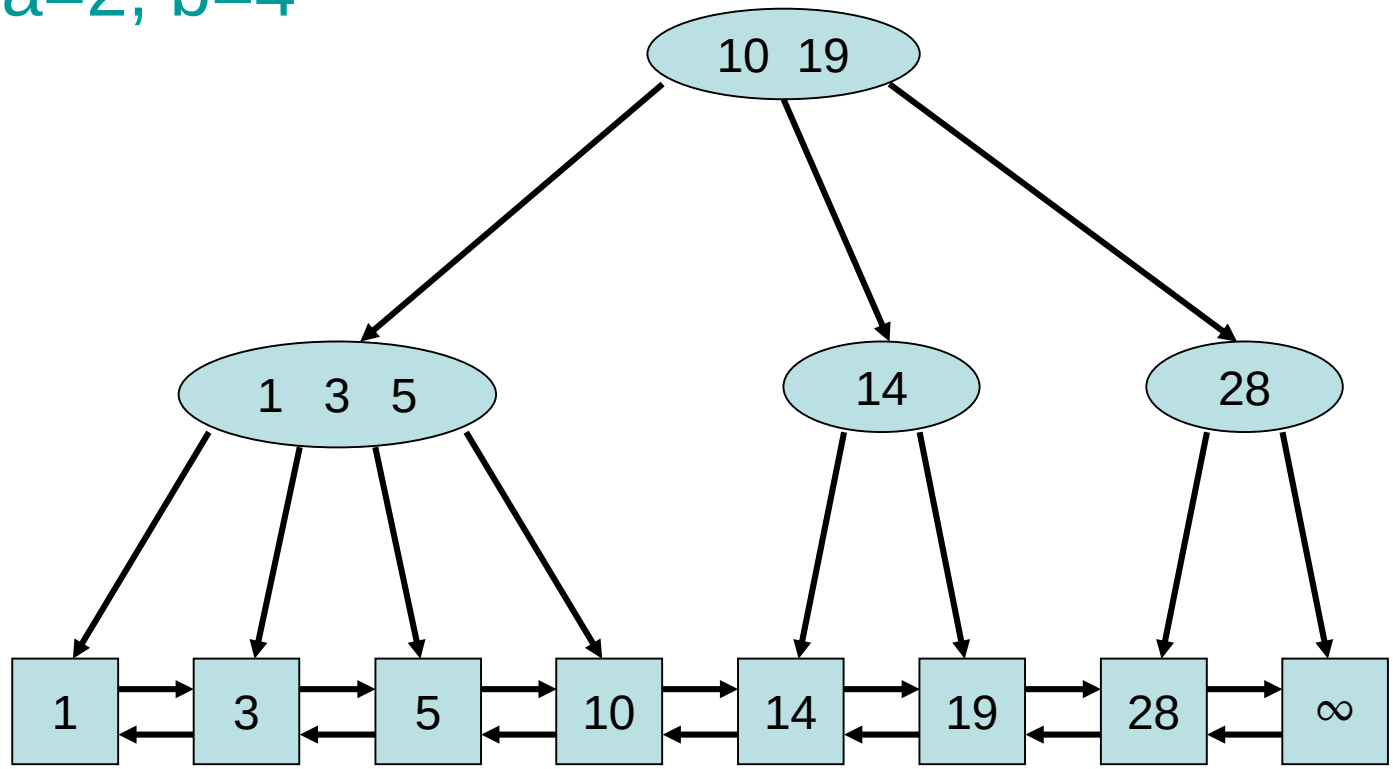
Delete(k) Operation

- Peform changes upwards until all inner nodes (except for the root) have degree $\geq a$. If root has degree <2: remove root.



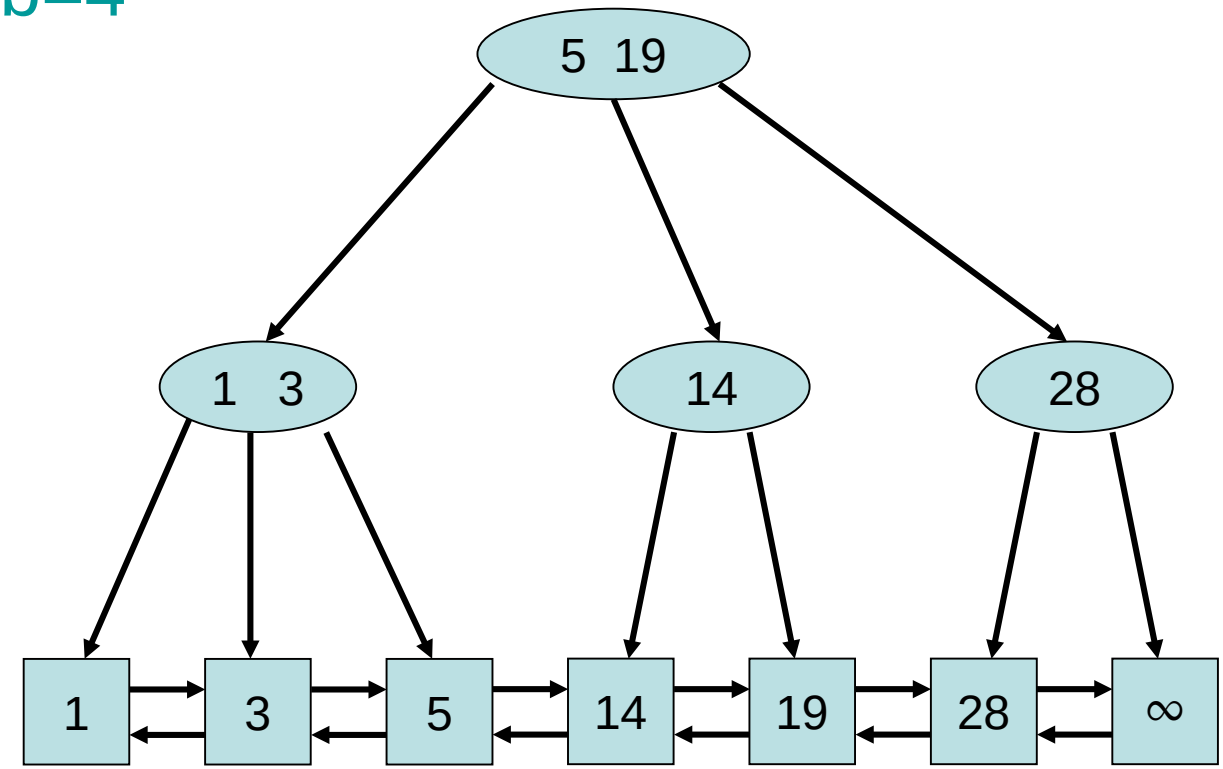
Delete(10)

a=2, b=4



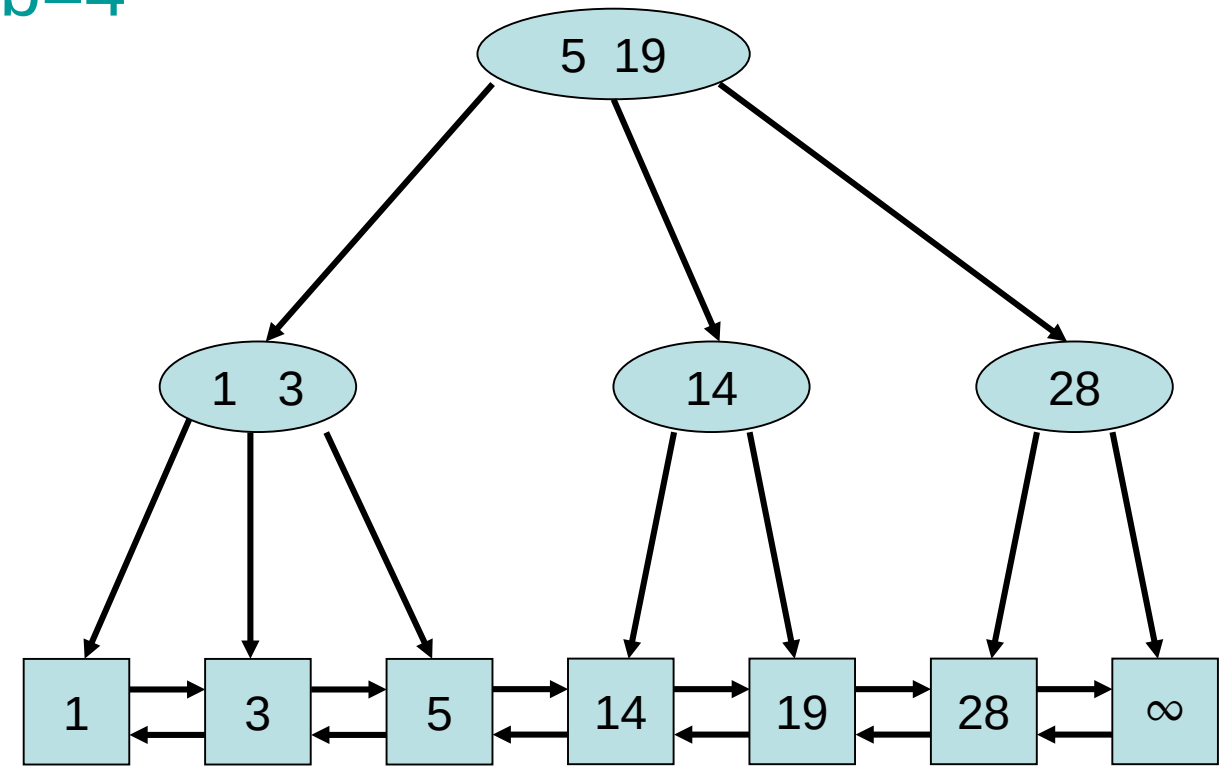
Delete(10)

a=2, b=4



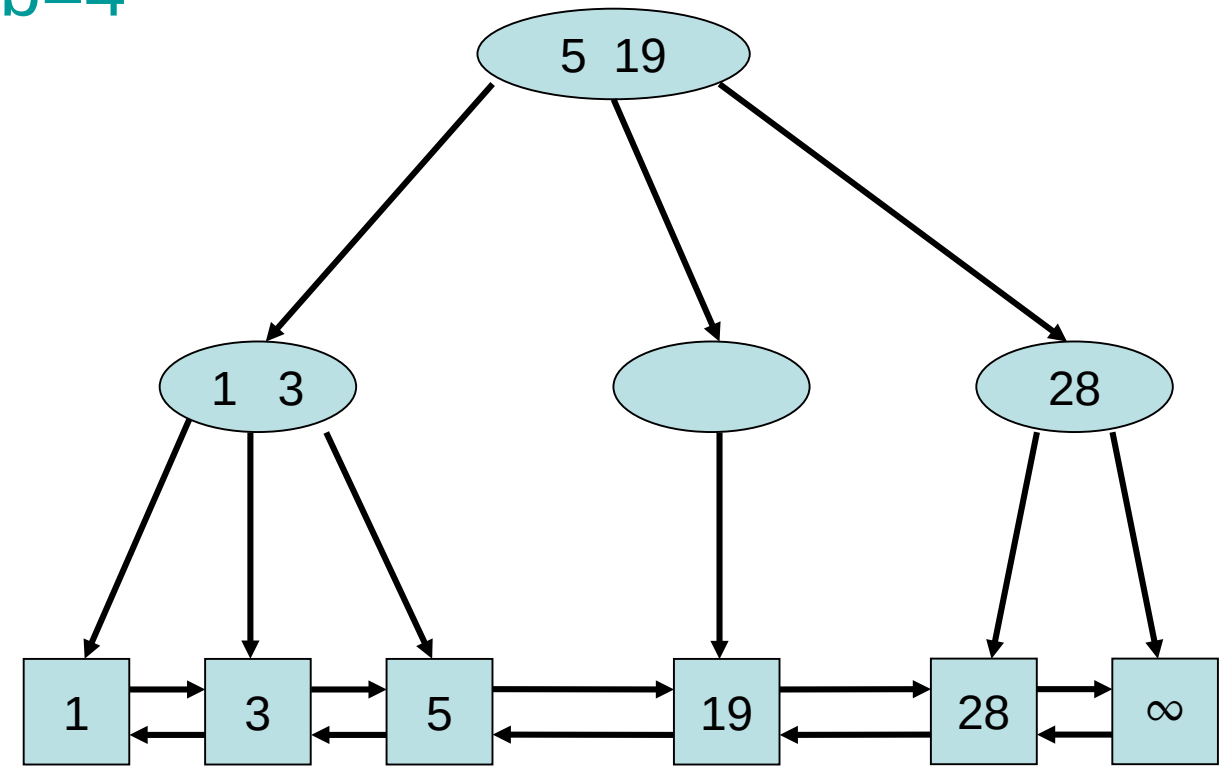
Delete(14)

a=2, b=4



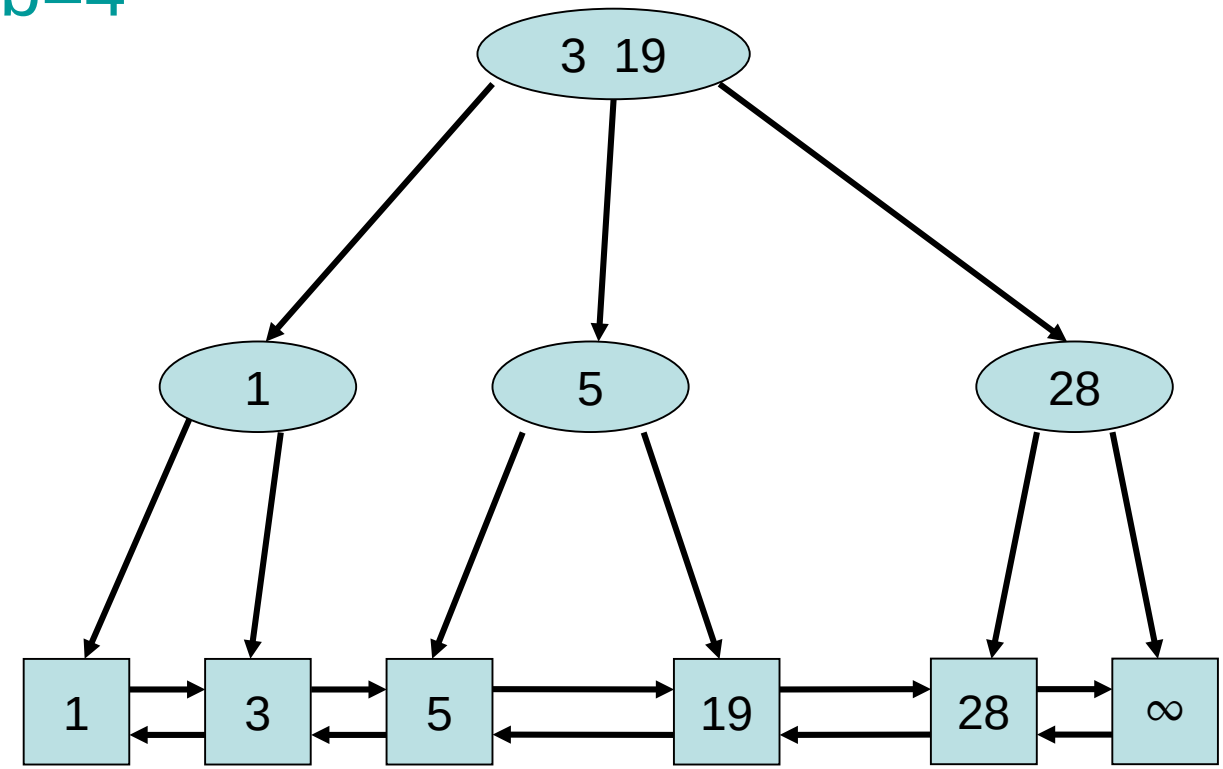
Delete(14)

a=2, b=4



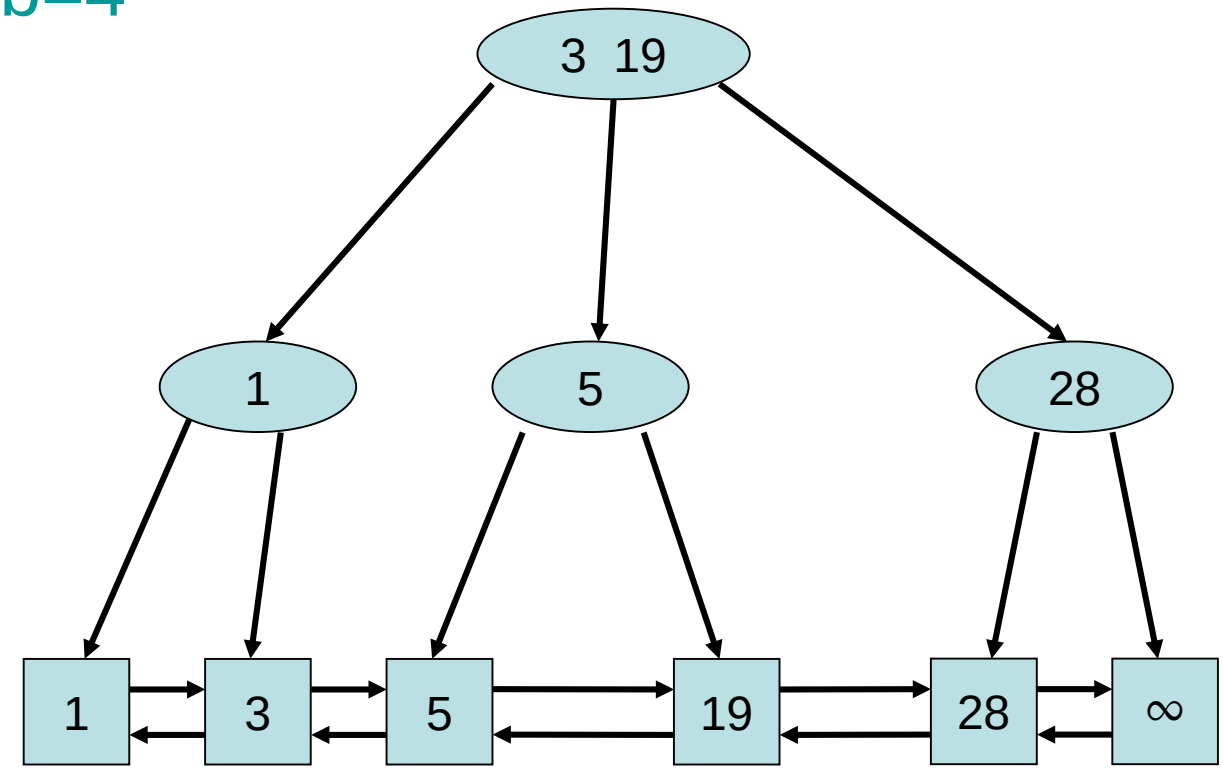
Delete(14)

a=2, b=4



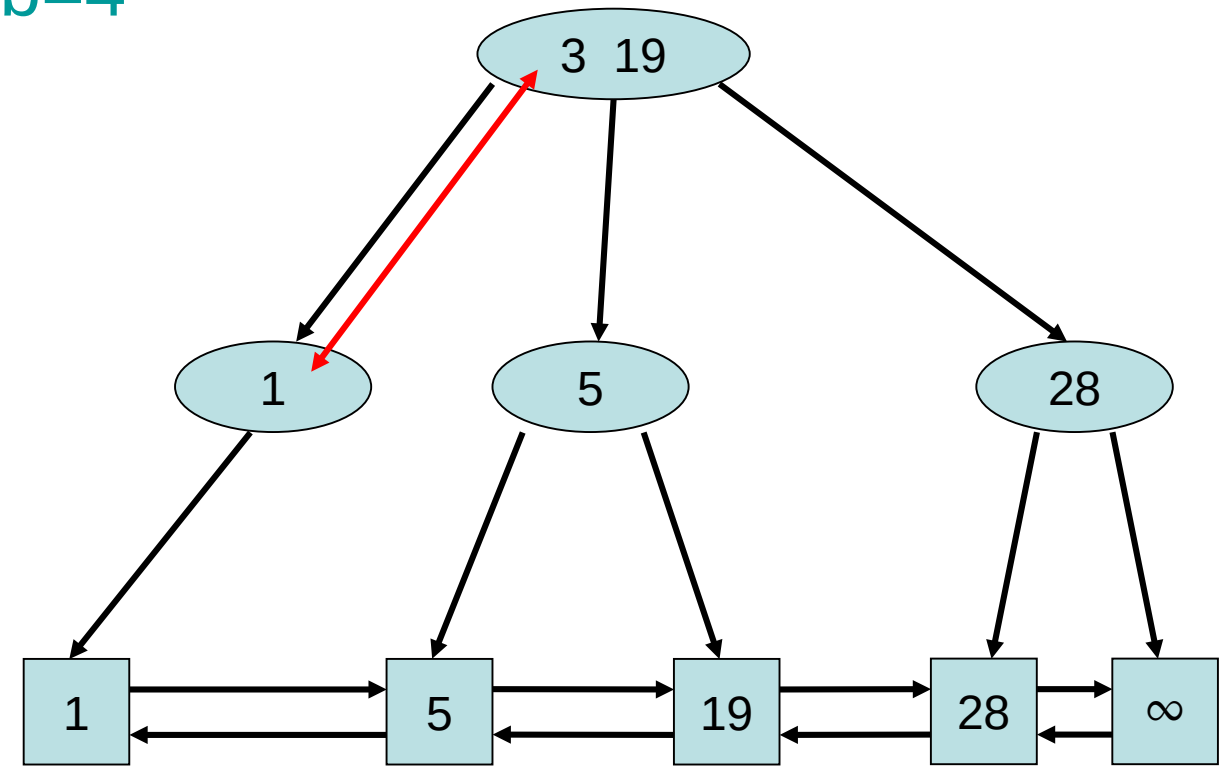
Delete(3)

a=2, b=4



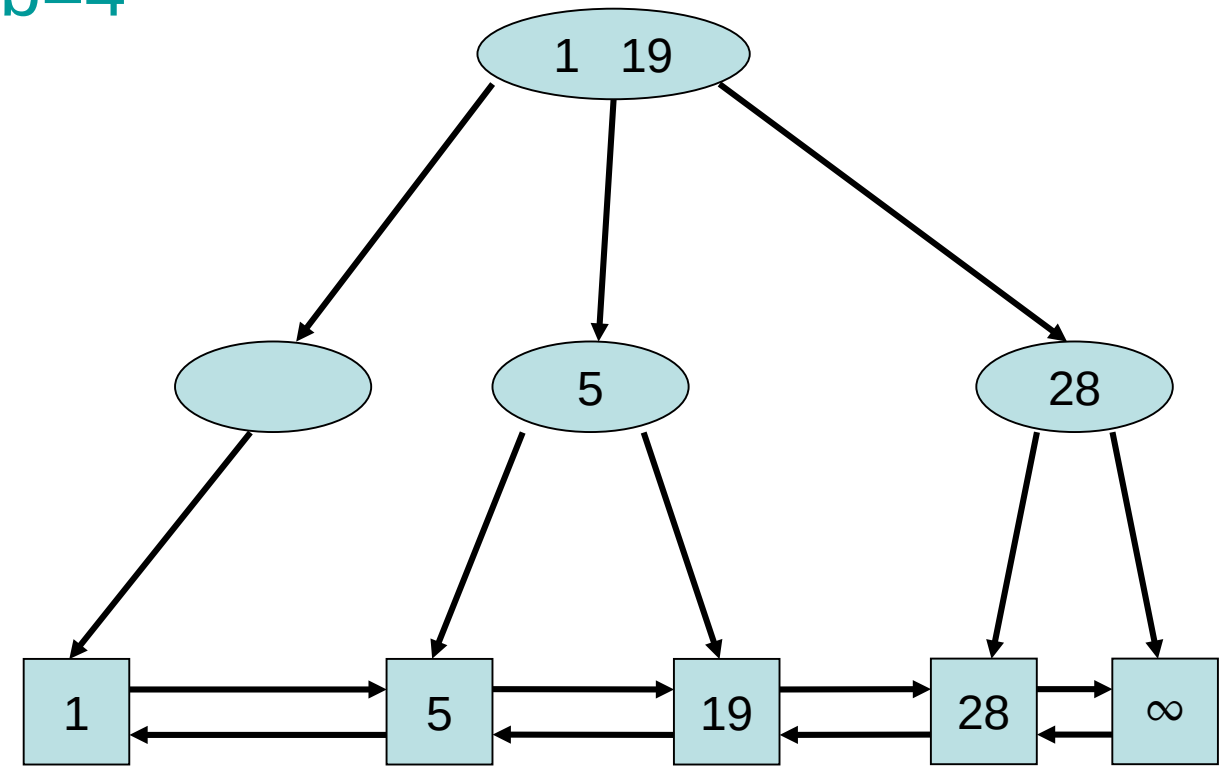
Delete(3)

a=2, b=4



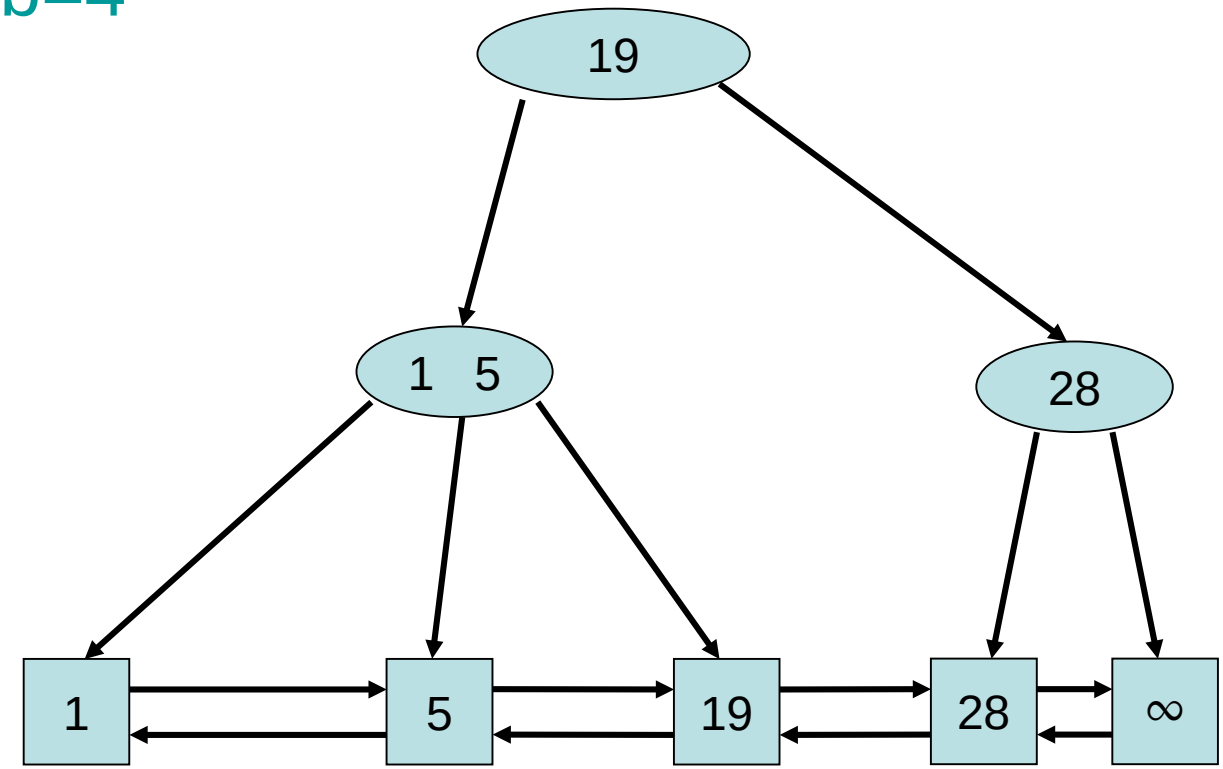
Delete(3)

a=2, b=4



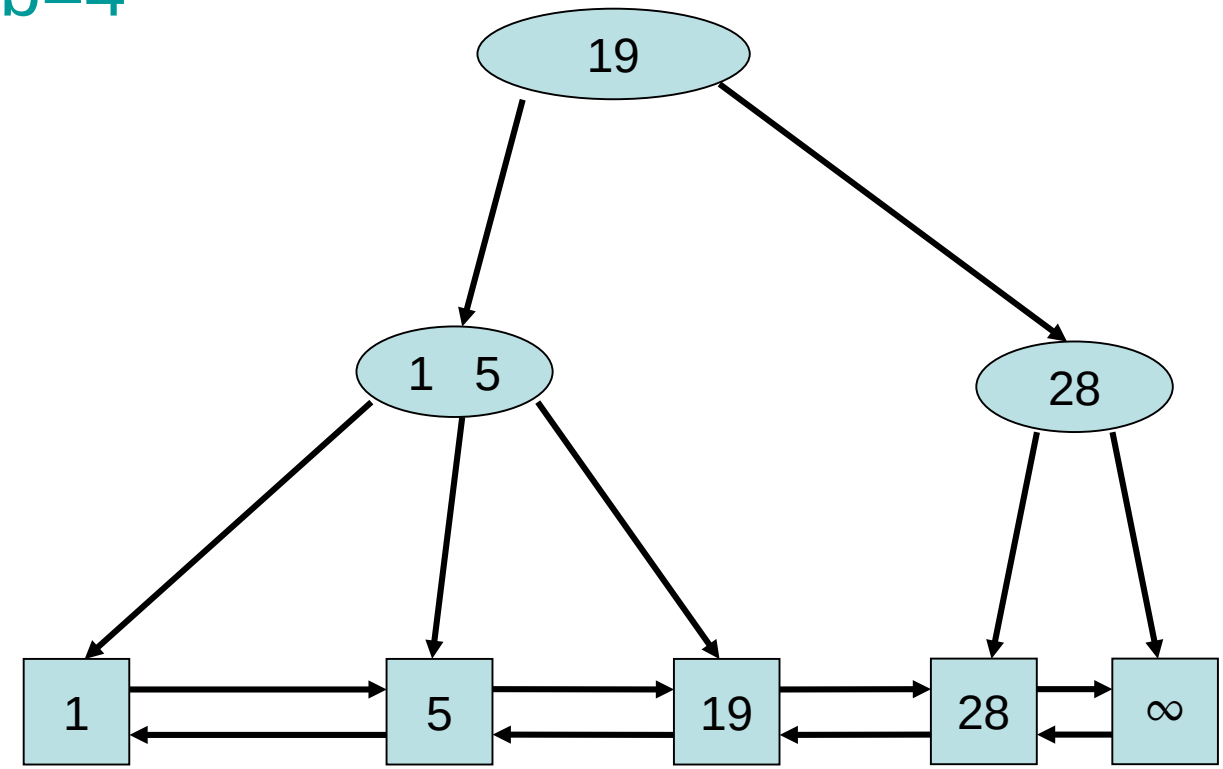
Delete(3)

a=2, b=4



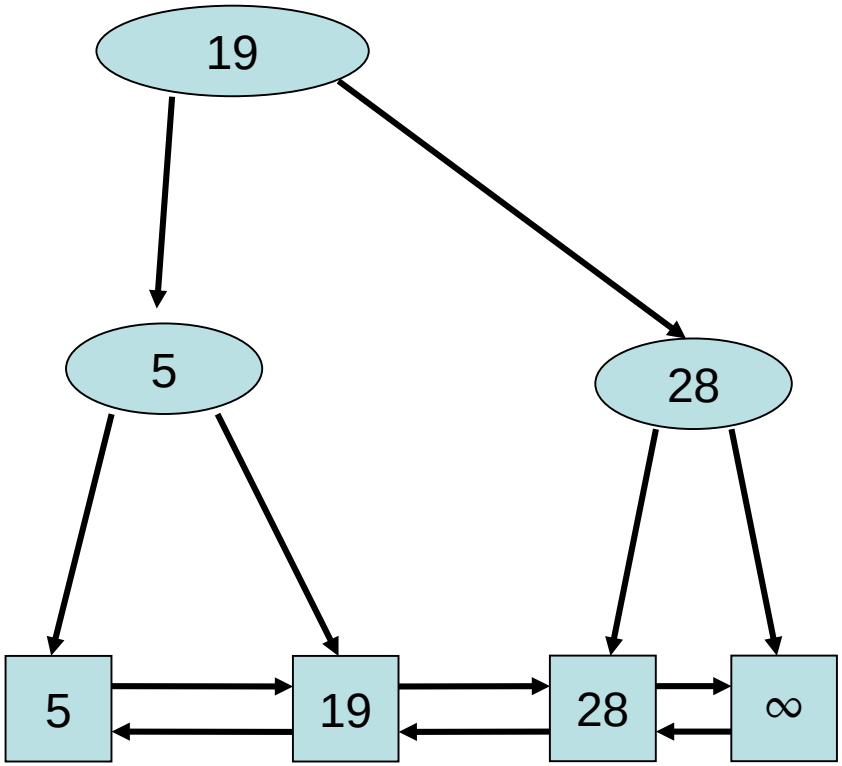
Delete(1)

a=2, b=4



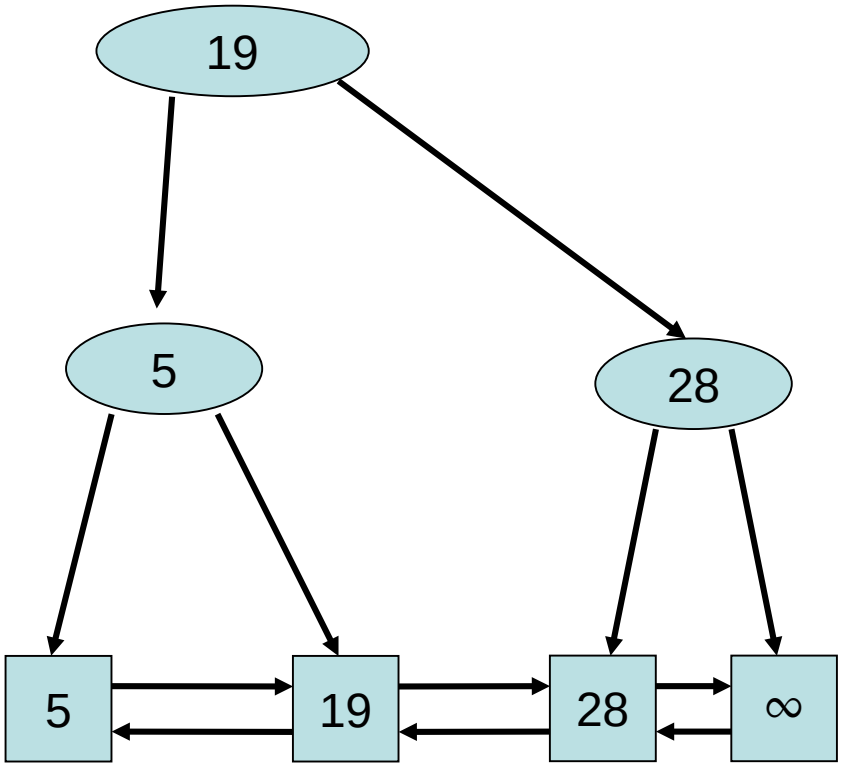
Delete(1)

a=2, b=4



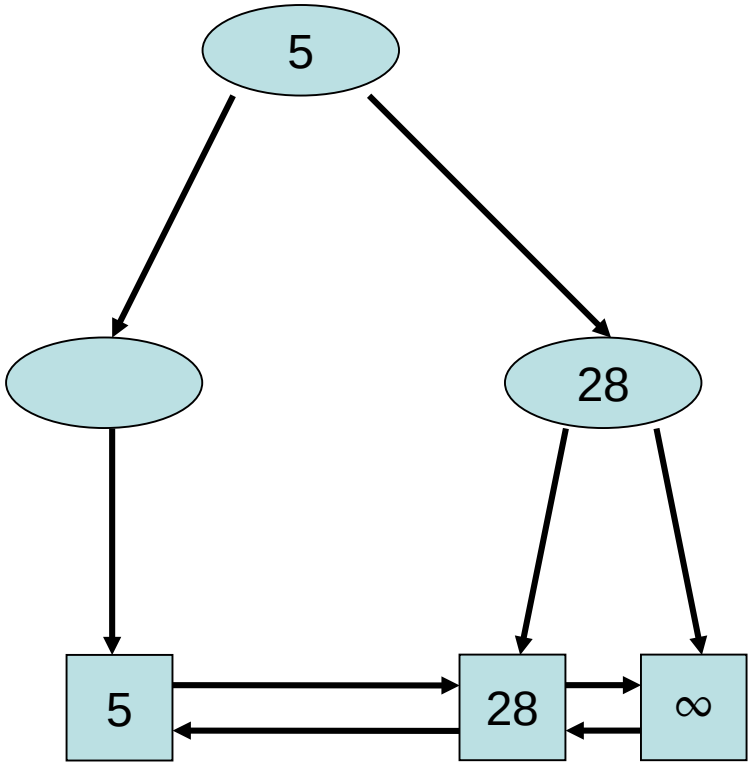
Delete(19)

a=2, b=4



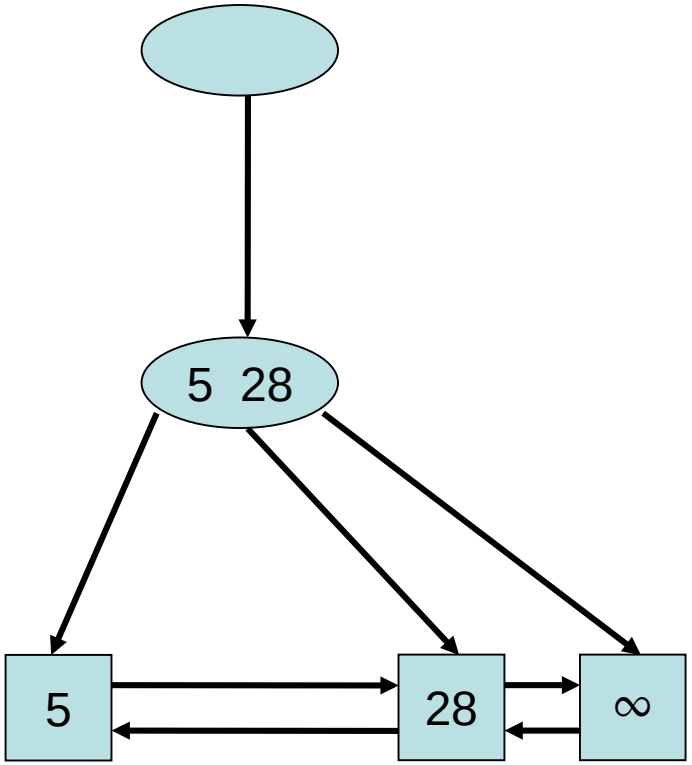
Delete(19)

a=2, b=4



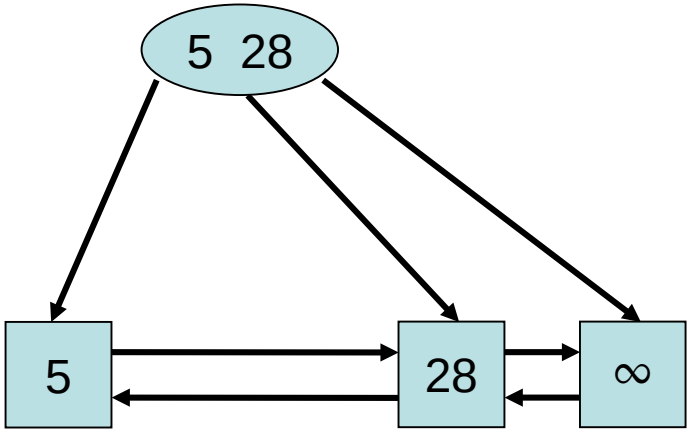
Delete(19)

a=2, b=4



Delete(19)

a=2, b=4



Delete Operation

- Form Invariant:
  For all leaves v,w: $t(v)=t(w)$
  Satisfied by Delete!

- Degree Invariant:
  For all inner nodes v except for the root: $d(v)\in[a,b]$, for root r: $d(r)\in[2,b]$

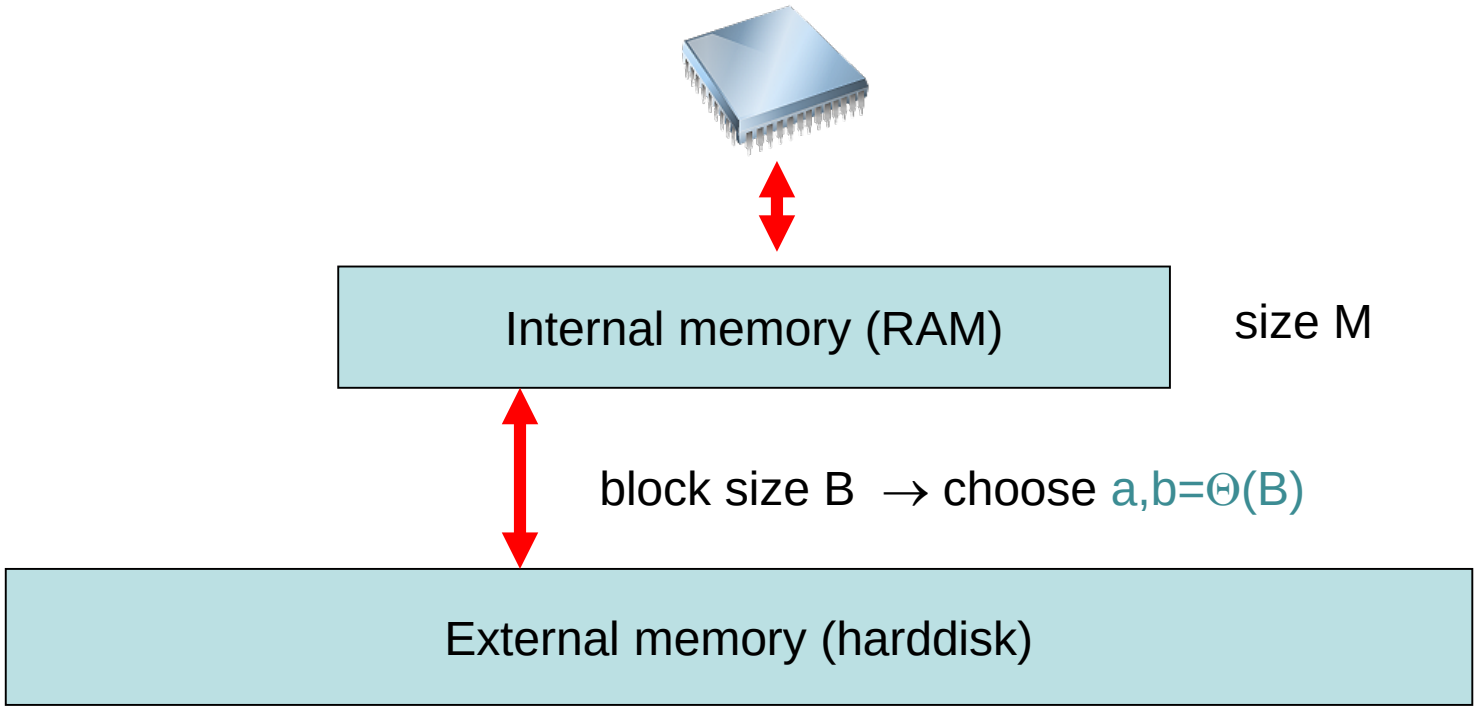
  1) Delete merges node of degree $a-1$ with node of degree a. Since $b\geq 2a-1$, the resulting node has degree at most b.

  2) Delete moves edge from a node of degree $>a$ to a node of degree $a-1$. Also OK.

  3) Root deleted: children have been merged, degree of the remaining child is $\geq a$ (and also $\leq b$), so also OK.

Application: External (a,b)-Tree

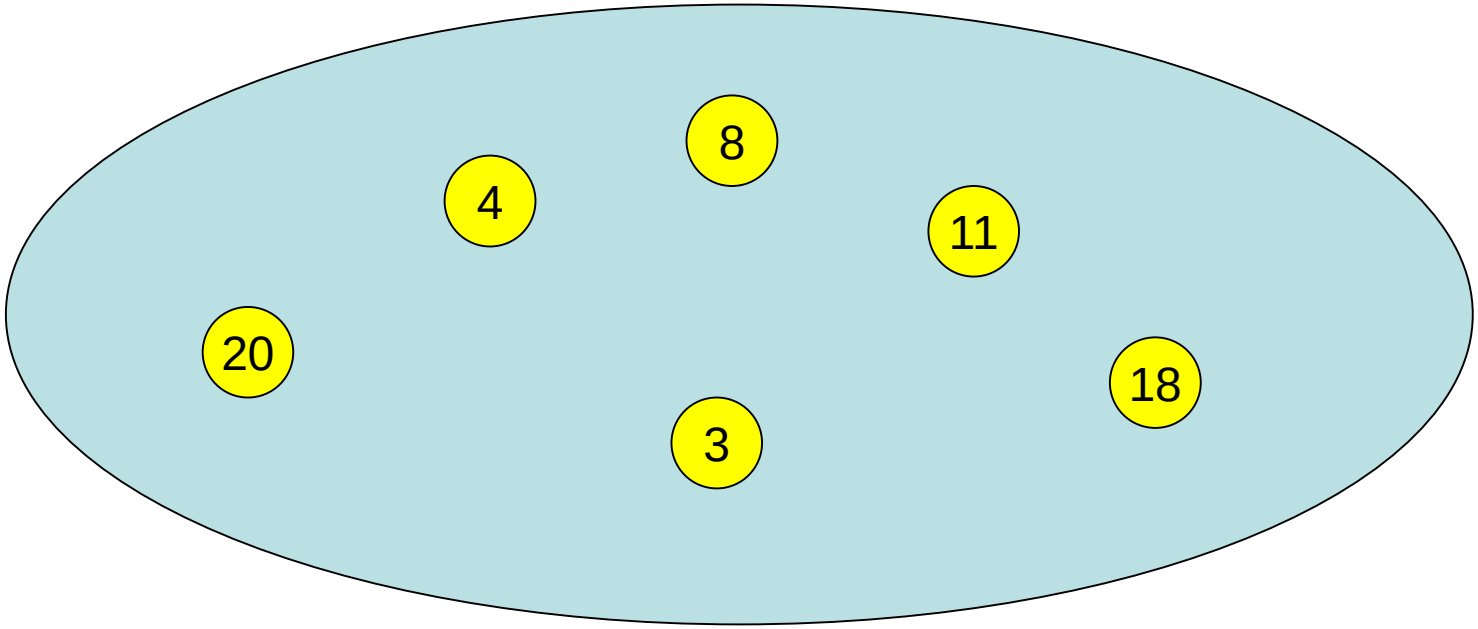
(a,b)-trees well suited for large amounts of data



Overview

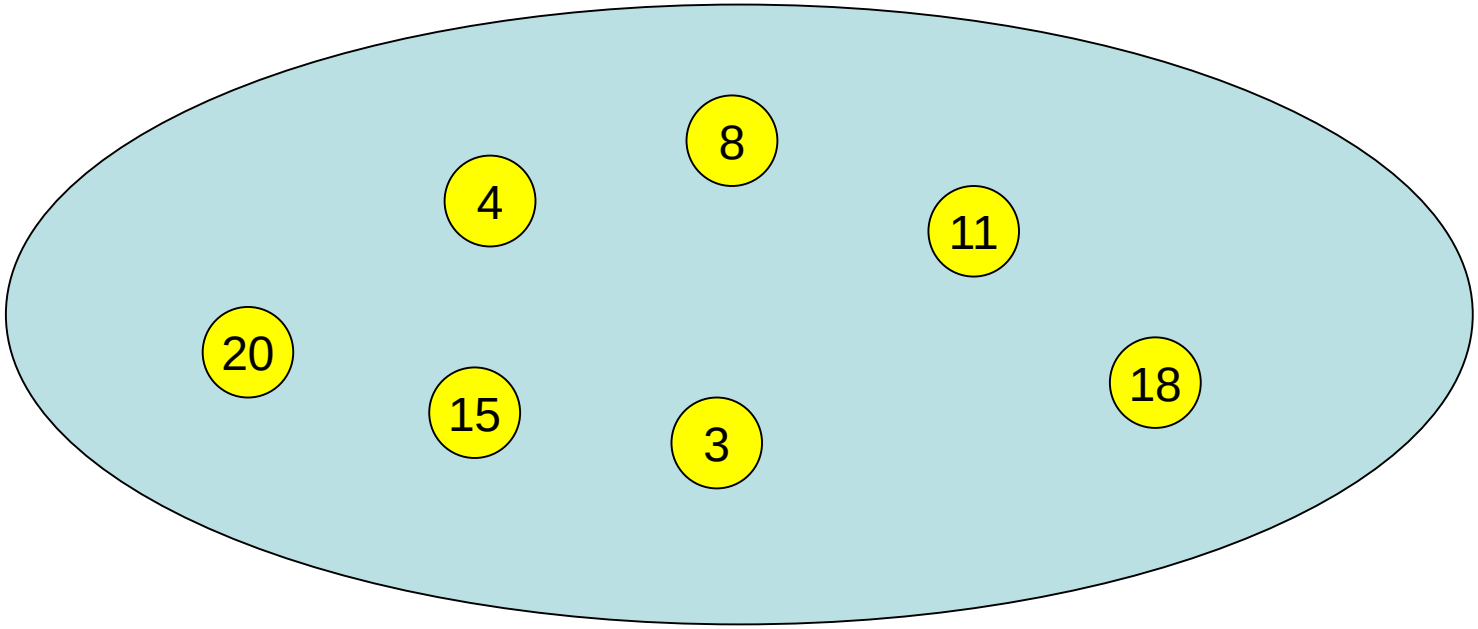
- Basic data structures
- Search structures (successor searching)
- Dictionaries (exact searching)

Dictionary



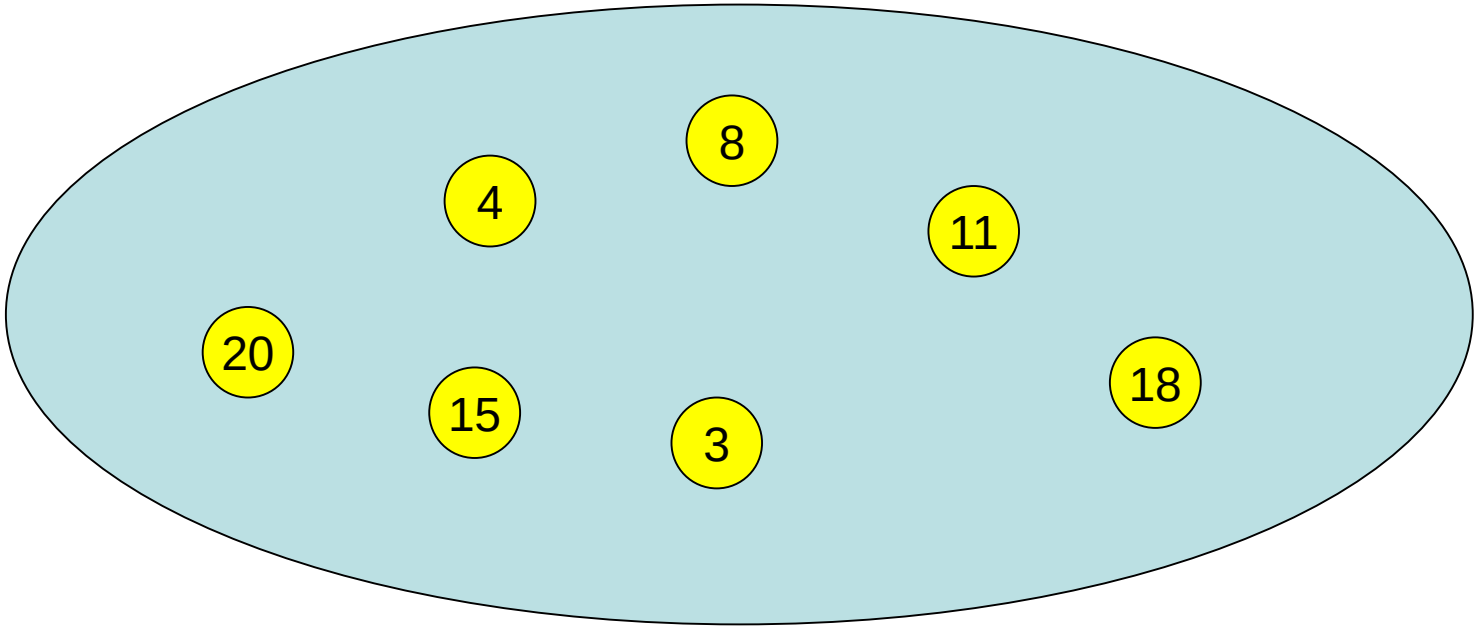
Dictionary

insert(15)



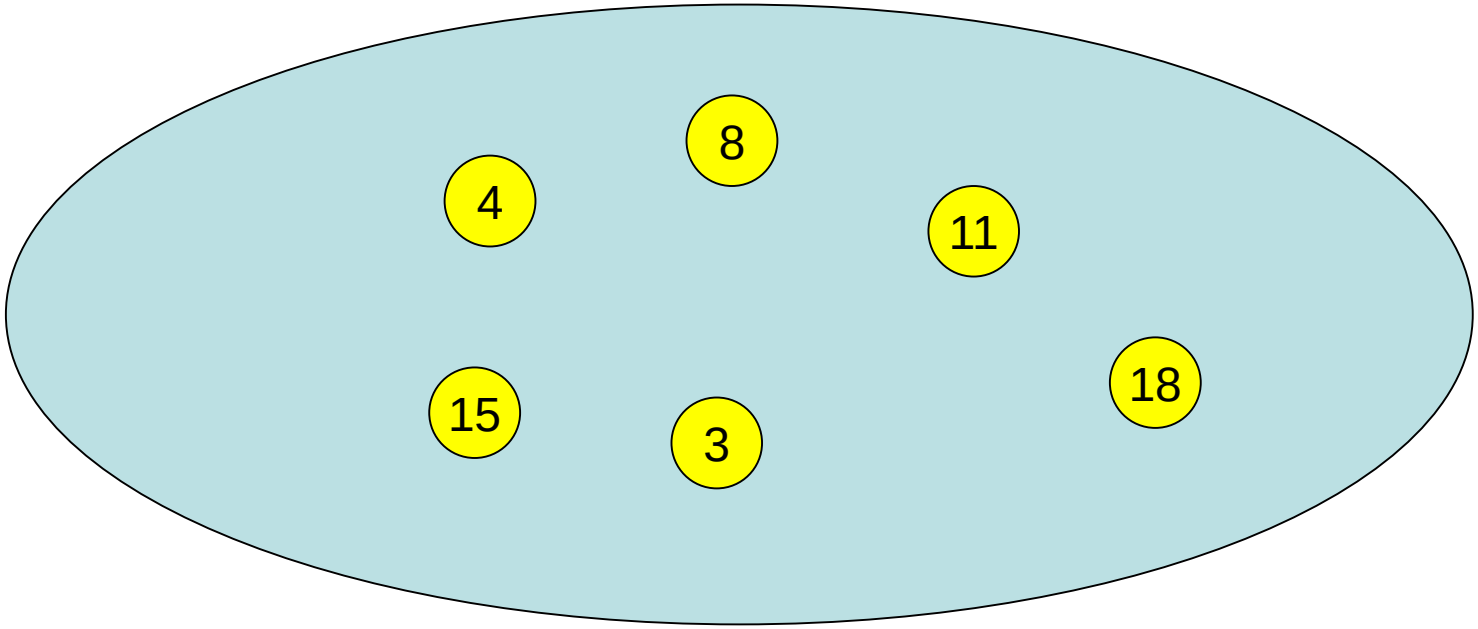
Dictionary

delete(20)



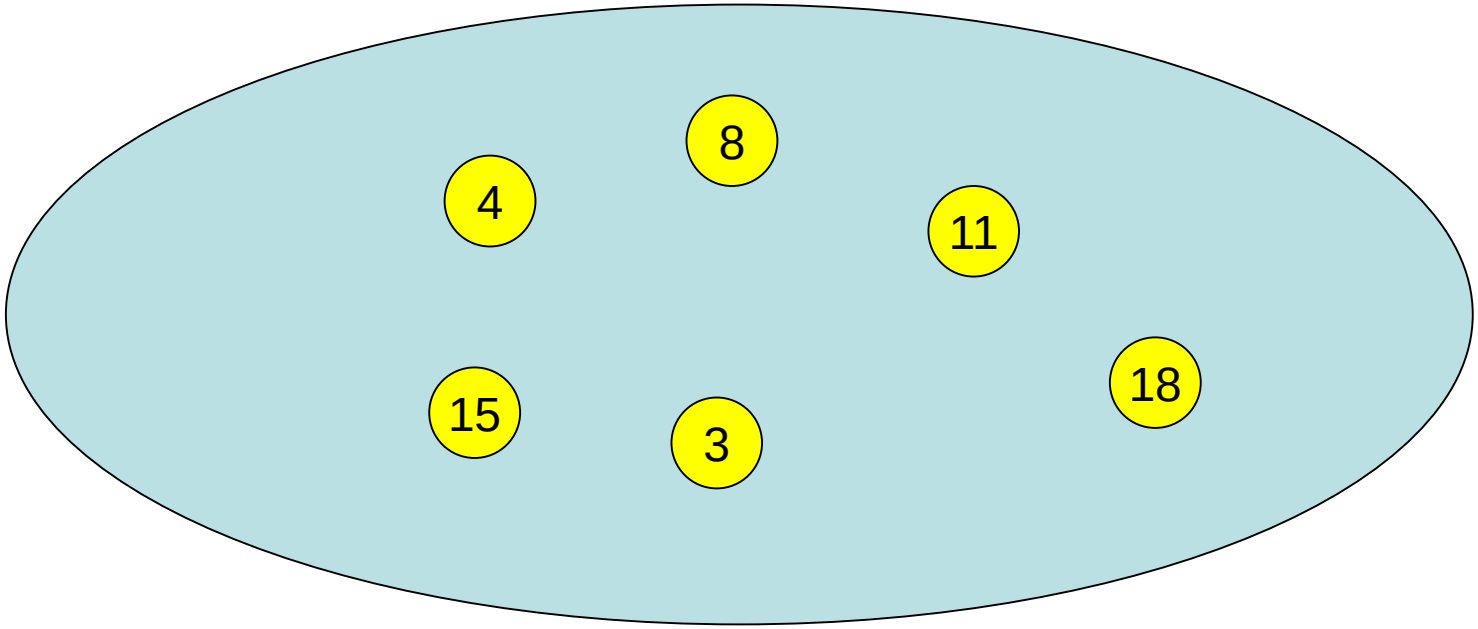
Dictionary

lookup(8) outputs 8



Dictionary

lookup(7) outputs $\perp$



Dictionary Data Structure

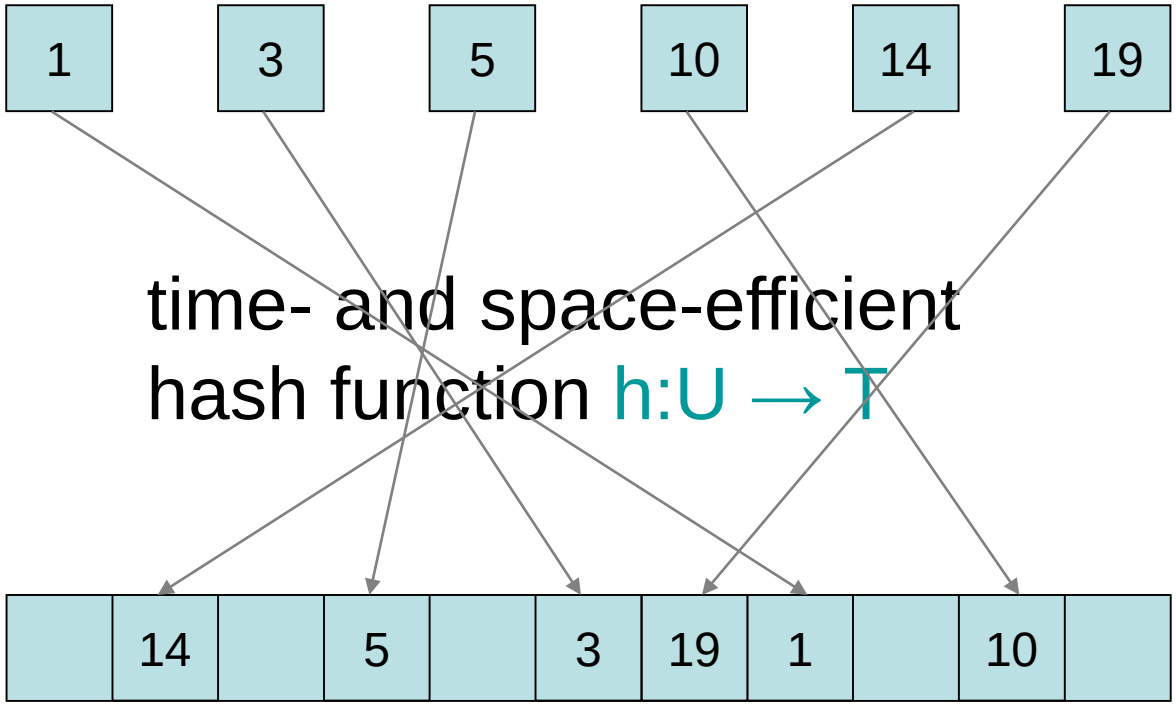
S: set of elements in data structure

Every element e identified by key(e).

Operations:

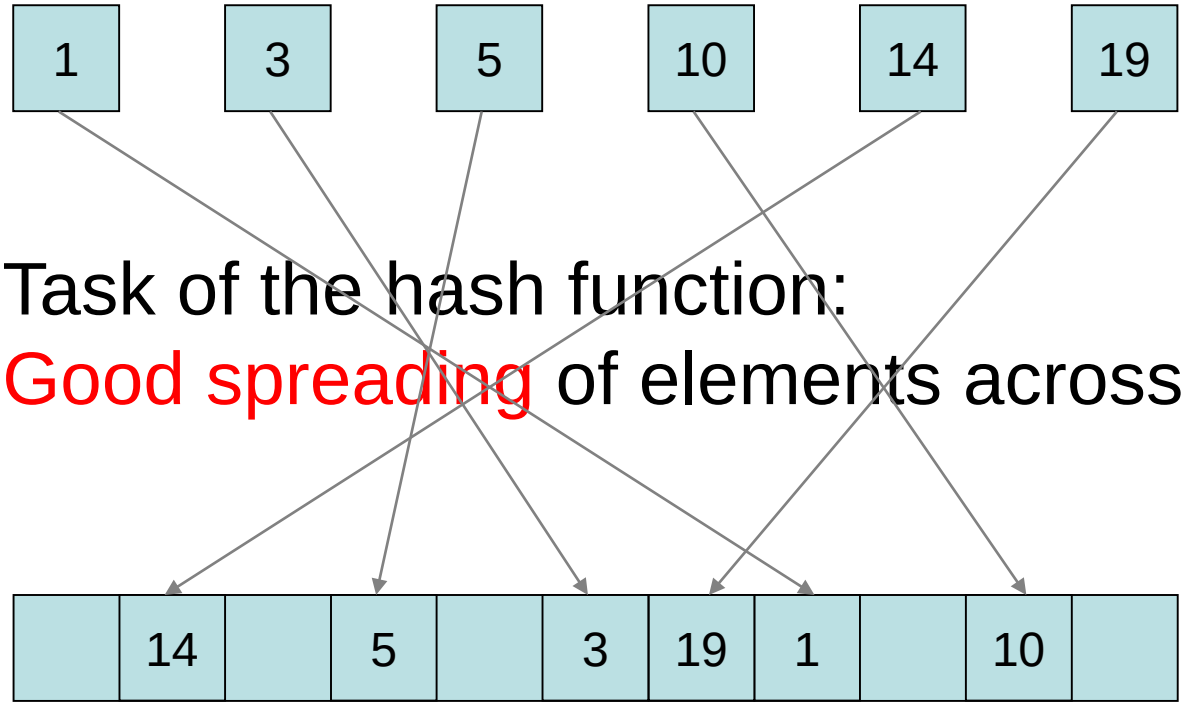
- S.insert(e: Element): $S:=S\cup\{e\}$
- S.delete(k: Key): $S:=S\setminus\{e\}$, where e is the element with key(e)=k
- S.lookup(k: Key): If there is an $e\in S$ with key(e)=k, output e, otherwise output $\perp$

Hashing



hash table T

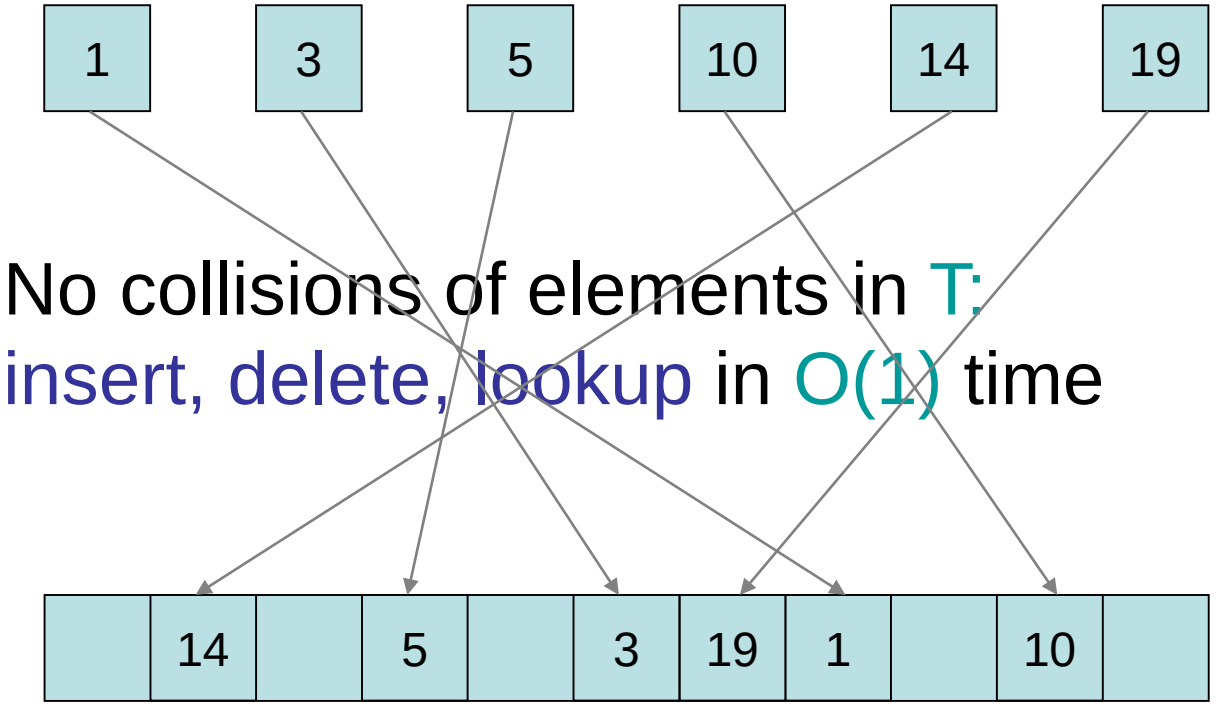
Hashing



Task of the hash function:
Good spreading of elements across T

hash table T

Hashing



No collisions of elements in T:
insert, delete, lookup in O(1) time

hash table T

Hashing (no collisions)

Procedure insert(e: Element)
  T[h(key(e))] := e

Procedure delete(k: Key)
  if key(T[h(k)])=k then T[h(k)] := ⊥

Function lookup(k: Key): Element∪{⊥}
  return T[h(k)]

Hashing

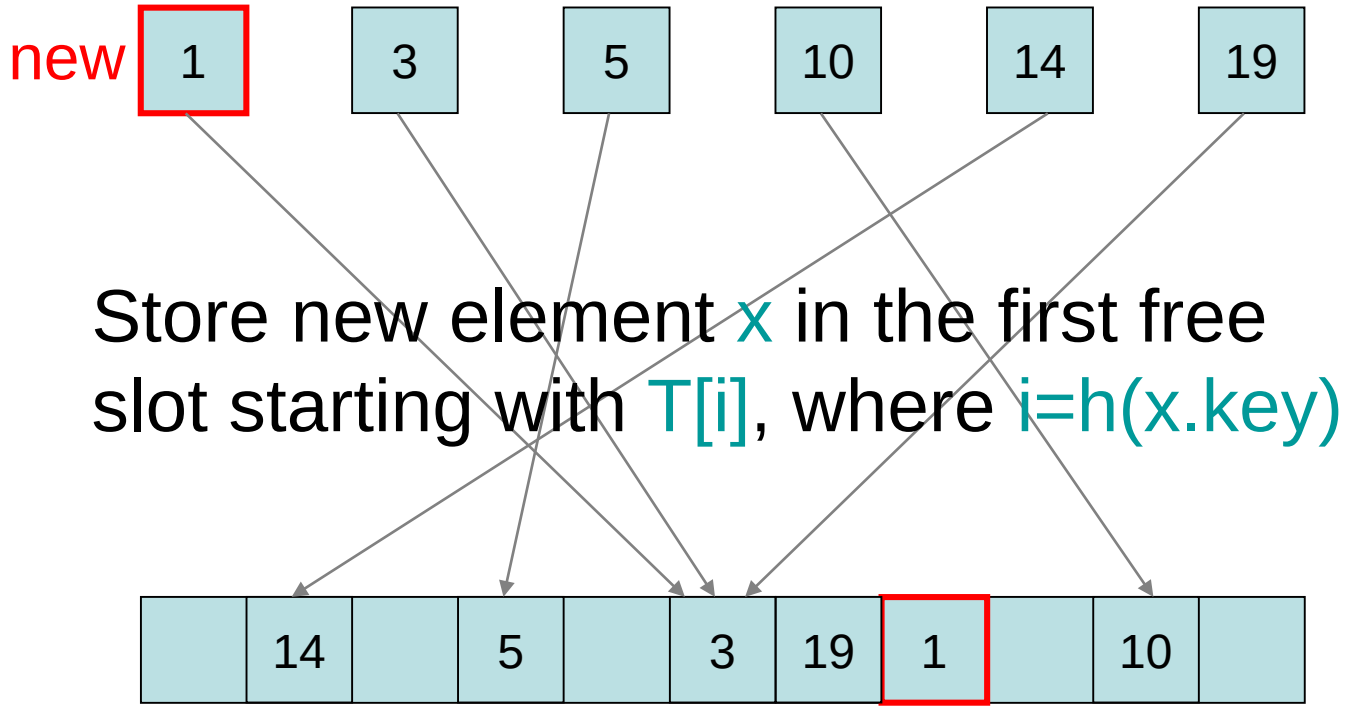
Problem: handling collisions

Solutions:

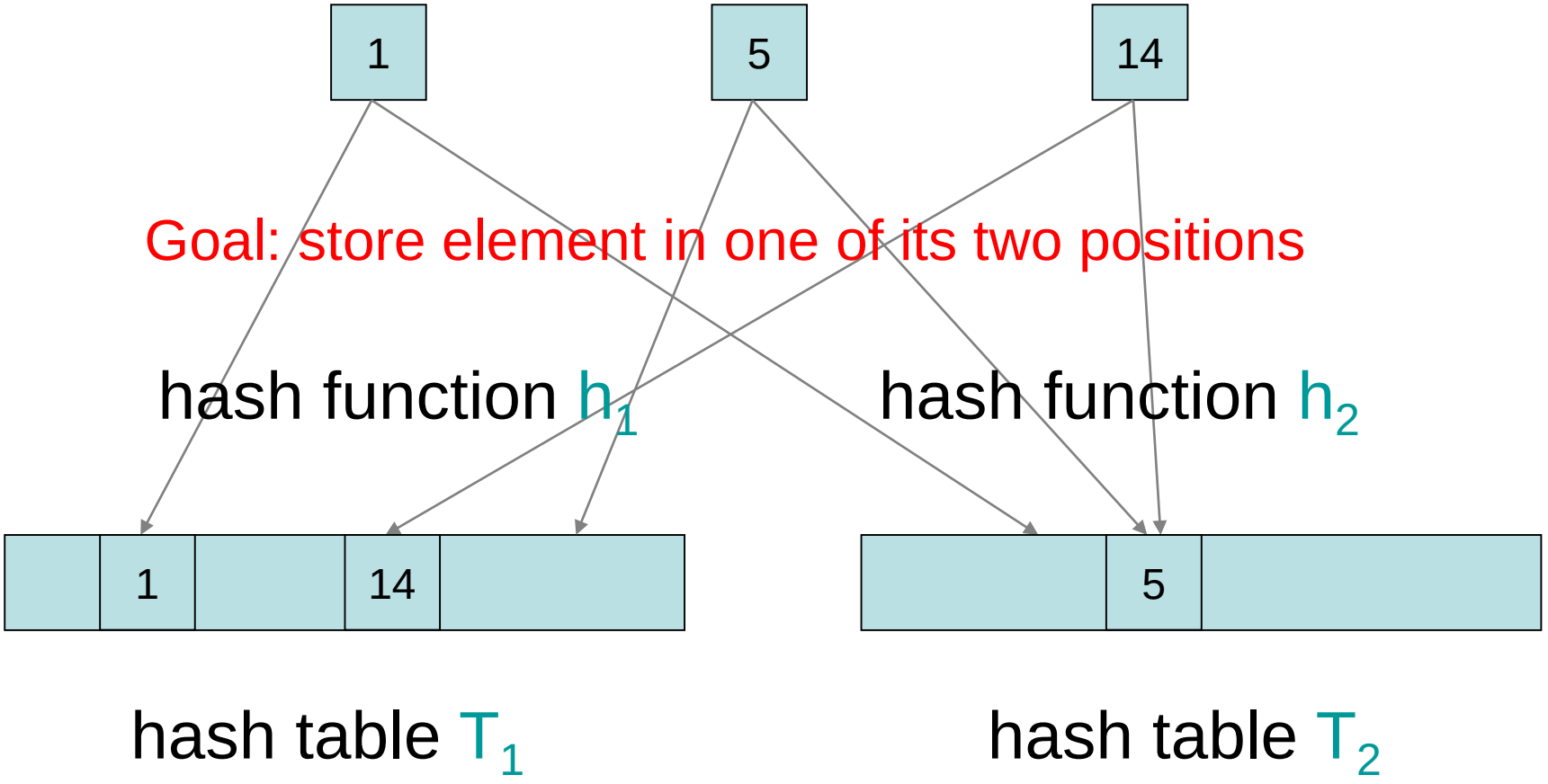
- Closed addressing:
  append elements in hash table entry to a list
- Open addressing:
  linear, quadratic probing
- Two hash tables:
  Cuckoo hashing

Open Addressing

Linear probing:



Cuckoo Hashing



Cuckoo Hashing

T_1, T_2: Array [0..m-1] of Element

Procedure lookup(k: Key): Element$\cup\{\perp\}$
  if key($T_1[h_1(k)]$)=k then return $T_1[h_1(k)]$
  if key($T_2[h_2(k)]$)=k then return $T_2[h_2(k)]$
  return $\perp$

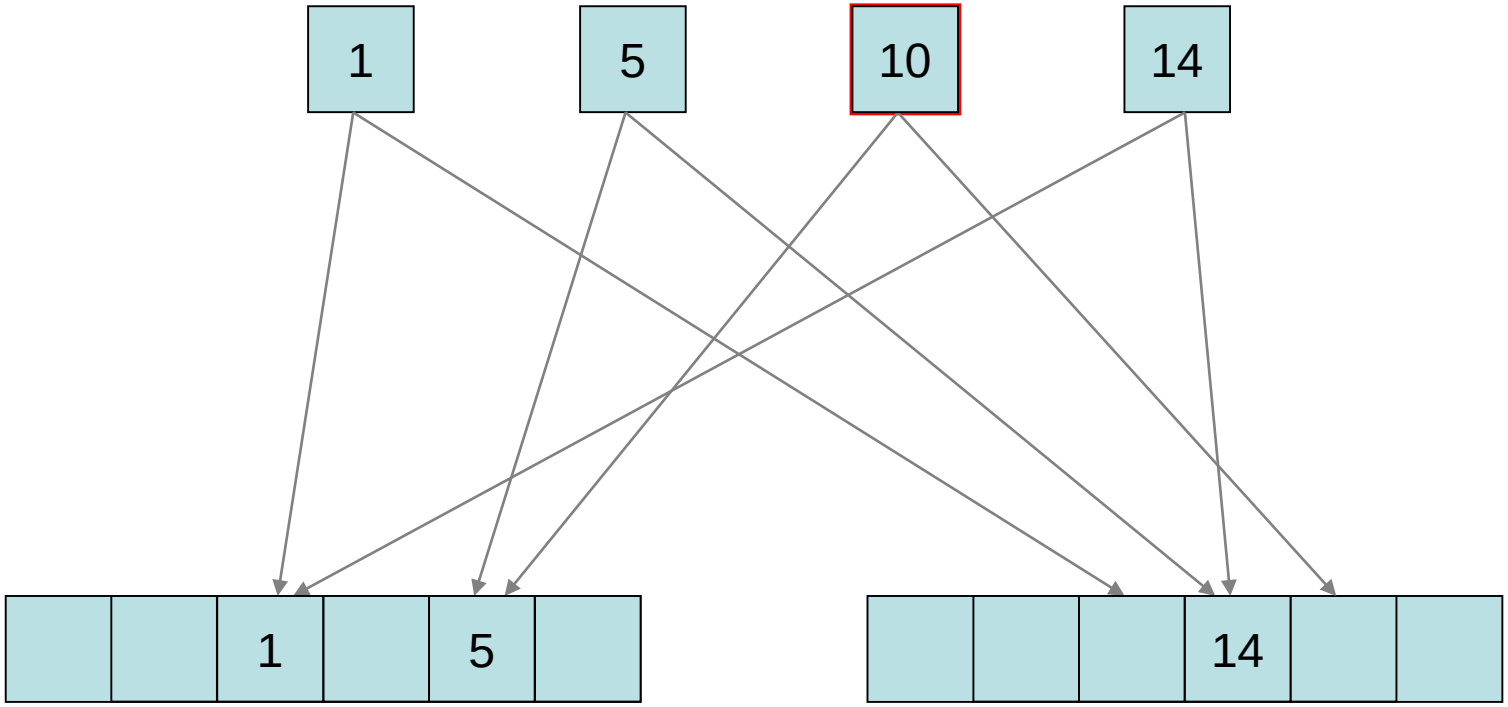
Procedure delete(k: Key)
  if key($T_1[h_1(k)]$)=k then $T_1[h_1(k)]:=\perp$
  if key($T_2[h_2(k)]$)=k then $T_2[h_2(k)]:=\perp$

Cuckoo Hashing

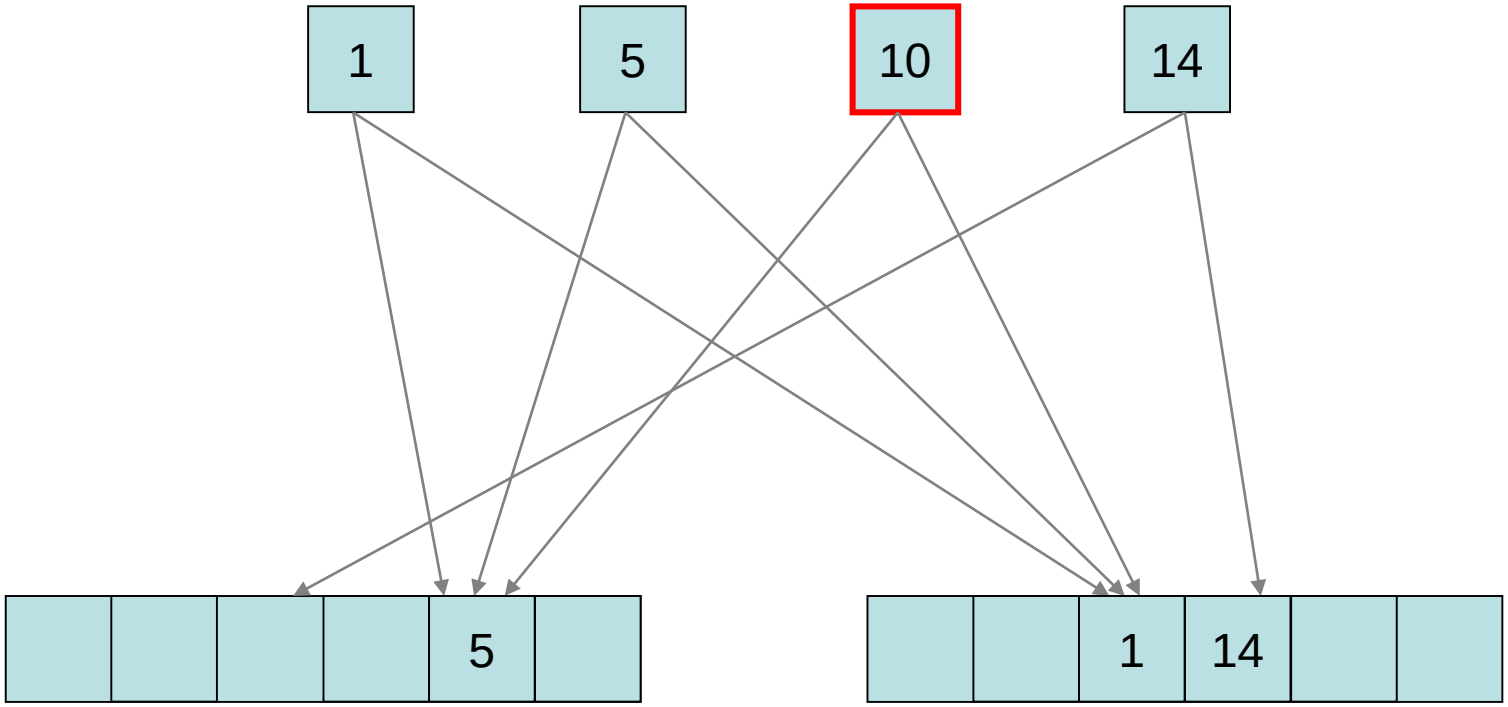
```
Procedure insert(e: Element)
    // key(e) already in hash table?
    if key(T1[h1(key(e))])=key(e) then
        T1[h1(key(e))] := e; return
    if key(T2[h2(key(e))])=key(e) then
        T2[h2(key(e))] := e; return
    // no, then insert it
    while true do
        e ↔ T1[h1(key(e))]  // switch e with element in T1
        if e=⊥ then return
        e ↔ T2[h2(key(e))]  // switch e with element in T2
        if e=⊥ then return
```

Better: at most d log n often, where the constant d is „sufficiently large“. If more than d log n rounds are needed, then rehash all elements with new h_1, h_2

Cuckoo Hashing



Cuckoo Hashing



Infinite loop!

Cuckoo Hashing

Runtime:

- lookup, delete: O(1) (worst case!)
- insert: O(number of replacements + possibly the time for a complete rehash)

Runtime of insert operation: 2 cases

1. after t rounds, insert enters an infinite loop, which requires rehash
2. insert terminates after t rounds

Nevertheless, expected runtime of insert: O(1) if $m \geq 2n$, where m is the hash table size.

Next Lecture

Chapter 4 (dynamic programming and Greedy algorithms) skipped this time.

We continue with

Chapter 5 (Basic graph algorithms).