Data structures

- Organize your data to support various queries using little time an space

  Example: Inventory

  Want to support

  SEARCH

  INSERT

  DELETE

- Given n elements A[1..n]
- Support SEARCH(A,x) := is x in A?
- Trivial solution: scan A. Takes time $\Theta(n)$
- Best possible given A, x.
- What if we are first given A, are allowed to preprocess it, can we then answer SEARCH queries faster?
- How would you preprocess A?

- Given n elements A[1..n]

- Support SEARCH(A,x) := is x in A?

- Preprocess step: Sort A. Takes time O(n log n), Space O(n)

- SEARCH(A[1..n],x) := /* Binary search */

  If n = 1 then return YES if A[1] = x, and NO otherwise

  else

  if A[n/2] ≤ x then return SEARCH(A[n/2..n])

  else return SEARCH(A[1..n/2])

- Time T(n) = ?

- Given n elements A[1..n]

- Support SEARCH(A,x) := is x in A?

- Preprocess step: Sort A. Takes time O(n log n), Space O(n)

- SEARCH(A[1..n],x) := /* Binary search */

  If n = 1 then return YES if A[1] = x, and NO otherwise

  else

  if A[n/2] ≤ x then return SEARCH(A[n/2..n])

  else return SEARCH(A[1..n/2])

- Time T(n) = O(log n).

- Given n elements A[1..n] each ≤ k, can you do faster?
- Support SEARCH(A,x) := is x in A?
- DIRECT ADDRESS:
- Preprocess step:

  Initialize S[1..k] to 0
  For (i = 1 to n) S[A[i]] = 1
- T(n) = O(n), Space O(k)
- SEARCH(A,x) = ?

- Given n elements A[1..n] each ≤ k, can you do faster?
- Support SEARCH(A,x) := is x in A?
- DIRECT ADDRESS:
- Preprocess step:

  Initialize S[1..k] to 0
  For (i = 1 to n) S[A[i]] = 1
- $T(n) = O(n)$, Space $O(k)$
- SEARCH(A,x) = return S[x]
- $T(n) = O(1)$

- Dynamic problems:
- Want to support SEARCH, INSERT, DELETE
- Support SEARCH(A,x) := is x in A?
- If numbers are small, ≤ k

  Preprocess: Initialize S to 0.

  SEARCH(x) := return S[x]

  INSERT(x) := …??

  DELETE(x) := …??

- Dynamic problems:

- Want to support SEARCH, INSERT, DELETE

- Support SEARCH(A,x) := is x in A?

- If numbers are small, ≤ k
  Preprocess: Initialize S to 0.
  SEARCH(x) := return S[x]
  INSERT(x) := S[x] = 1
  DELETE(x) := S[x] = 0

- Time T(n) = O(1) per operation
- Space O(k)

- Dynamic problems:
- Want to support SEARCH, INSERT, DELETE
- Support SEARCH(A,x) := is x in A?
- What if numbers are not small?
- There exist a number of data structure that support each operation in O(log n) time
- Trees: AVL, 2-3, 2-3-4, B-trees, red-black, AA, ...
- Skip lists, deterministic skip lists,
- Let's see binary search trees first

Binary tree

Vertices, aka nodes = {a, b, c, d, e, f, g, h, i}

Root = a

Left subtree = {c}

Right subtree ={b, d, e, f, g, h, i}

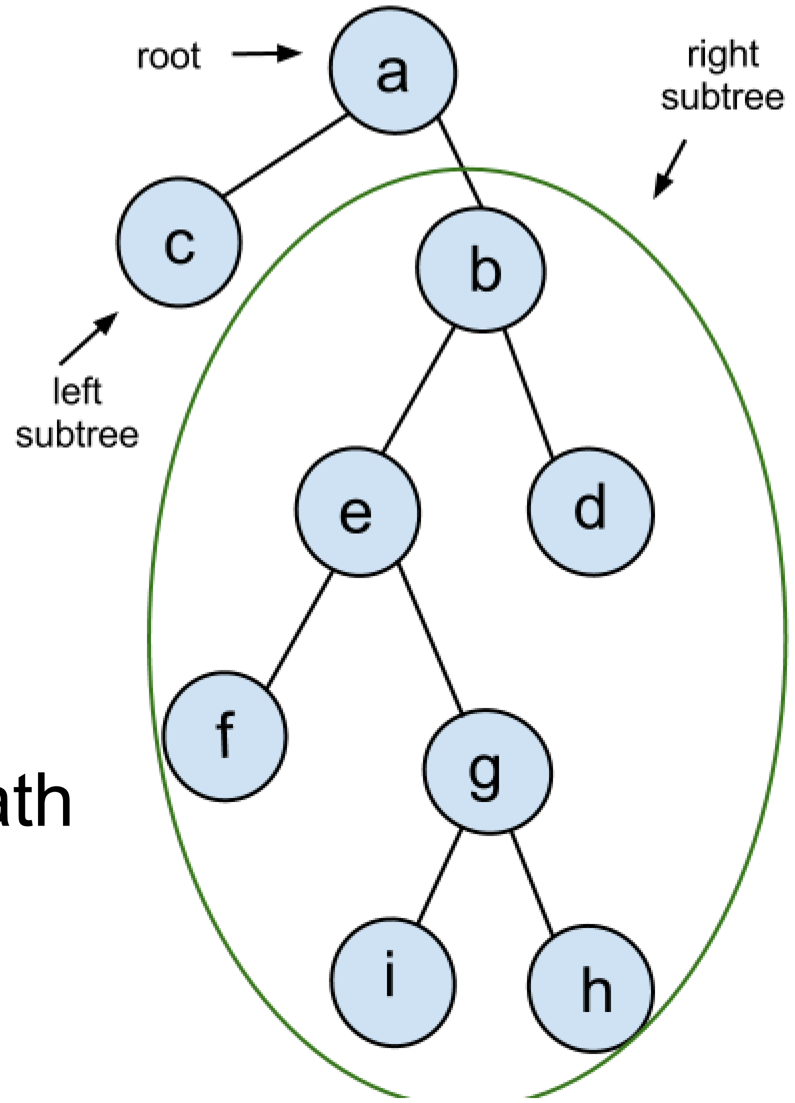
Parent(b) = a

Leaves = nodes with no children

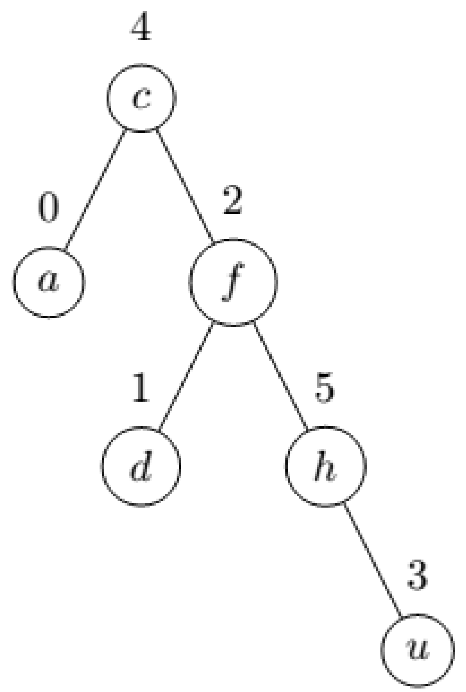
= {c, f, i, h, d}

Depth = length of longest root-leaf path

= 4



How to represent a binary tree using arrays



Index	0	1	2	3	4	5
Key	a	d	f	u	c	h
Parent	4	2	4	5	NULL	2
LeftChild	NULL	NULL	1	NULL	0	NULL
RightChild	NULL	NULL	5	NULL	2	3

Root = 4

NumNodes = 6

Binary Search Tree is a data structure where we store data in nodes of a binary tree and refer to them as key of that node.

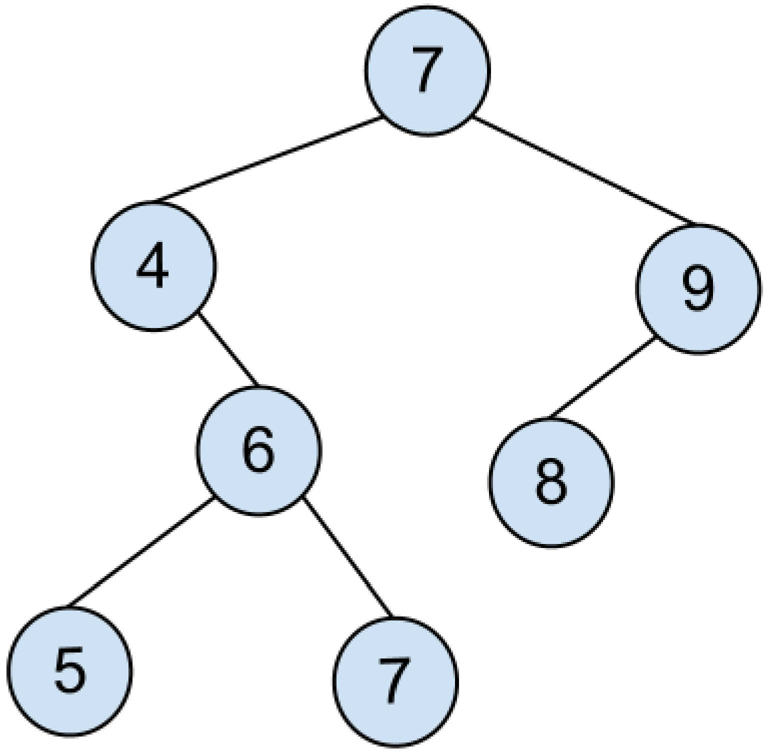
The keys in a binary search tree satisfy the

binary search tree property:

Let $x,y \in V$, if y is in left subtree of x $\Longrightarrow$ $key(y) \leq key(x)$

if y is in right subtree of y $\Longrightarrow$ $key(x) < key(y)$.

Example:



Tree-search(x,k) \\ Looks for k in binary search tree rooted at x

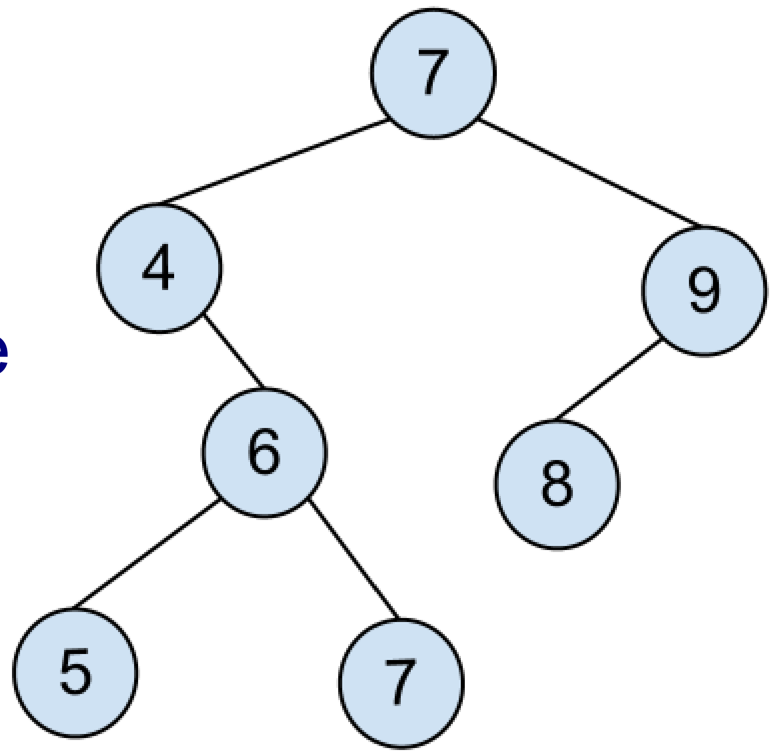
if x = NULL or k = Key[x]

return x

if k ≤ key[x]

return Tree-search(LeftChild[x],k) else

return tree-search(RightChild[x],k)



Running time = O(Depth)

Depth = O(log n) ⇨ search time O(log n)

Tree-Search is a generalization of binary search
in an array that we saw before.

A sorted array can be thought of as a balanced tree
(we'll return to this)

Trees make it easier to think about inserting and removing

Insert(k) // Inserts k

If the tree is empty

Create a root with key k and return

Let y be the last node visited during Tree-Search(Root,k)

If k ≤ Key[y]

Insert new node with key k as left child of y

If k > Key[y]

Insert new node with key k as right child of y

Running time = O(Depth)

Depth = O(log n) ⇨ insert time O(log n)

Let us see the code in more detail

```
Insert(k):
//If there is no room, do nothing
if NumNodes >= MAXNODES
  return
//Otherwise puts a new node at the end of the arrays
Key[NumNodes] ← k
LeftChild[NumNodes] ← NULL
RightChild[NumNodes] ← NULL
//It remains to determine the parent
//If tree is empty then there is none
if NumNodes = 0
  Root ← 0
  Parent[NumNodes] ← NULL
  NumNodes++
  return
//Otherwise looks for the parent in the tree
x ← Root
forever
  //two ifs to check if x is parent, otherwise search
  if k ≤ Key[x] and LeftChild[x] = NULL
    LeftChild[x] ← NumNodes
    Parent[NumNodes] ← x
    NumNodes++
    return
  if k > Key[x] and RightChild[x] = NULL
    RightChild[x] ← NumNodes
    Parent[NumNodes] ← x
    NumNodes++
    return
  if k ≤ Key[x] and LeftChild[x] ≠ NULL
    x ← LeftChild[x]
  if k > Key[x] and RightChild[x] ≠ NULL
    x ← RightChild[x]
```

Goal: SEARCH, INSERT, DELETE in time O(log n)

We need to keep the depth to O(log n)

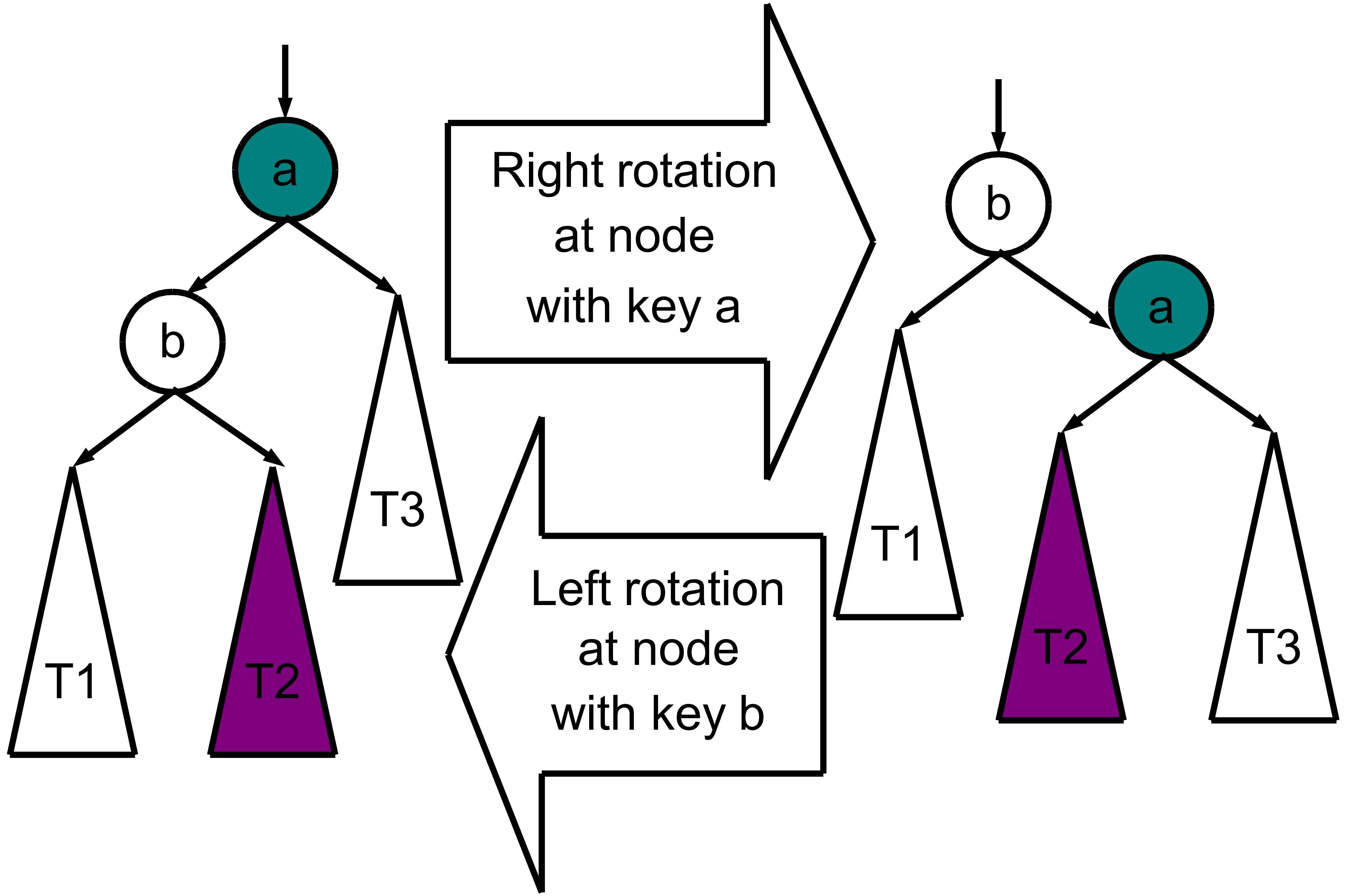
When inserting and deleting, the depth may change.

Must restructure the tree to keep depth O(log n)

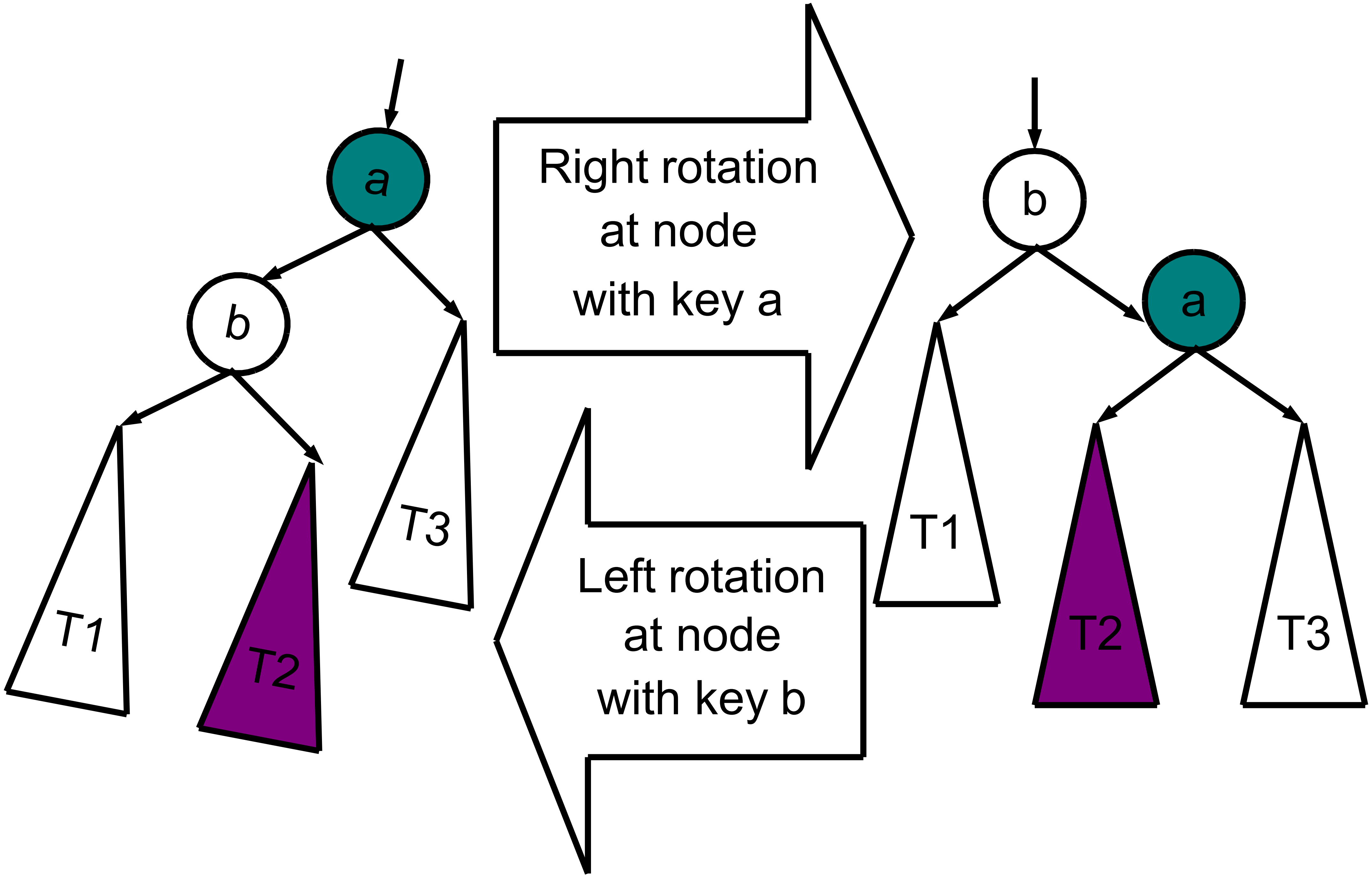
A basic restructing operation is a rotation

Rotation is then used by more complicated operations

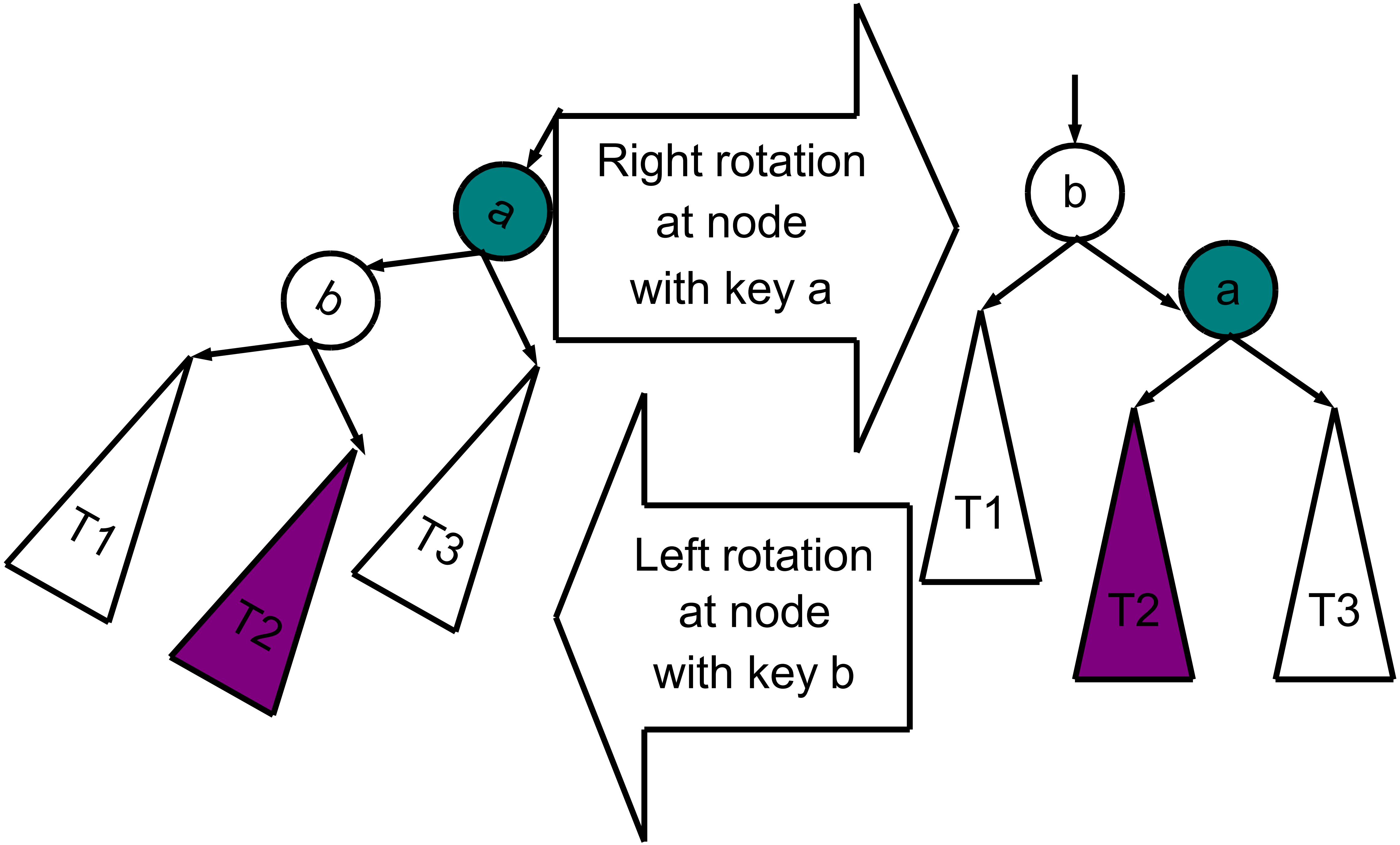
Tree rotations



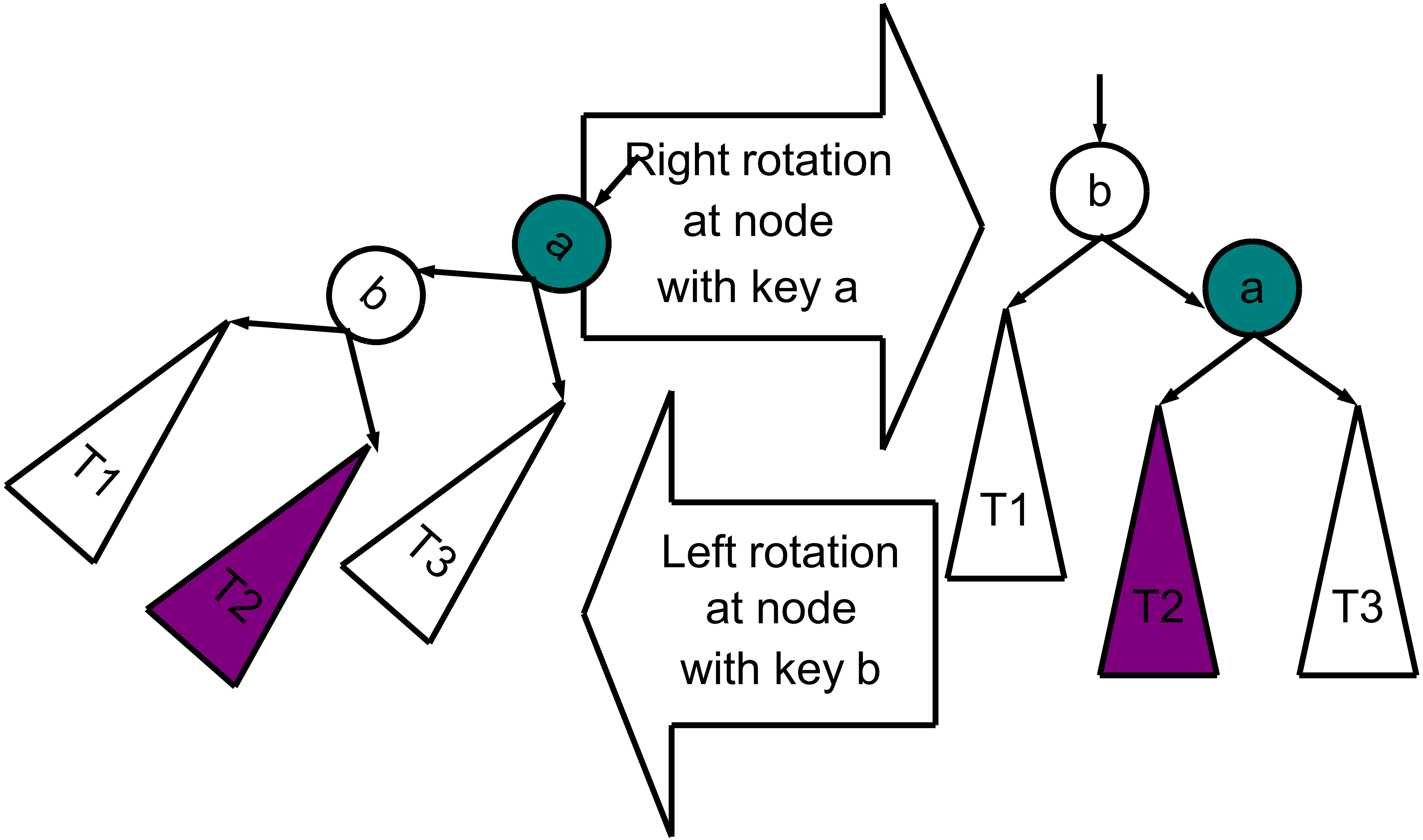
Tree rotations



Tree rotations



Tree rotations

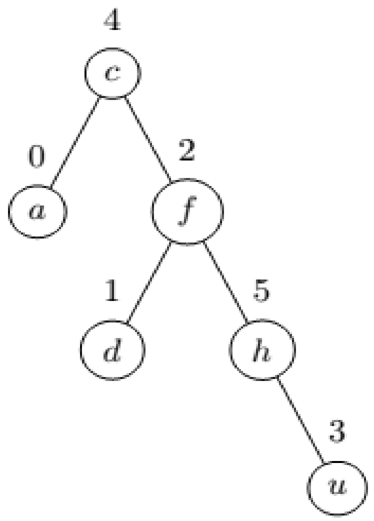


Tree rotations, code using our representations

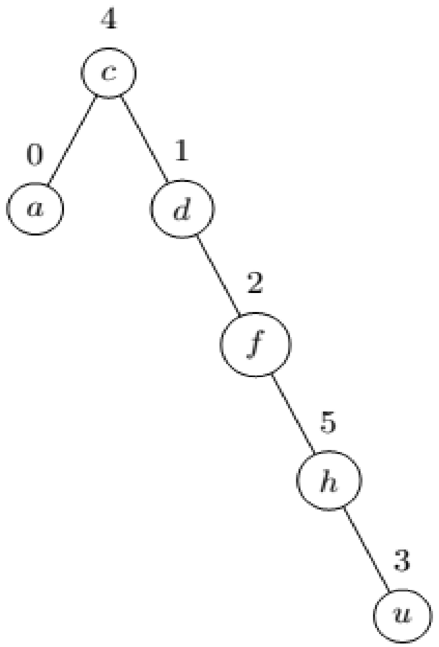
```
Rotate-Right(i):

if i does not have a left child
  return
L ← LeftChild[i]
if i is the root
  Root ← L, Parent[L] ← NULL
If i is a right child of P
  RightChild[P] ← L, Parent[L] ← P
If i is a left child of P
  LeftChild[P] ← L, Parent[L] ← P
LR ← RightChild[L]
RightChild[L] ← i
Parent[i] ← L
LeftChild[i] ← LR
If LR ≠ NULL
  Parent[LR] ← i
```

Tree rotations, code using our representations



Index	0	1	2	3	4	5
Key	a	d	f	u	c	h
Parent	4	2	4	5	NULL	2
LeftChild	NULL	NULL	1	NULL	0	NULL
RightChild	NULL	NULL	5	NULL	2	3



Index	0	1	2	3	4	5
Key	a	d	f	u	c	h
Parent	4	4	1	5	NULL	2
LeftChild	NULL	NULL	NULL	NULL	0	NULL
RightChild	NULL	2	5	NULL	1	3

Using rotations to keep the depth small

- AVL trees: binary trees. In any node, heights of children differ by ≤ 1. Maintain by rotations
- 2-3-4 trees: nodes have 1,2, or 3 keys and 2, 3, or 4 children. All leaves same level. To insert in a leaf: add a child. If already 4 children, split the node into one with 2 children and one with 4, add a child to the parent recursively. When splitting the root, create new root.

  Deletion is more complicated.
- B-trees: a generalization of 2-3-4 trees where can have more children. Useful in some disk applications where loading a node corresponds to reading a chunk from disk
- Red-black trees: A way to “simulate” 2-3-4 trees by a binary tree. E.g. split 2 keys in same 2-3-4 node into 2 red-black nodes. Color edges red or black depending on whether the child comes from this splitting or not, i.e., is a child in the 2-3-4 tree or not.

●AVL trees: binary tre ights of children
differ by ≤ 1. Mai

●2-3-4 trees 4
children. a
child. If 2
childre vely.
When

Dele

●B-tre
more c
loading sk

●Red-blac a
binary tree. E to 2 red-
black nodes. Co ing on
whether the child co g or not, i.e., is a
child in the 2-3-4 tree or not.

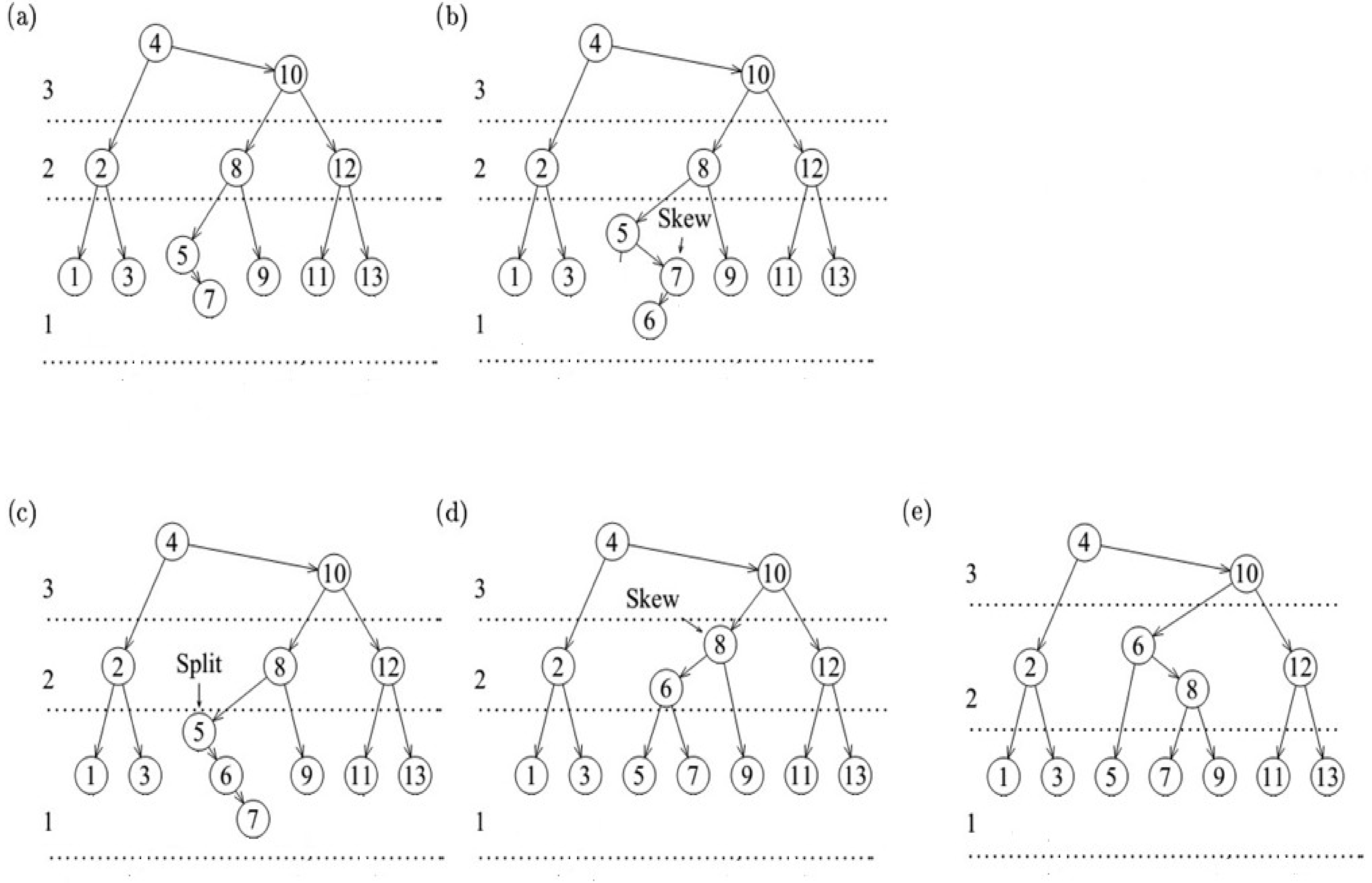
We see in detail what may be the simplest variant of these:

AA Trees

First we see pictures,

then formalize it,

then go back to pictures.


(a)
4
10
3
2
8
12
5
1
3
9
11
13
7
1
(b)
4
10
3
2
8
12
5
Skew
1
3
7
9
11
13
6
1
(c)
4
10
3
2
Split
8
12
5
1
3
6
9
11
13
7
1
(d)
4
10
3
Skew
8
2
6
12
1
3
5
7
9
11
13
1
(e)
4
10
3
6
2
8
12
2
1
3
5
7
9
11
13
1

- Definition: An AA Tree is a binary search tree where each node has a level, satisfying:

(1) The level of every leaf node is one.

(2)The level of every left child is exactly one less than that of its parent.

(3)The level of every right child is equal to or one less than that of its parent.

(4)The level of every right grandchild is strictly less than that of is grandparent.

(5) Every node of level greater than one has two children.

- Intuition: “the only path with nodes of the same level is a single left-right edge”

- Fact: An AA Tree with n nodes has depth O(log n)

- Proof:

Suppose the tree has depth d.

The level of the root is at least d/2.

Since every node of level > 1 has two children, the tree contains a full binary tree of depth at least d/2-1. Such a tree has at least $2^{d/2-1}$ nodes.

□

- Restructuring an AA tree after an addition:

- Rule of thumb:

First make sure that only left-right edges are within nodes of the same level (Skew)

then worry about length of paths within same level (Split)

Restructuring operations:

Skew(x): If x has left-child with same level

RotateRight(x)

Split(x): If the level of the right child of the right child of x

is the same as the level of x,

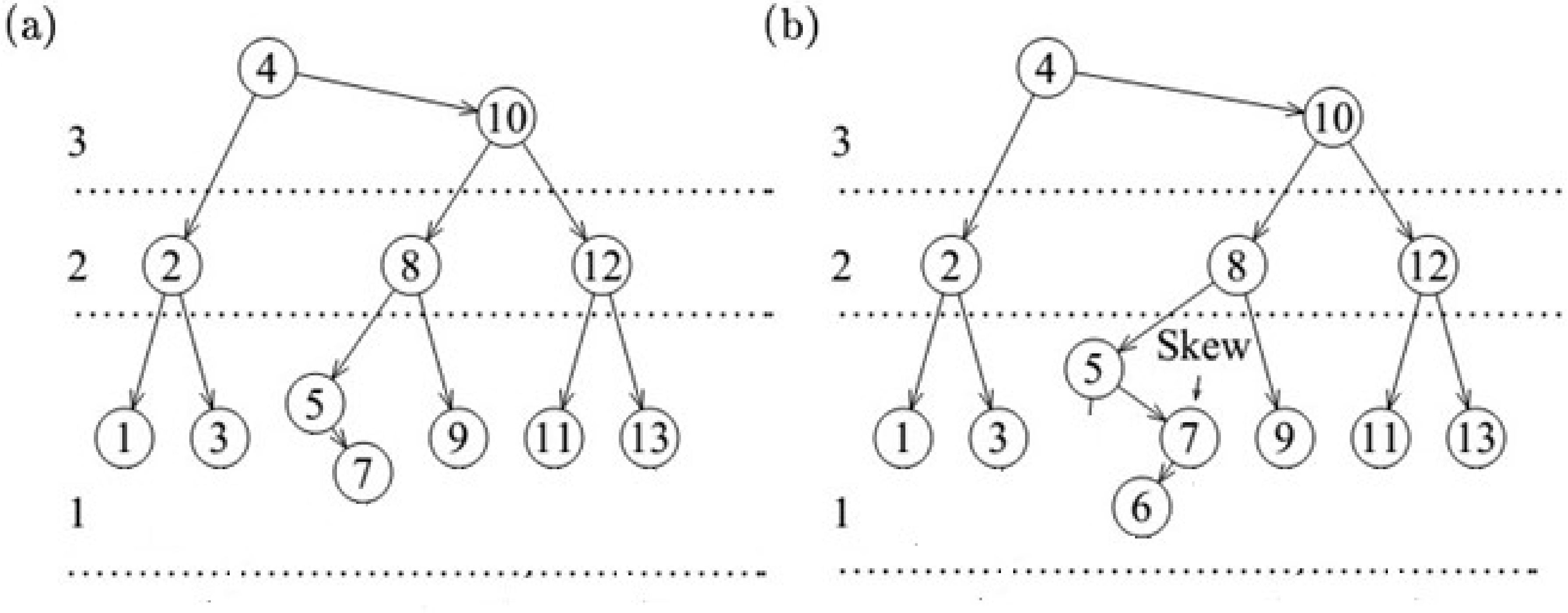
Level[RightChild[x]]++;

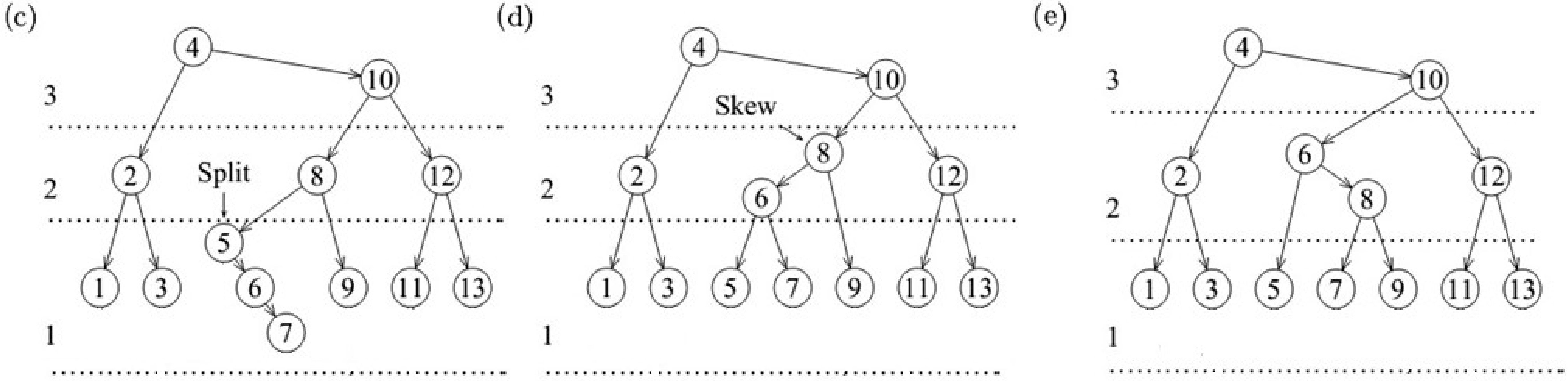
RotateLeft(x)

AA-Insert(k):

```
Insert k as in a binary search tree
/* For every node from new one back to root,
    do skew and split
*/
x ← NumNodes-1   //New node is last in array
while x ≠ NULL
  Skew(x)
  Split(x)
  x ← Parent[x]
```

Inserting 6





Deleting in an AA tree:

Decrease Level(x):

If one of x's children is two levels below x,

decrese the level of x by one.

If the right child of x had the same level of x, decrease the level of the right child of x by one too.

Delete(x): Suppose x is a leaf

Delete x.

Follow the path from x to the root and at each node y do:

Decrease level(y).

Skew(y); Skew(y.right); Skew(y.right.right);

Split(y); Split(y.right);

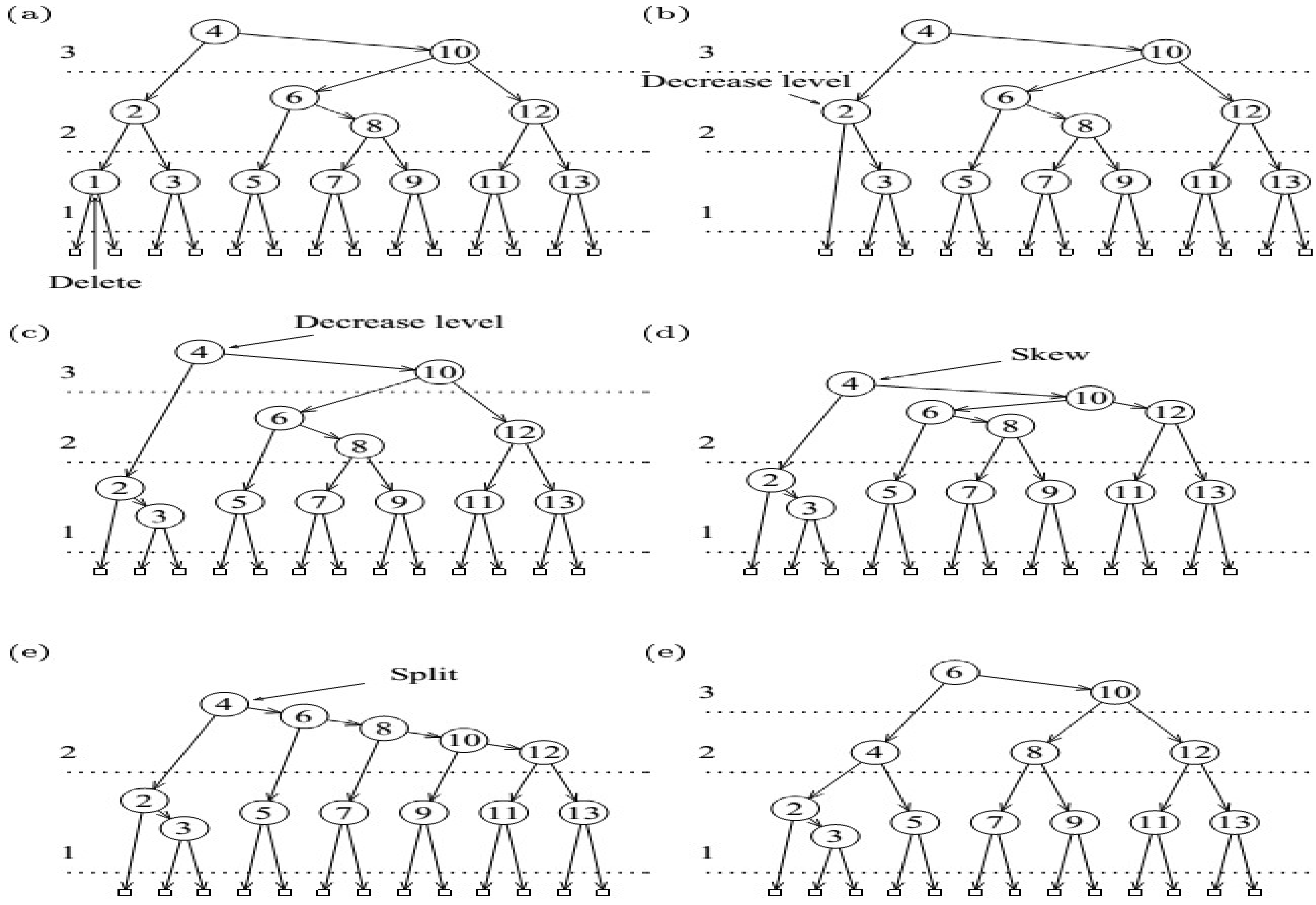


Fig. 2. *Example of deletion.*

Rotate right 10,
get 8 ← 10,
so again
rotate right 10

Note: The way to think of restructuring is that you work at a node. You call all these skew and split operations from that node. As an effect of these operations, the node you are working at may move. For example, in figure (e) before, when you work at node 4, you do a split. This moves the node. Then you are done with 4. While before node 4 was a root, now it's not a root anymore. So you'll move to its parent, which is 6. We now have an intermediate tree which isn't shown in the slide. We call the skews and we don't do anything. We call split at 6 and don't do anything. Now we finally call split at the right child of 6. This sees the path 8 -> 10 -> 12 in the same level, and fixes it to obtain the last tree.

Delete(x):

If x is a not a leaf, find the smallest leaf bigger than x.key, swap it with x, and remove that leaf.

To find that leaf, just perform search, and when you hit x go, for example, right.

It's the same thing as searching for x.key + ε

So swapping these two won't destroy the tree properties

Remark about memory implementation:

Could use new/malloc free/dispose to add/remove nodes.

However, this may cause memory segmentation.

It is possible to implement any tree using an array A so that:

at any point in time, if n elements are in the tree, those will take elements A[1..n] in the array only.

To do this, when you remove node with index i in the array, swap A[i] and A[n]. Use parent's pointers to update.

Summary

Can support SEARCH, INSERT, DELETE in time O(log n) for arbitrary keys

Space: O(n). For ach key we need to store level and pointers.

Can we get rid of the pointers and achieve space n?

Surprisingly, this is possible:

Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees,
by Franceschini and Grossi

Hash functions

- We have seen how to support SEARCH, INSERT, and DELETE in time O(log n) and space O(n) for arbitrary keys

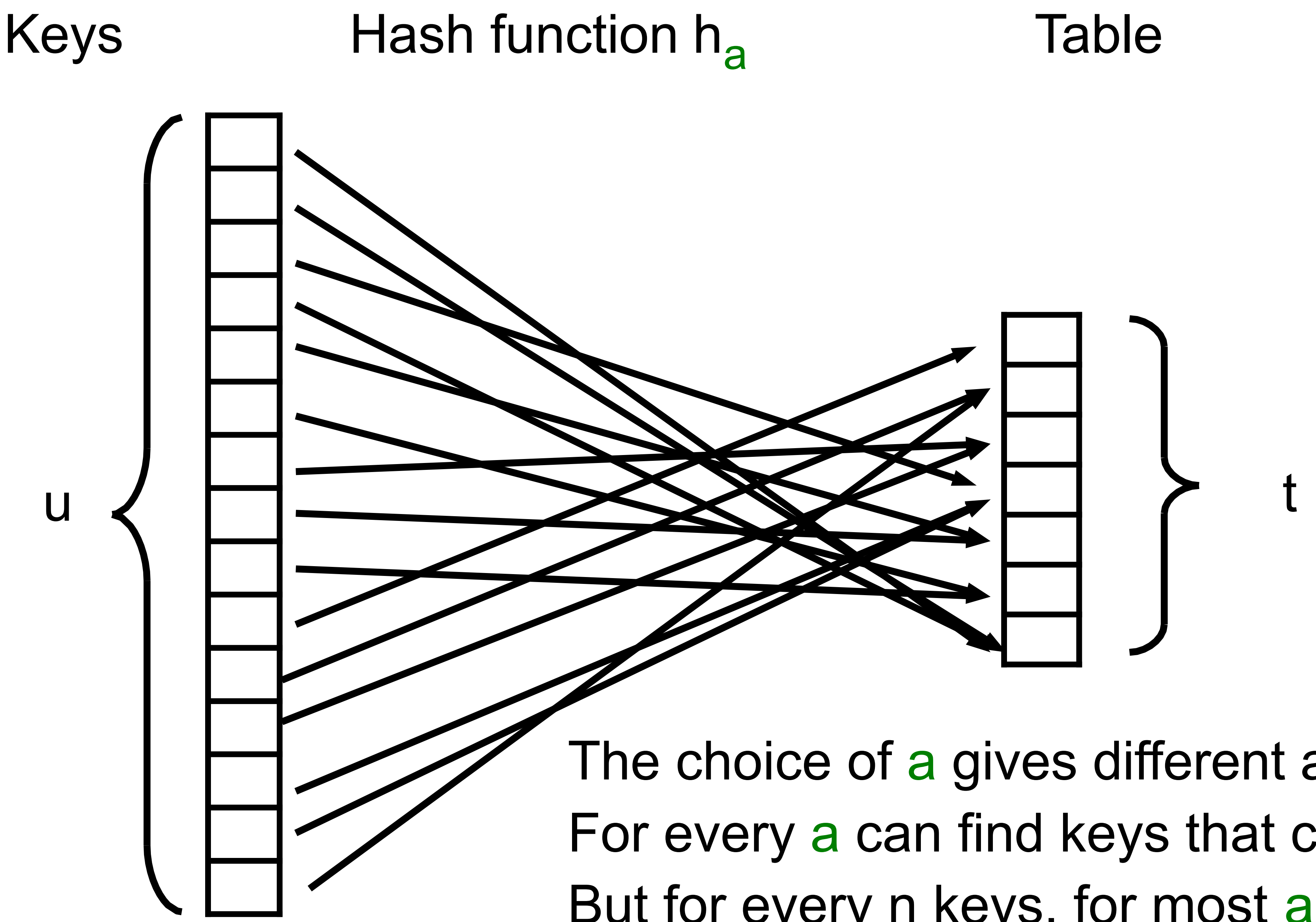
- If the keys are small integers, say in {1,2,...,t} for a small t we can do it in time ?? and space ??

- We have seen how to support SEARCH, INSERT, and DELETE in time O(log n) and space O(n) for arbitrary keys

- If the keys are small integers, say in {1,2,...,t} for a small t we can do it in time O(1) and space O(t)

- Can we have the same time for arbitrary keys?

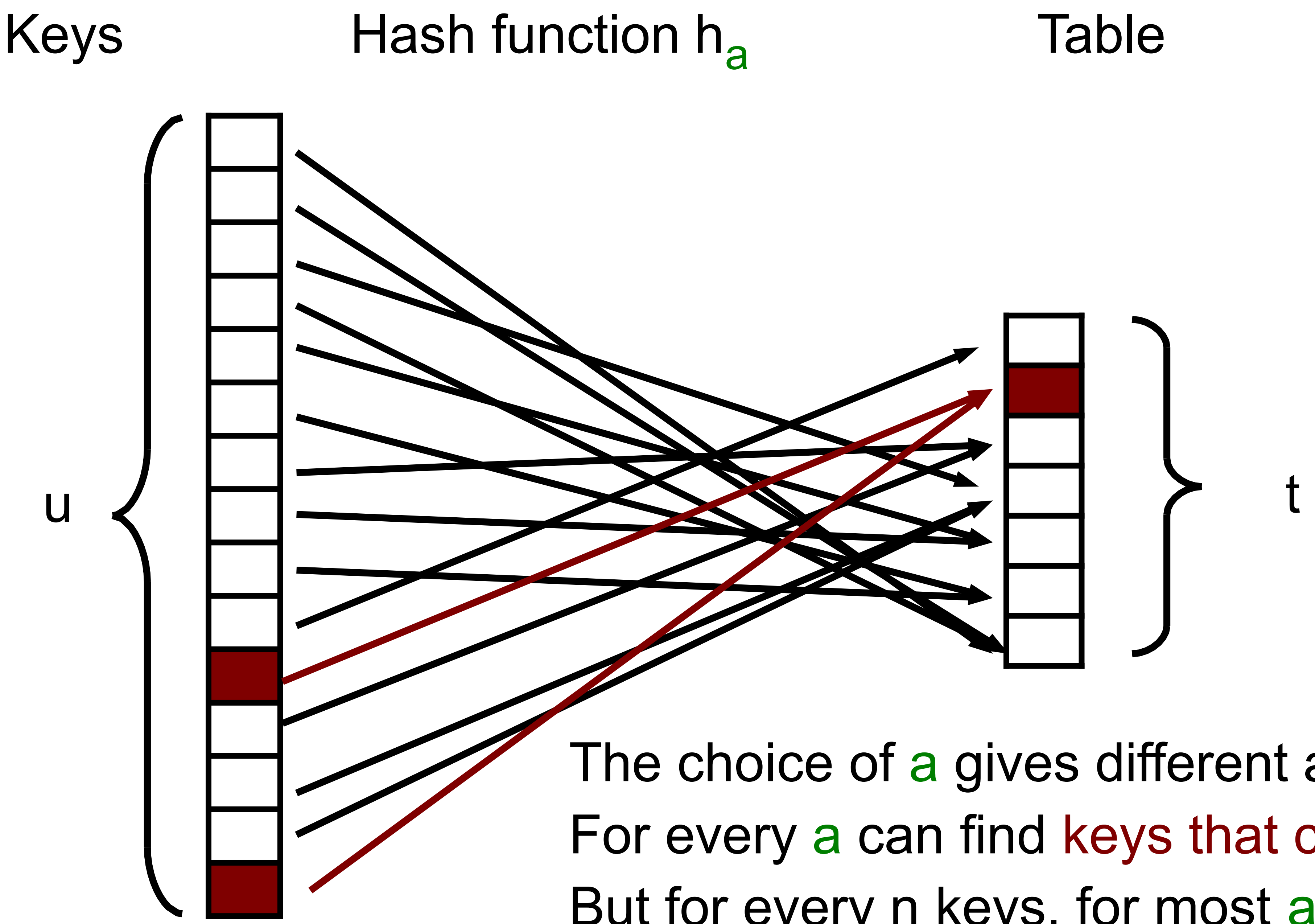
- Idea: Let's make the keys small.



The choice of a gives different arrows

For every a can find keys that collide

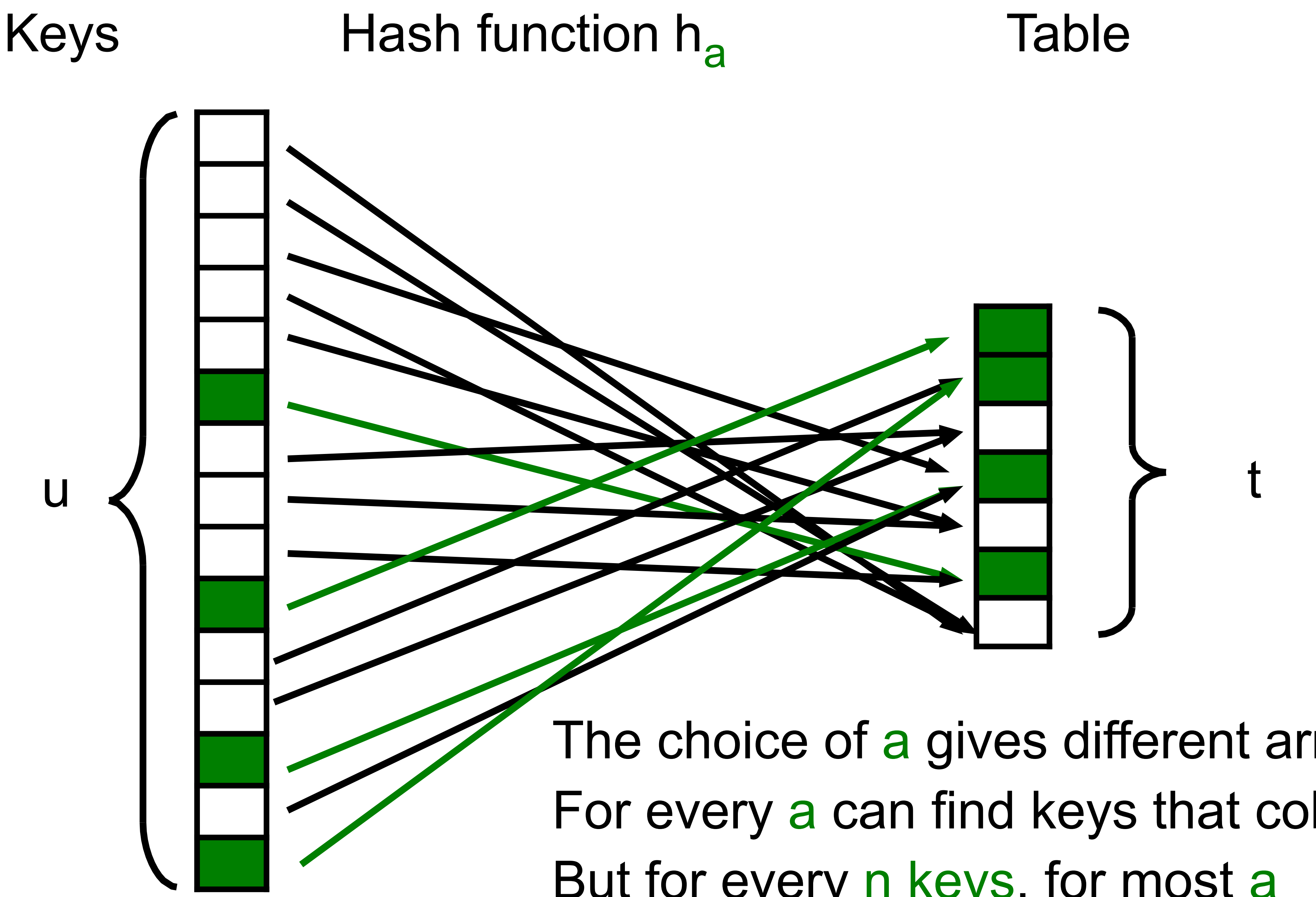
But for every n keys, for most a there are no collisions



The choice of a gives different arrows

For every a can find keys that collide

But for every n keys, for most a there are no collisions



The choice of a gives different arrows

For every a can find keys that collide

But for every n keys, for most a

there are no collisions

- Want to support INSERT, DELETE, SEARCH for n keys

  Keys come from large UNIVERSE = {1, 2, ..., u}

  We map UNIVERSE into a smaller set {1, 2, ..., t }

  using a **hash function** h : UNIVERSE → {1, 2, ..., t}

- We want that for each of our n keys, the values of h are different, so that we have no collisions

- In this case we can keep an array S[1..t] and

  SEARCH(x): ?

  INSERT(x): ?

  DELETE(x): ?

- Want to support INSERT, DELETE, SEARCH for n keys

  Keys come from large UNIVERSE = $\{1, 2, ..., u\}$

  We map UNIVERSE into a smaller set $\{1, 2, ..., t\}$

  using a **hash function** h : UNIVERSE $\to \{1, 2, ..., t\}$

- We want that for each of our n keys, the values of h are different, so that we have no collisions

- In this case we can keep an array $S[1..t]$ and

  SEARCH(x): return $S[h(x)]$

  INSERT(x): $S[h(x)] \leftarrow 1$

  DELETE(x): $S[h(x)] \leftarrow 0$

- Want to support INSERT, DELETE, SEARCH for n keys

  Keys come from large UNIVERSE = $\{1, 2, ..., u\}$

  We map UNIVERSE into a smaller set $\{1, 2, ..., t\}$

  using a **hash function** h : UNIVERSE $\rightarrow \{1, 2, ..., t\}$

- We want that for each of our n keys, the values of h are different, so that we have no collisions

- Example, think $n = 2^{10}$, $u = 2^{1000}$, $t = 2^{20}$

- Want to support INSERT, DELETE, SEARCH for n keys

  Keys come from large UNIVERSE = $\{1, 2, ..., u\}$

  We map UNIVERSE into a smaller set $\{1, 2, ..., t\}$

  using a **hash function** h : UNIVERSE $\to \{1, 2, ..., t\}$

- We want that for each of our n keys, the values of h are different, so that we have no collisions

- Can a fixed function h do the job?

- Want to support INSERT, DELETE, SEARCH for n keys

  Keys come from large UNIVERSE = {1, 2, ..., u}

  We map UNIVERSE into a smaller set {1, 2, ..., t }

  using a **hash function** h : UNIVERSE → {1, 2, ..., t}

- We want that for each of our n keys, the values of h are different, so that we have no collisions

- Can a fixed function h do the job?

  No, if h is fixed, then one can find two keys x ≠ y such that

  h(x)=h(y) whenever u > t

  So our function will use randomness.

  Also need compact representation so can actually use it.

- Construction of hash function:

  Let t be prime. Write a key x in base t:

  $x = x_1\, x_2\, \ldots\, x_m$ for $m = \log_t (u) = \log_2 (u)/\log_2 (t)$

  Hash function specified by seed element $a = a_1\, a_2 \ldots a_m$

$$h_a (x) := \sum_{i \leq m} x_i\, a_i \text{ modulo } t$$

- Example: t = 97, x = 171494

  $x_1 = 18,\ x_2 = 21,\ x_3 = 95$

  $a_1 = 45,\ a_2 = 18,\ a_3 = 7$

  $h_a (x) = 18{*}45 + 21{*}18 + 95{*}7 \bmod 97 = 10$

- Different constructions of hash function:
  Think of hashing s-bit keys to r bits

Classic solution: for a prime $p>2^s$, and a in [p],

$h_a(x) := ((ax) \bmod p) \bmod 2^r$

Problem: mod p is slow

Alternative: let b be a random odd s-bit number and

$h_b(x)$ = bits from s-r to s of integer product bx

Faster in practice. In C, think x unsigned integer of s=64 bits

$h_b(x) = (b*x) >> (u-r)$

- Analyzing hash functions

The function $h_a(x) := \sum_{i \leq m} x_i a_i$ modulo t satisfies

- 2-Hash Claim: $\forall x \neq x'$, $\Pr_a [h_a(x) = h_a(x')] = 1/t$

In other words, on any two fixed inputs, the function behaves like a completely random function

- n-hash Claim:

  Let h_a be a function from UNIVERSE to {1, 2, ..., t}

  Suppose h_a satisfies 2-hash claim

  If $t \geq 100\, n^2$ then for any n keys the probability that two have same hash is at most 1/100

- Proof: $Pr_a[\, \exists\, x \neq y : h_a(x) = h_a(y)\,]$

  $\leq \sum_{x,\, y\, :\, x \neq y} Pr_a[h_a(x) = h_a(y)]$ (union bound)

  $= \sum_{x,\, y\, :\, x \neq y}$?????

- n-hash Claim:

  Let h_a be a function from UNIVERSE to $\{1, 2, ..., t\}$

  Suppose h_a satisfies 2-hash claim

  If $t \geq 100\, n^2$ then for any n keys the probability that two have same hash is at most 1/100

- Proof: $Pr_a [\, \exists\, x \neq y : h_a(x) = h_a(y)\,]$

  $\leq \sum_{x,\, y\, :\, x \neq y} Pr_a [h_a(x) = h_a(y)]$ (union bound)

  $= \sum_{x,\, y\, :\, x \neq y} (1/t)$ (2-hash claim)

  $\leq n^2 (1/t) = 1/100$ □

- So, just make your table size $100n^2$ and you avoid collision
- Can you have no collisions with space $O(n)$?

- Theorem:

Given n keys, can support SEARCH in O(1) time and O(n) space

- Proof:

Two-level hashing:

(1) First hash to $t = O(n)$ elements,

(2) Then hash again using the previous method:

if i-th cell in first level has c_i elements, hash to c_i^2 cells

Expected total size $\leq E[\sum_{i \leq t} c_i^2]$

$= \Theta$(expected number of colliding pairs in first level) =

$= O(n^2 / t)$

$= O(n)$ □

- Trees vs. hashing

Trees maintain order: can be augmented to support other queries, like MIN, RANK

Hash functions are faster, but destroy order, and may fail with some small probability.

Queues and heaps

Queue

Operations: ENQUEUE, DEQUEUE

First-in-first-out

Simple, constant-time implementation using arrays:

A[0..n-1]

First $\leftarrow$ 0

Last $\leftarrow$ 0

ENQUEUE(x): If (Last < n), A[Last++] $\leftarrow$ x

DEQUEUE: If First < Last, return A[First++]

Priority queue

- Want to support
  INSERT
  EXTRACT-MIN

- Can do it using ??
  Time = ?? per query.
  Space = ??

Priority queue

- Want to support
  INSERT
  EXTRACT-MIN

- Can do it using AA trees.
  Time = O(log n) per query.
  Space = O(n).

- We now see a data structure that is simpler and somewhat more efficient.
  In particular, the space will be n rather than O(n)

A binary tree is complete if all the nodes have two children except the nodes in the last level.

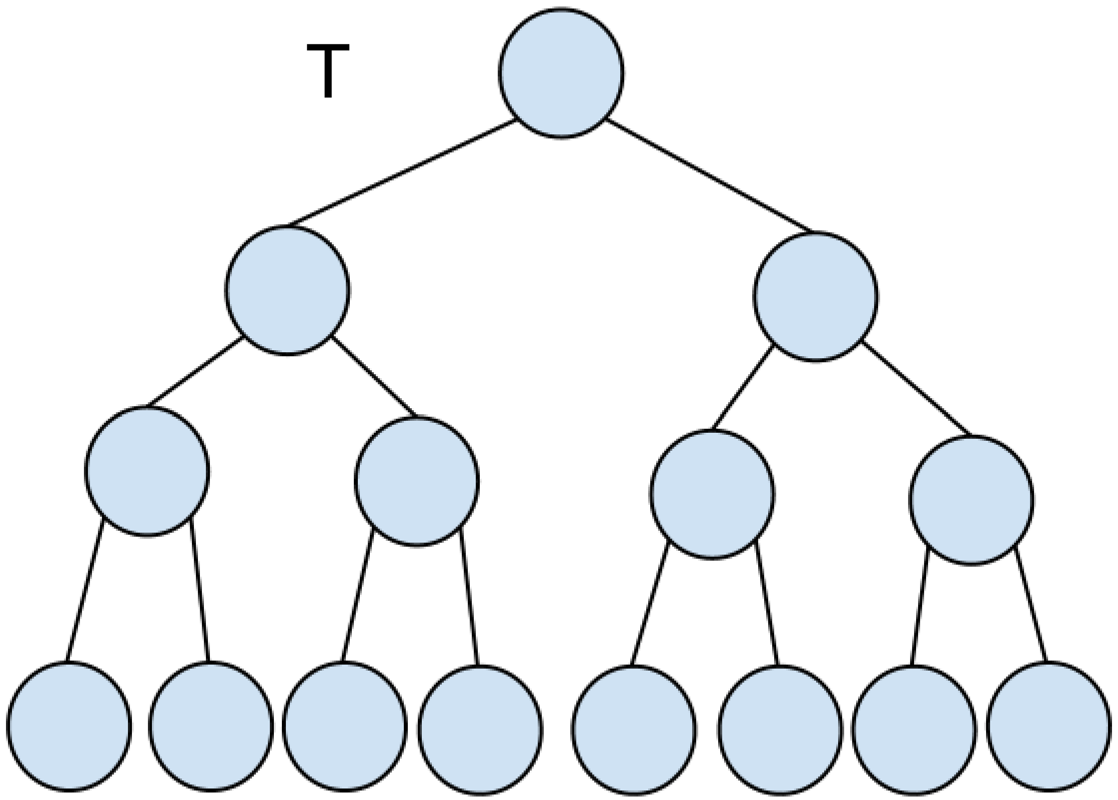
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

Example:

Depth of T=?

Number of leaves in T=?

Number of nodes in T=?



A binary tree is complete if all the nodes have two children except the nodes in the last level.

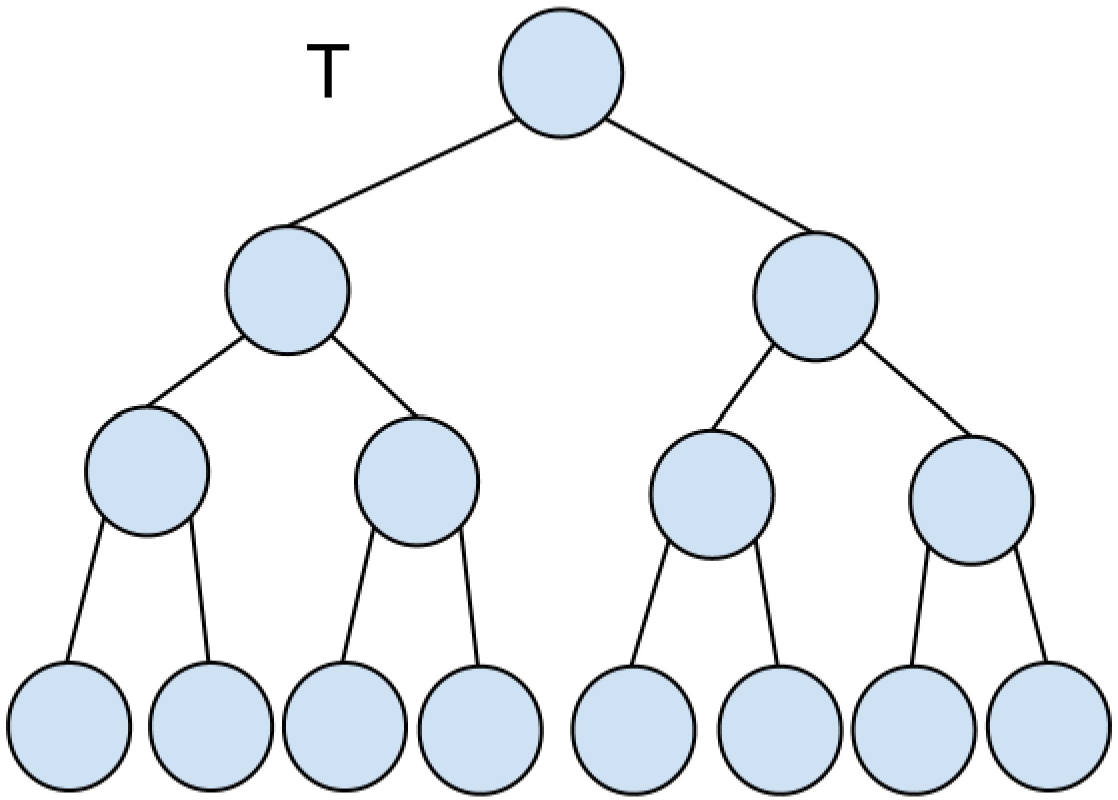
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

Example:

Depth of T=3.

Number of leaves in T=?

Number of nodes in T=?



A binary tree is complete if all the nodes have two children except the nodes in the last level.

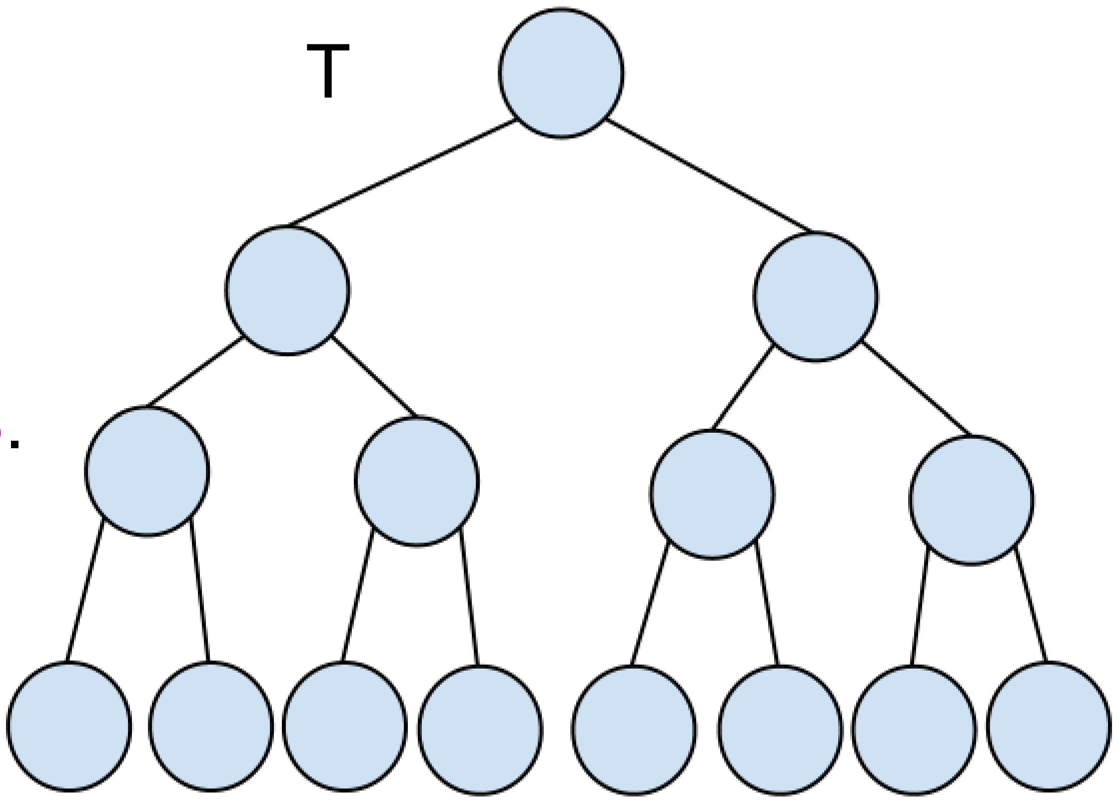
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

Example:

Depth of T=3.

Number of leaves in T=$2^3=8$.

Number of nodes in T=?



A binary tree is complete if all the nodes have two children except the nodes in the last level.

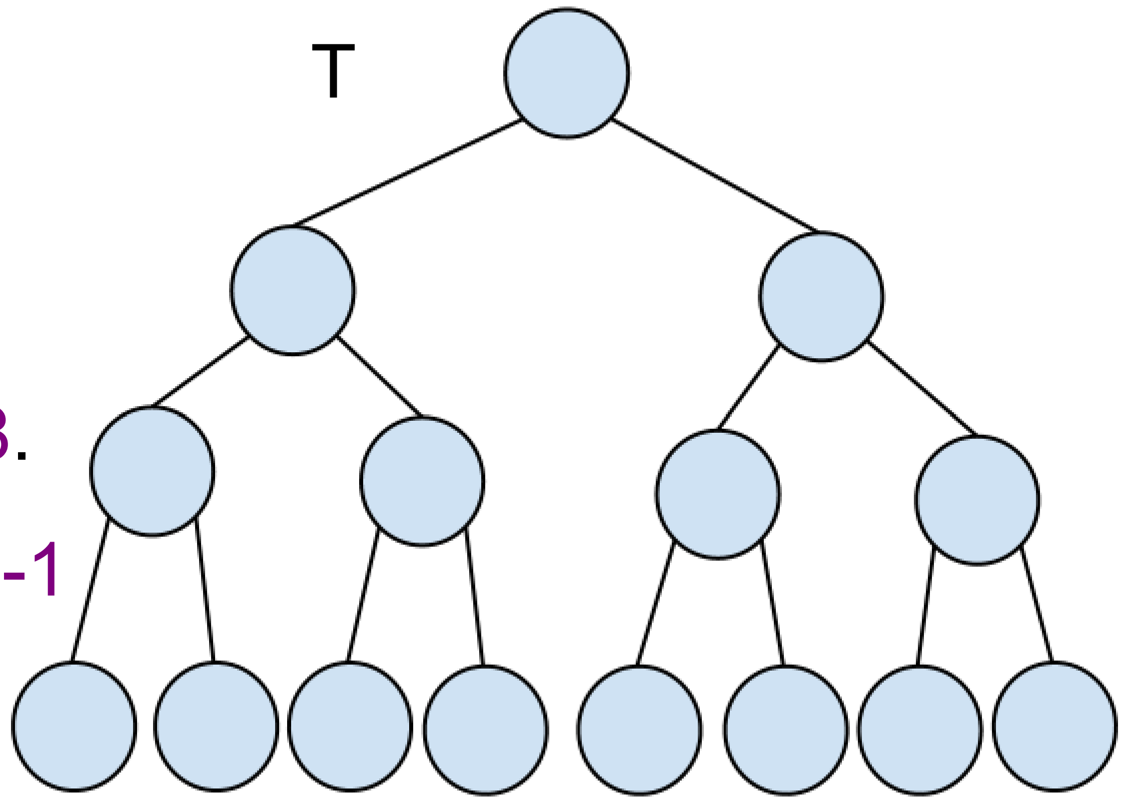
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

Example:

Depth of T=3.

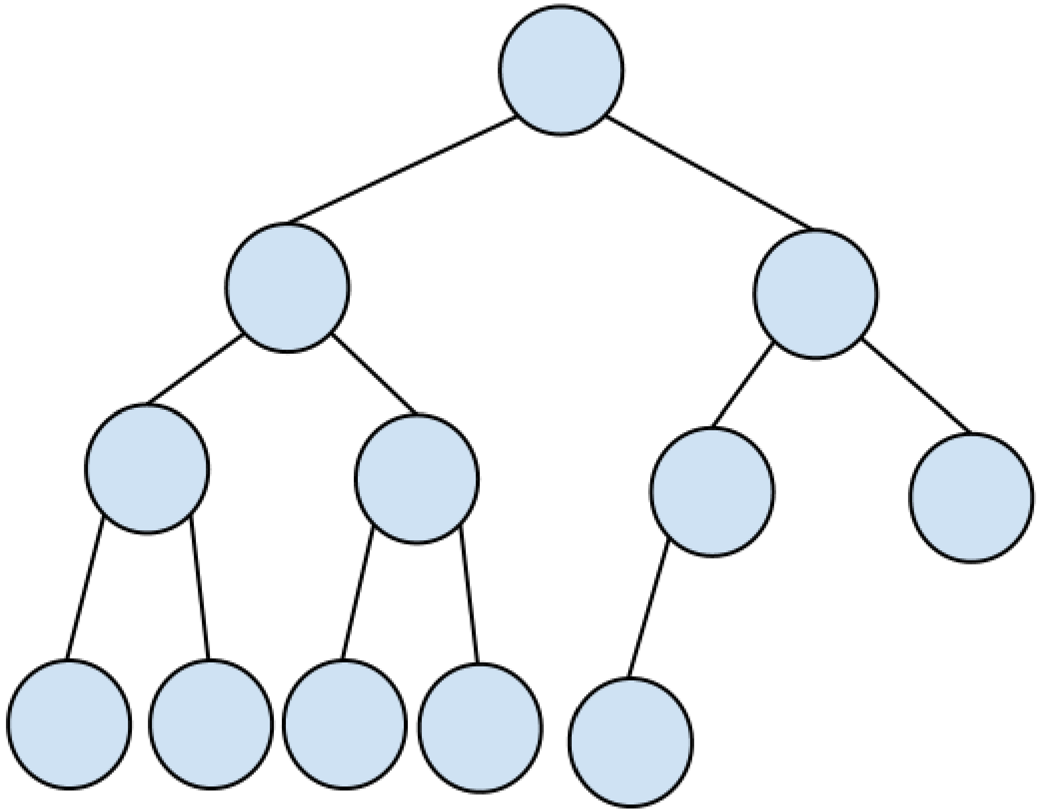
Number of leaves in T=$2^3=8$.

Number of nodes in T=$2^{3+1}-1$ =15.



Heap is like a complete binary tree except that the last level may be missing nodes, and if so is filled from left to right.

Note: A complete binary tree is a special case of a heap.



A heap is conveniently represented using arrays

Navigating a heap:

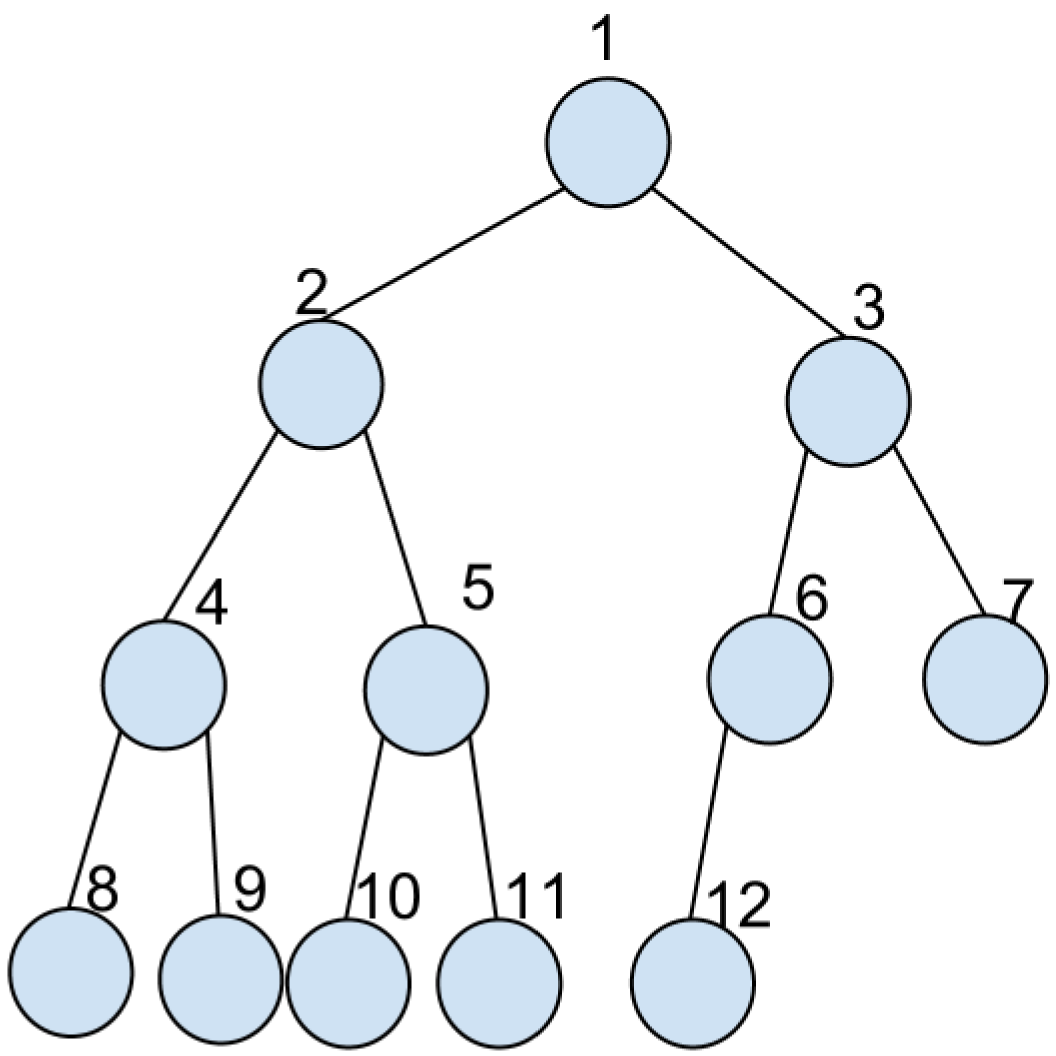
Root is A[1].

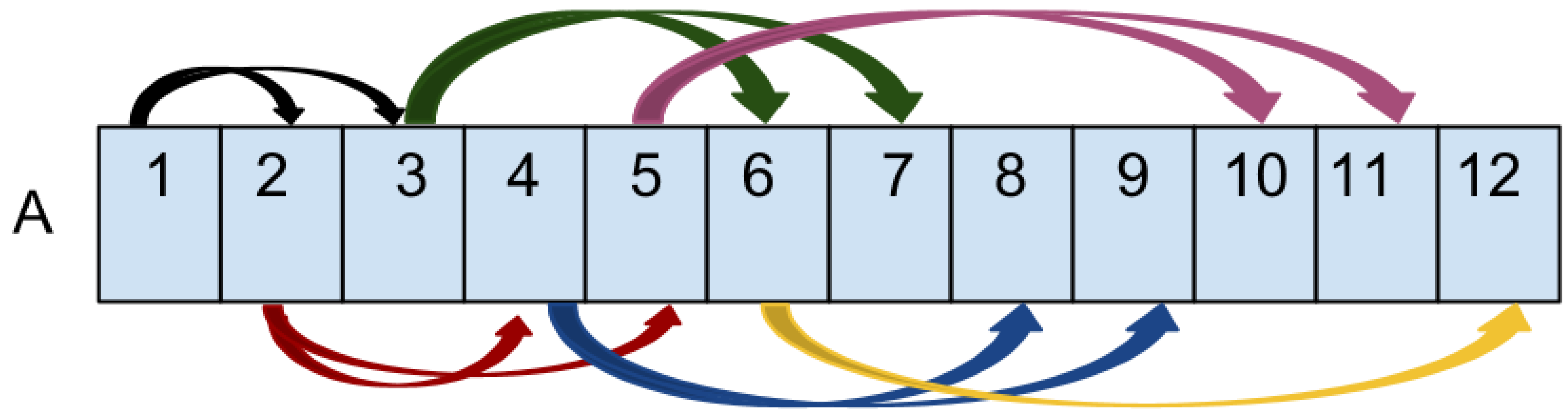
Given index i to a node:

Parent(i) = i/2

Left-Child(i) = 2i

Right-Child(i) = 2i+1



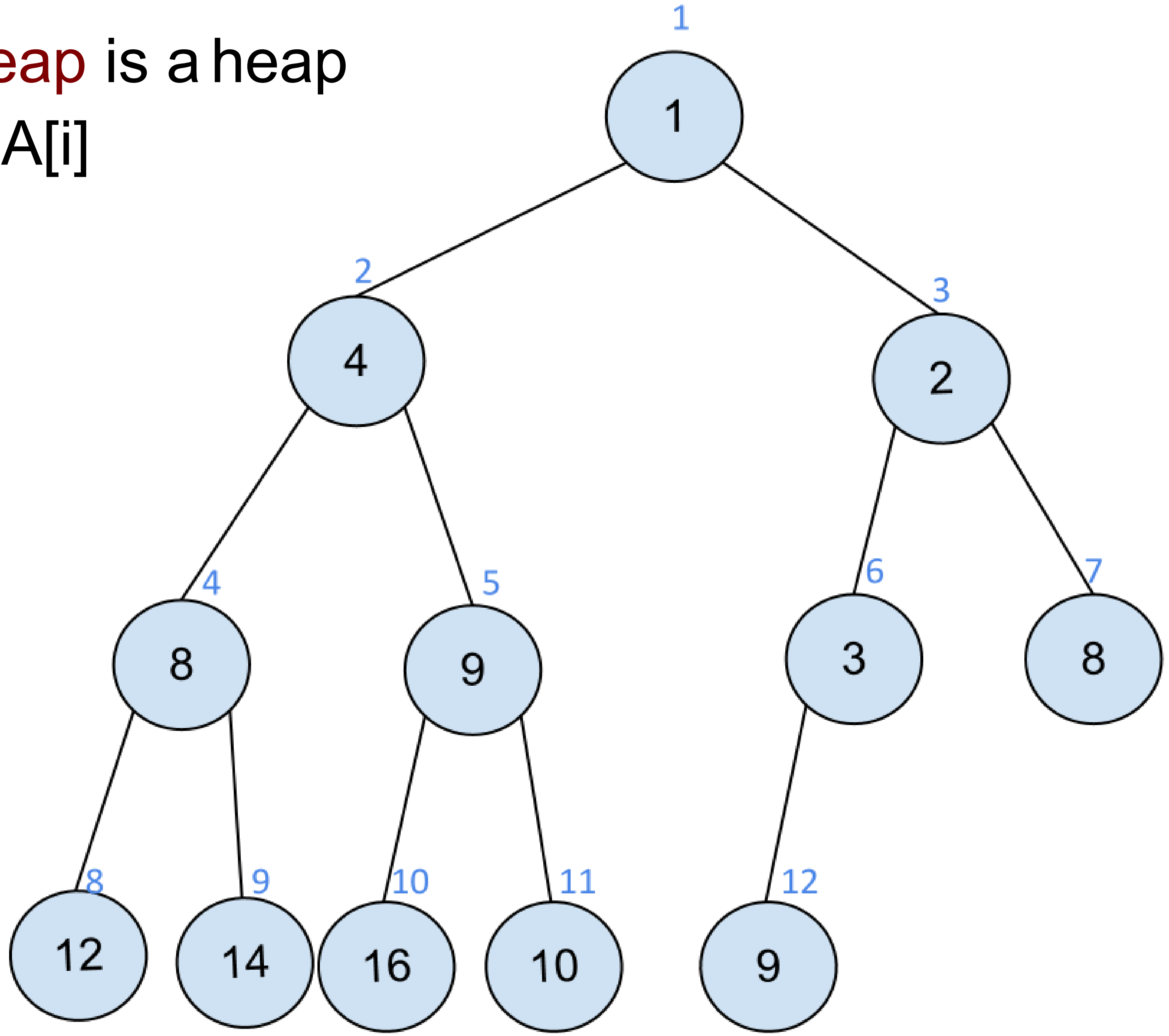


Heaps are useful to dynamically maintain a set of elements while allowing for extraction of minimum (priority queue)

The same results hold for extraction of maximum

We focus on minimum for concreteness.

●Definition: A min-heap is a heap where A[Parent(i)] ≤ A[i] for every i



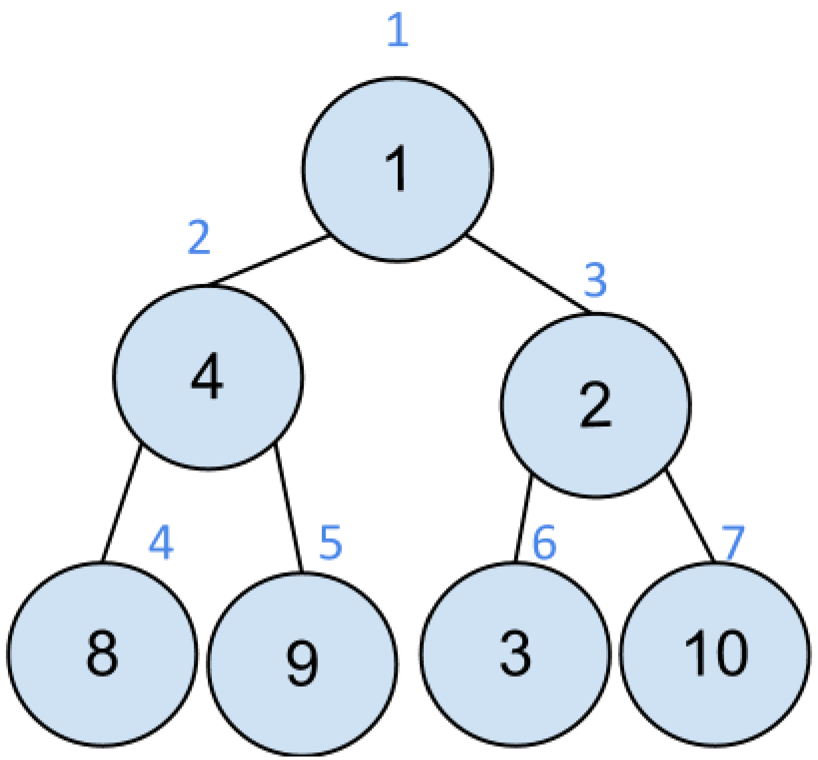
	1	2	3	4	5	6	7	8	9	10	11	12
A	1	4	2	8	9	3	8	12	14	16	10	9

Extracting the minimum element

In min-heap A , the minimum element is A[1].

Extract-Min-heap(A)

```
min:= A[1];
A[1]:= A[heap-size];
heap-size:= heap-size – 1;
Min-heapify(A, 1)
Return min;
```



Let's see the steps

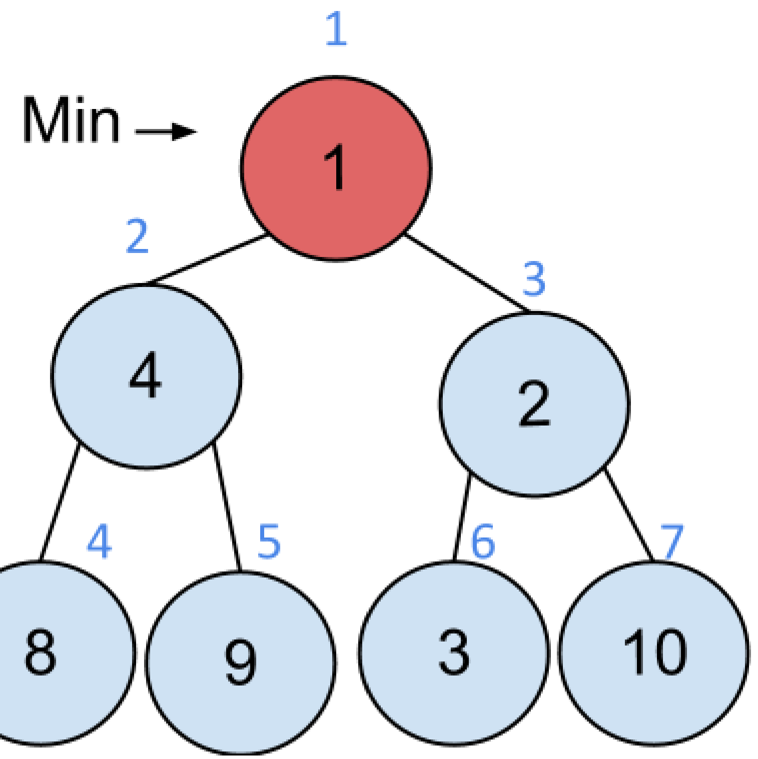
Extracting the minimum element

In min-heap A , the minimum element is A[1].

Extract-Min-heap(A)

min:= A[1];

A[1]:= A[heap-size];
heap-size:= heap-size – 1;
Min-heapify(A, 1)
Return min;



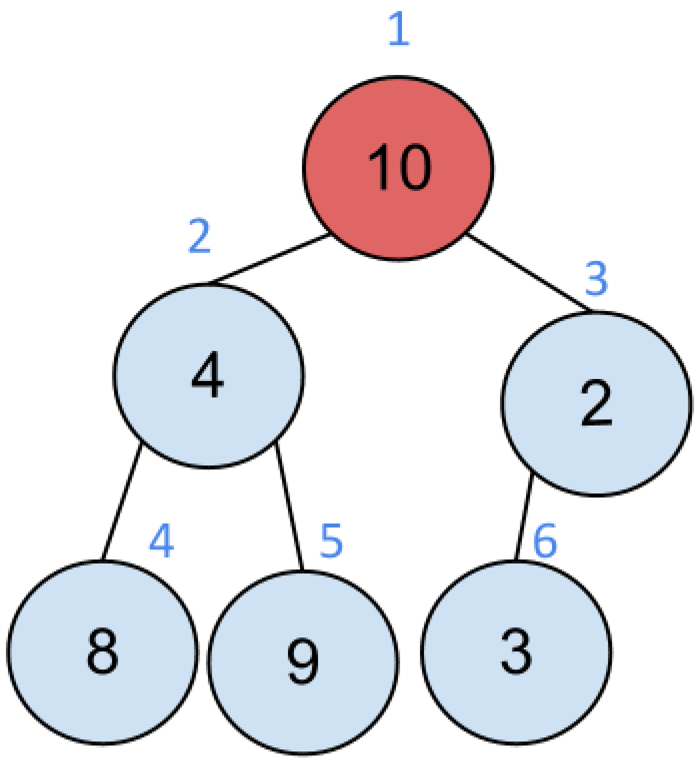
Extracting the minimum element

In min-heap A , the minimum element is A[1].

Extract-Min-heap(A)

min:= A[1];

A[1]:= A[heap-size];
heap-size:= heap-size – 1;
Min-heapify(A, 1)
Return min;



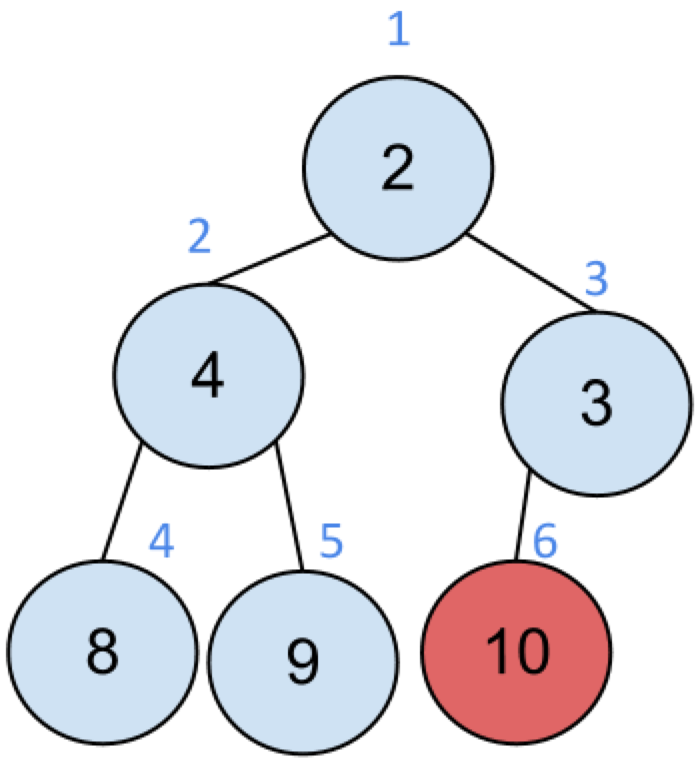
Extracting the minimum element

In min-heap A , the minimum element is A[1].

Extract-Min-heap(A)

min:= A[1];

A[1]:= A[heap-size];
heap-size:= heap-size – 1;
Min-heapify(A, 1)
Return min;



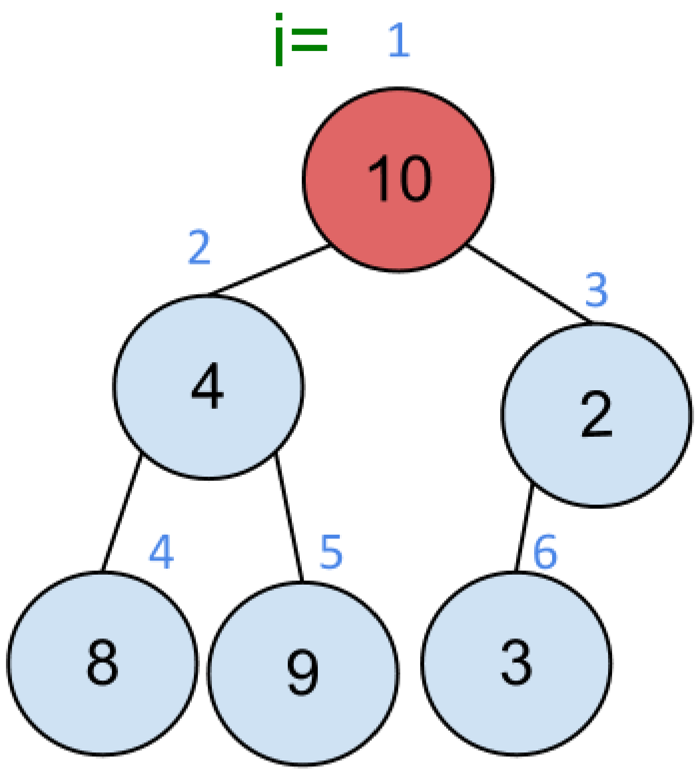
Min-heapify is a function that restores the min property

Min-heapify restores the min-heap property

given array A and index i such that trees rooted at left[i] and right[i] are min-heap, but A[i] maybe greater than its children

```
Min-heapify(A, i)
  Let j be the index of smallest node
    among {A[i], A[Left[i]], A[Right[i]] }

 If j ≠ i then {
    exchange A[i] and A[j]
    Min-heapify(A, j)
 }
```

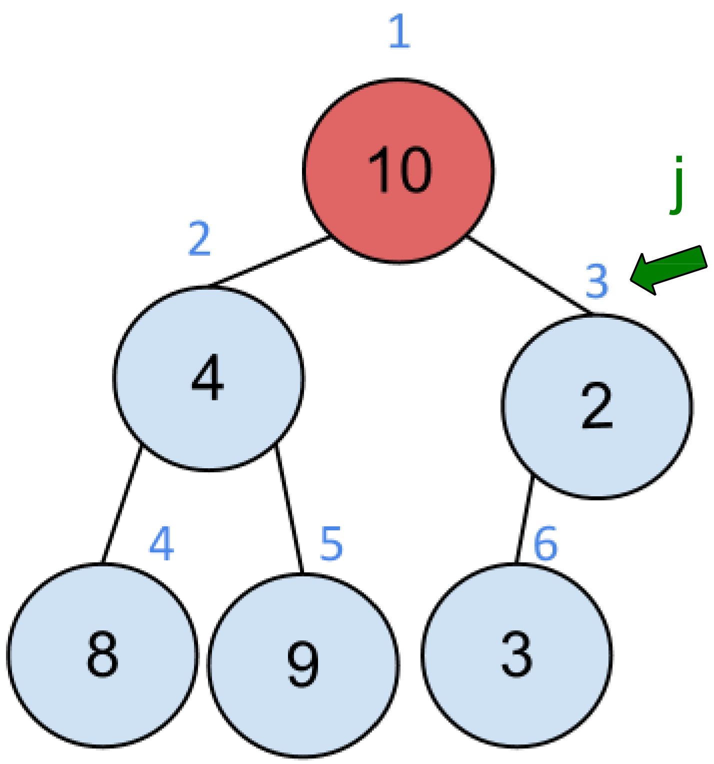


Min-heapify restores the min-heap property

given array A and index i such that trees rooted at left[i] and right[i] are min-heap, but A[i] maybe greater than its children

```
Min-heapify(A, i)
  Let j be the index of smallest node
    among {A[i], A[Left[i]], A[Right[i]] }

 If j ≠ i then {
    exchange A[i] and A[j]
    Min-heapify(A, j)
 }
```

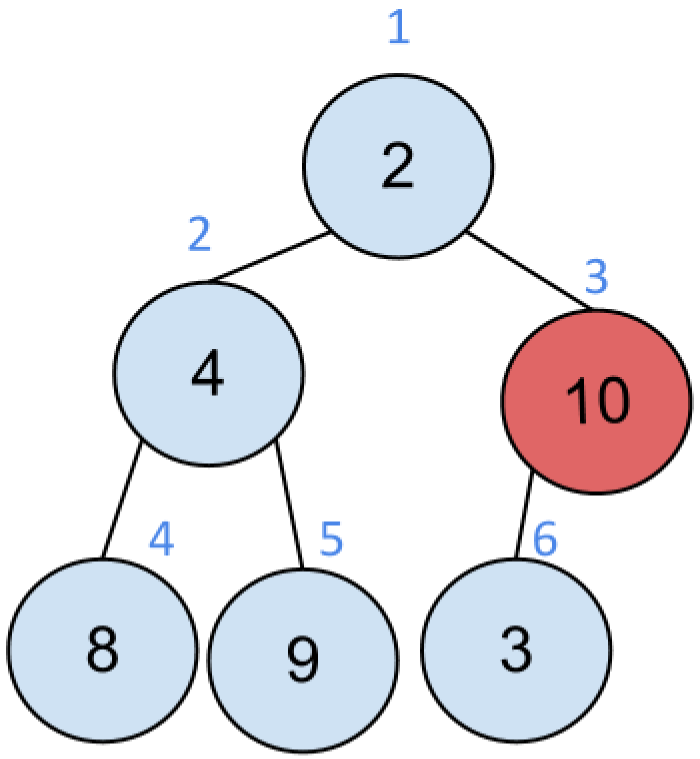


Min-heapify restores the min-heap property

given array A and index i such that trees rooted at left[i] and right[i] are min-heap, but A[i] maybe greater than its children

```
Min-heapify(A, i)
  Let j be the index of smallest node
    among {A[i], A[Left[i]], A[Right[i]] }

 If j ≠ i then {
    exchange A[i] and A[j]
    Min-heapify(A, j)
 }
```

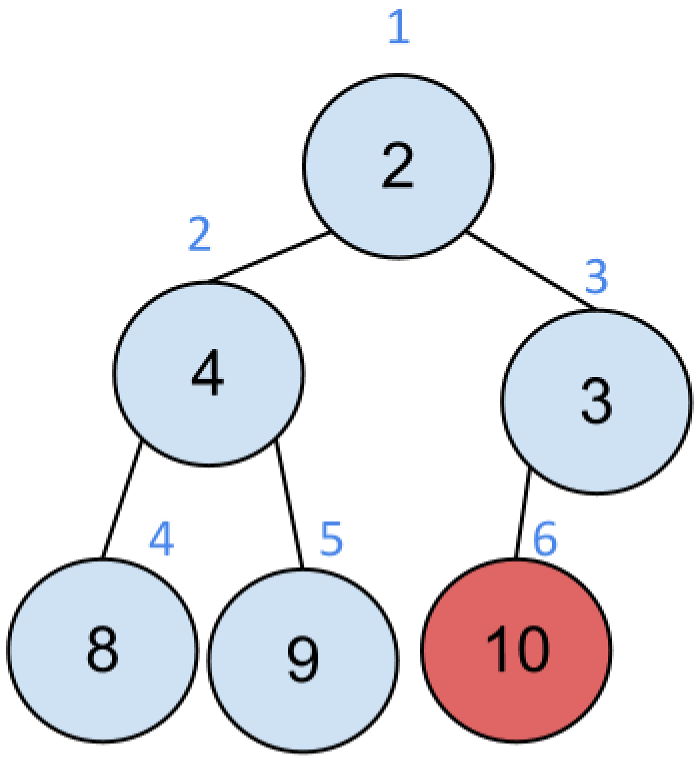


Min-heapify restores the min-heap property

given array A and index i such that trees rooted at left[i] and right[i] are min-heap, but A[i] maybe greater than its children

```
Min-heapify(A, i)
  Let j be the index of smallest node
    among {A[i], A[Left[i]], A[Right[i]] }

 If j ≠ i then {
    exchange A[i] and A[j]
    Min-heapify(A, j)
 }
```

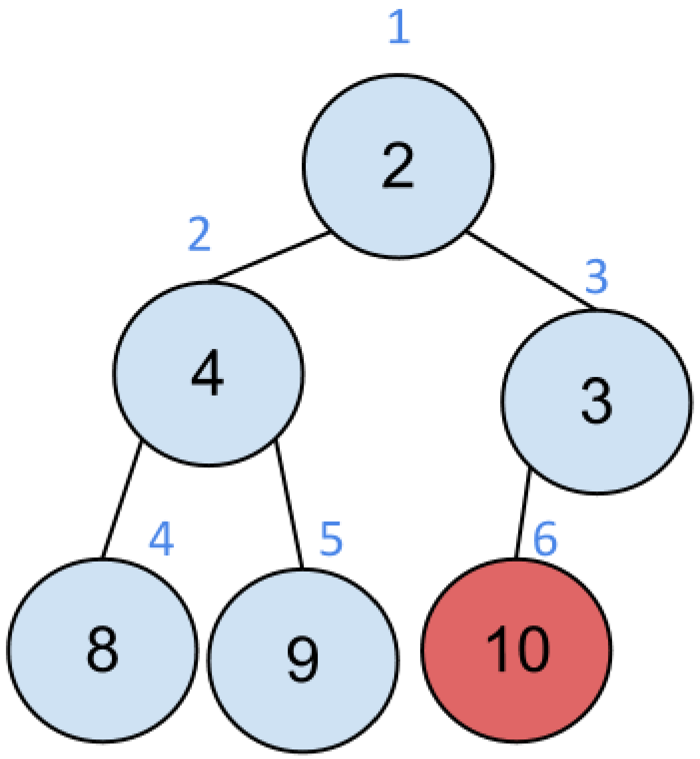


Min-heapify restores the min-heap property

given array A and index i such that trees rooted at left[i] and right[i] are min-heap, but A[i] maybe greater than its children

```
Min-heapify(A, i)
  Let j be the index of smallest node
    among {A[i], A[Left[i]], A[Right[i]] }

 If j ≠ i then {
    exchange A[i] and A[j]
    Min-heapify(A, j)
 }
```



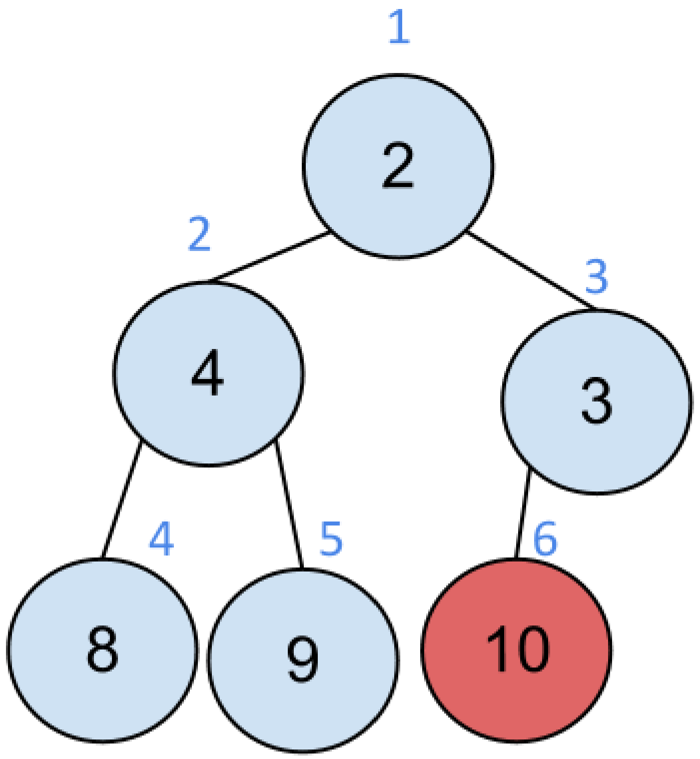
Running time = ?

Min-heapify restores the min-heap property

given array A and index i such that trees rooted at left[i] and right[i] are min-heap, but A[i] maybe greater than its children

```
Min-heapify(A, i)
  Let j be the index of smallest node
    among {A[i], A[Left[i]], A[Right[i]] }

 If j ≠ i then {
    exchange A[i] and A[j]
    Min-heapify(A, j)
 }
```



Running time = depth = O(log n)

Recall Extract-Min-heap(A)

```
min:= A[1];
A[1]:= A[heap-size];
heap-size:= heap-size –  1;
Min-heapify(A, 1)
Return min;
```

Hence both Min-heapify and
Extract-Min-Heap take time O(log n).

Next: How do you insert into a heap?

Insert-Min-heap (A, key)

```
 heap-size[A] := heap-size[A]+1;
 A[heap-size] := key;

for(i:= heap-size[a]; i>1 and A[parent(i)] > A[i]; i:= parent[i])
    exchange(A[parent(i)], A[i])
```

Running time = ?

Insert-Min-heap (A, key)

```
 heap-size[A] := heap-size[A]+1;
 A[heap-size] := key;

for(i:= heap-size[a]; i>1 and A[parent(i)] > A[i]; i:= parent[i])
     exchange(A[parent(i)], A[i])
```

Running time = O(log n).

Suppose we start with an empty heap and insert n elements.
By above, running time is O(n log n).

But actually we can achieve O(n).

Build Min-heap

Input: Array A, output: Min-heap A.

```
For ( i := length[A]/2;  i <0;  i - -)
      Min-heapify(A, i)
```

Running time = ?

Min-heapify takes time O(h) where h is depth.

How many trees of a given depth h do you have?

Build Min-heap

Input: Array A, output: Min-heap A.

For (i := length[A]/2; i <0; i - -)
    Min-heapify(A, i)

Running time = $O(\sum_{h < \log n} n/2^h)\, h$

$= n\, O(\sum_{h < \log n} h/2^h)$

= ?

Build Min-heap

Input: Array A, output: Min-heap A.

For (i := length[A]/2; i <0; i - -)
    Min-heapify(A, i)

Running time = $O(\sum_{h < \log n} n/2^h) h$

$= n\, O(\sum_{h < \log n} h/2^h)$

$= O(n)$

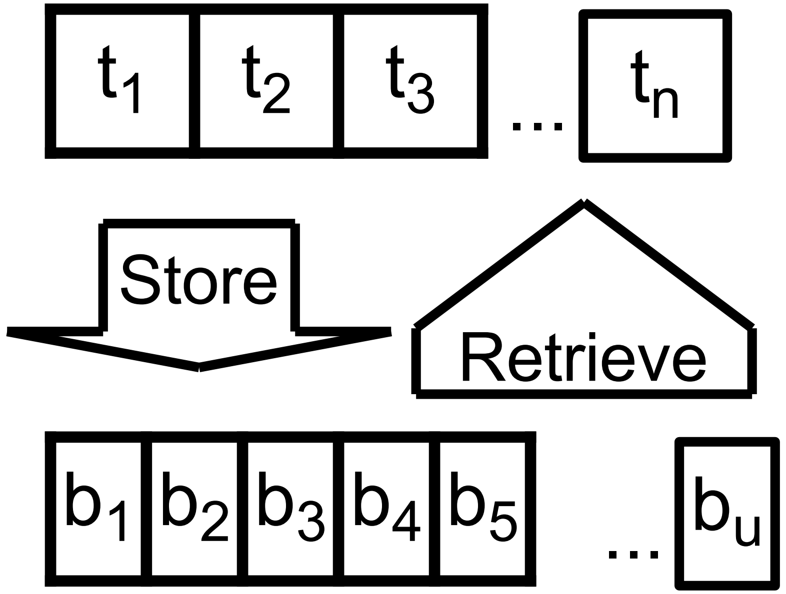
Next:

Compact (also known as succinct) arrays

Bits vs. trits

- Store n "trits" $t_1, t_2, \ldots, t_n \in \{0,1,2\}$

  In u bits $b_1, b_2, \ldots, b_u \in \{0,1\}$



- Want:

  Small space u (optimal $= \lceil n \lg_2 3 \rceil$)

  Fast retrieval: Get t_i by probing few bits (optimal = 2)

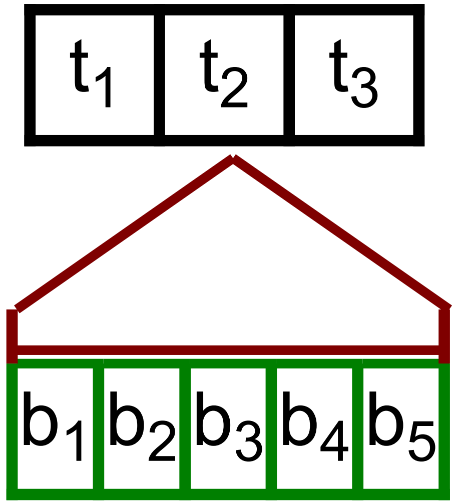
Two solutions

- Arithmetic coding:

  Store bits of $(t_1, \ldots, t_n) \in \{0, 1, \ldots, 3^n - 1\}$

  Optimal space: $\lceil n \lg_2 3 \rceil \approx n \cdot 1.584$

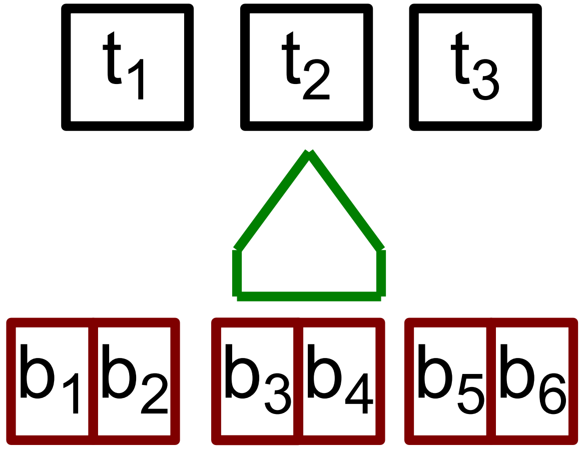
  Bad retrieval: To get t_i probe all > n bits



- Two bits per trit

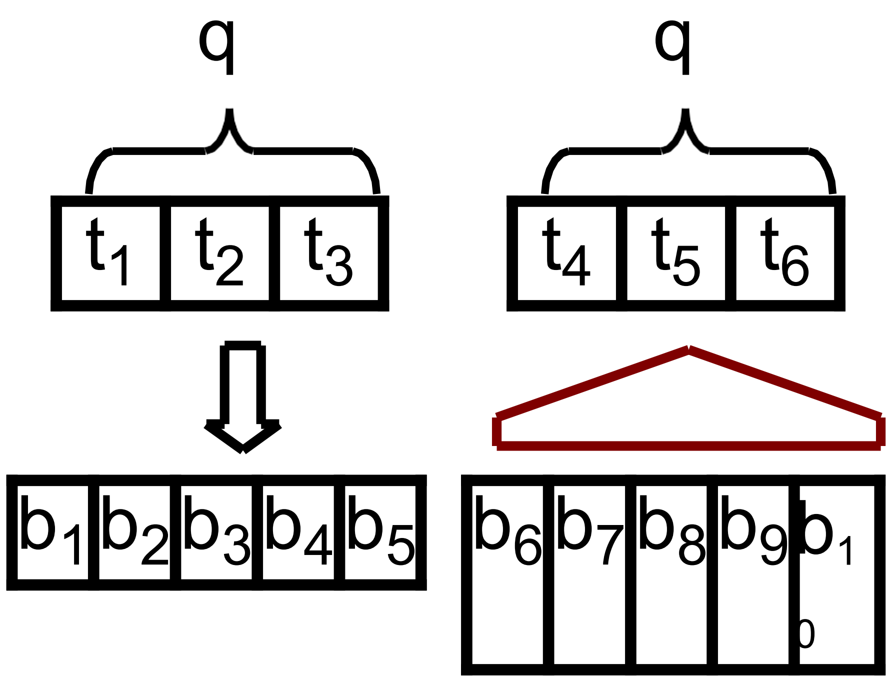
  Bad space: $n \cdot 2$

  Optimal retrieval: Probe 2 bits



Polynomial tradeoff

- Divide n trits $t_1, \ldots, t_n \in \{0,1,2\}$ in blocks of q
- Arithmetic-code each block



Space: $\lceil q \lg_2 3 \rceil n/q < (q \lg_2 3 + 1)\, n/q$

$= n \lg_2 3 + n/q$

Retrieval: Probe O(q) bits

polynomial tradeoff between probes, redundancy

Exponential tradeoff

- Breakthrough [Pătraşcu '08, later + Thorup]

Space: $n \lg_2 3 + n/2^{\Omega(q)}$

Retrieval: Probe q bits

exponential tradeoff between redundancy, probes

- E.g., optimal space $\lceil n \lg_2 3 \rceil$, probe $O(\lg n)$

Delete scenes

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

Function	**Search time**	**Extra space**
$f(x) = x$	?	?
$t = 2^n$, open addressing		

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

Function	Search time	Extra space
$f(x) = x$ $t = 2^n$, open addressing	$O(1)$	2^u
Any deterministic function	?	?

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

Function	**Search time**	**Extra space**
$f(x) = x$ $t = 2^n$, open addressing	$O(1)$	2^u
Any deterministic function	n	0
Random function	? expected	?

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \to [t]$, resolve collisions by chaining

Function	Search time	Extra space
$f(x) = x$ $t = 2^n$, open addressing	$O(1)$	2^u
Any deterministic function	n	0
Random function	n/t expected $\forall\ x \neq y, \Pr[f(x)=f(y)] \leq 1/t$	$2^u \log(t)$
Now what? We ``derandomize'' random functions		

Problem: Dynamically support n search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

Function	**Search time**	**Extra space**
$f(x) = x$ $t = 2^n$, open addressing	$O(1)$	2^u
Any deterministic function	n	0
Random function	n/t expected $\forall\ x \neq y$, $\Pr[f(x)=f(y)] \leq 1/t$	$2^u \log(t)$
Pseudorandom function A.k.a. hash function	n/t expected Idea: Just need $\forall\ x \neq y$, $\Pr[f(x)=f(y)] \leq 1/t$	$O(u)$

Stack

Operations: Push, Pop
Last-in-first-out

Queue

Operations: Enqueue, Dequeue
First-in-first-out

Simple implementation using arrays.
Each operation supported in $O(1)$ time.