Data structures

- Organize your data to support various queries using little time and/or space

- 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, ...
- 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

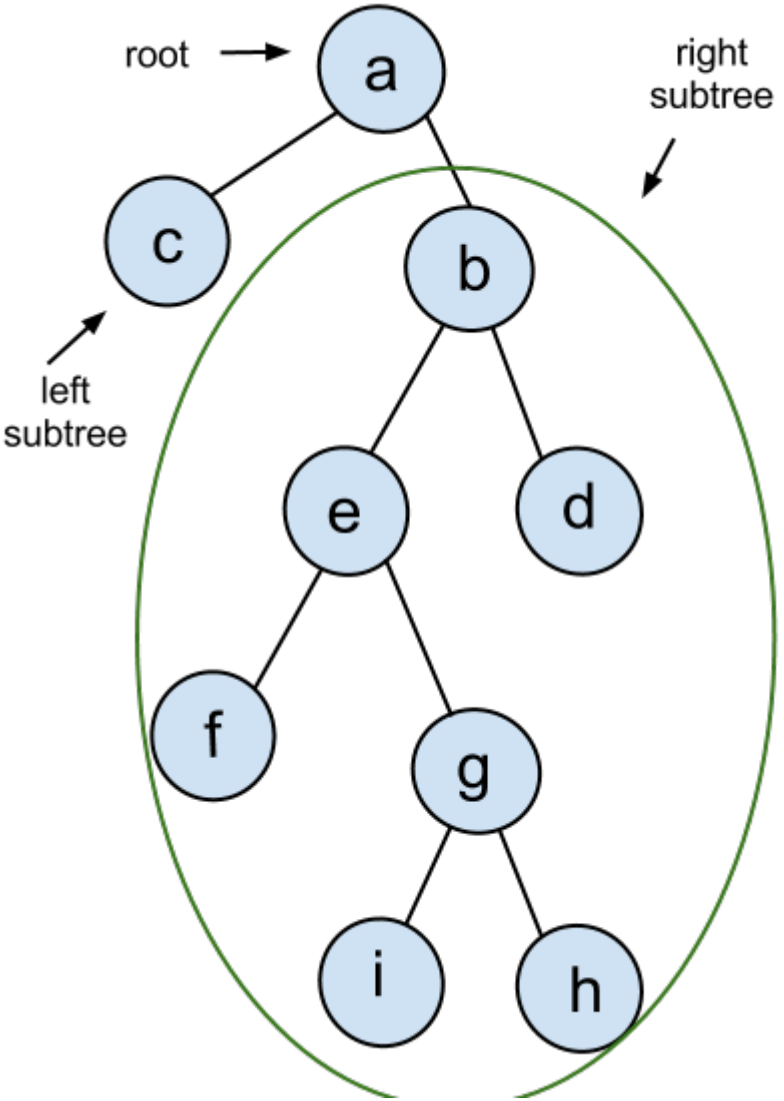
Left subtree = {c}

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

Parent(b) = a

Leaves = nodes with no children

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



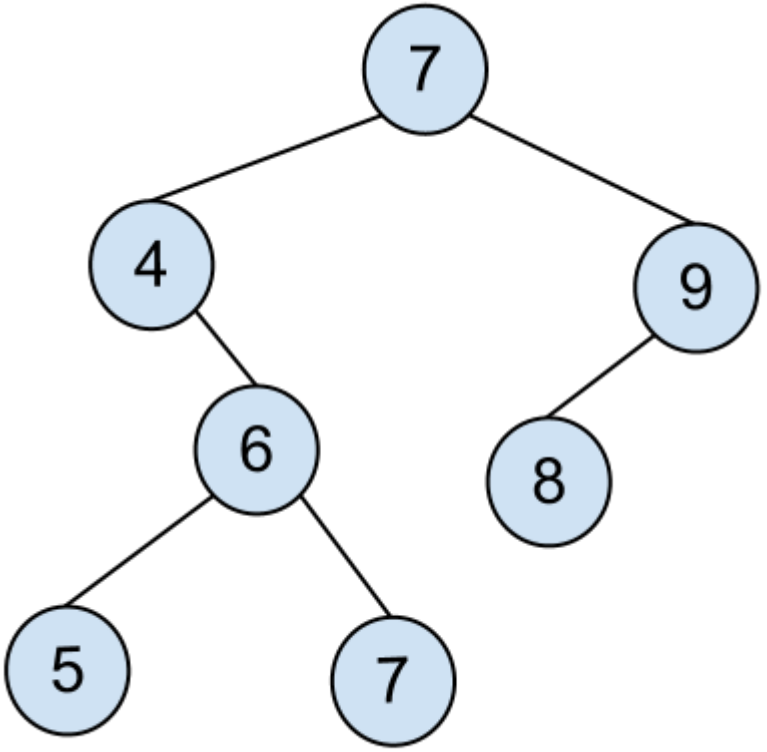
Binary Search Tree is a data structure where we store data in nodes of a binary tree and refer to them as key of that node.

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) ≤ key(x)

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

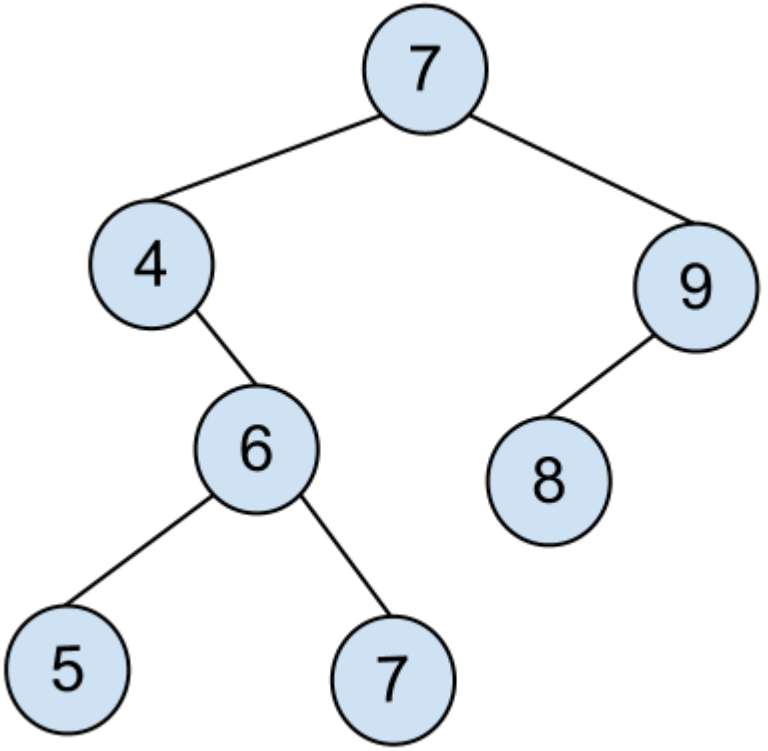
Example:



Binary Search

Looking for k in tree T given root x:

```
tree-search(x,k)
  If k=key[x]
     then return x
  if k< key[x]
     then return tree-search(left[x],k)
  else
     return tree-search(right[x],k)
```



Running time = the depth of the tree $\in$ [log n, n]

Tree is balanced if depth $\leq$ 1+log n $\Rightarrow$ search time O(log n)

Binary Search in a tree 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

Want to support search, insert, and delete in time O(log n)

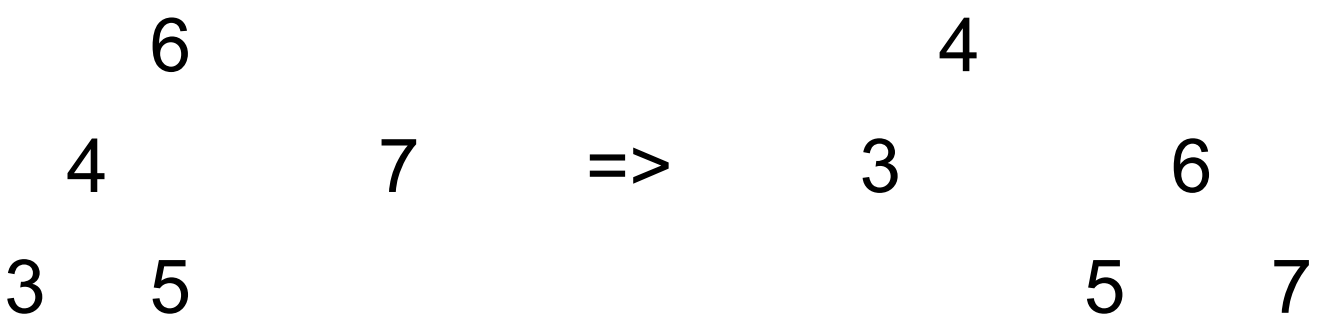
When inserting and deleting, must restructure the tree to keep it balanced or almost balanced.

- AVL trees: binary trees.

  In any node, heights of children differ by ≤ 1.

  Maintain by rotations

Right rotation at node x with key 6:



Temp = x; x = x.left; temp.left =x.right; x.right = temp;

Left rotation is symmetric.

● 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.

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

First we see pictures,

then formalize it,

then go back to pictures.

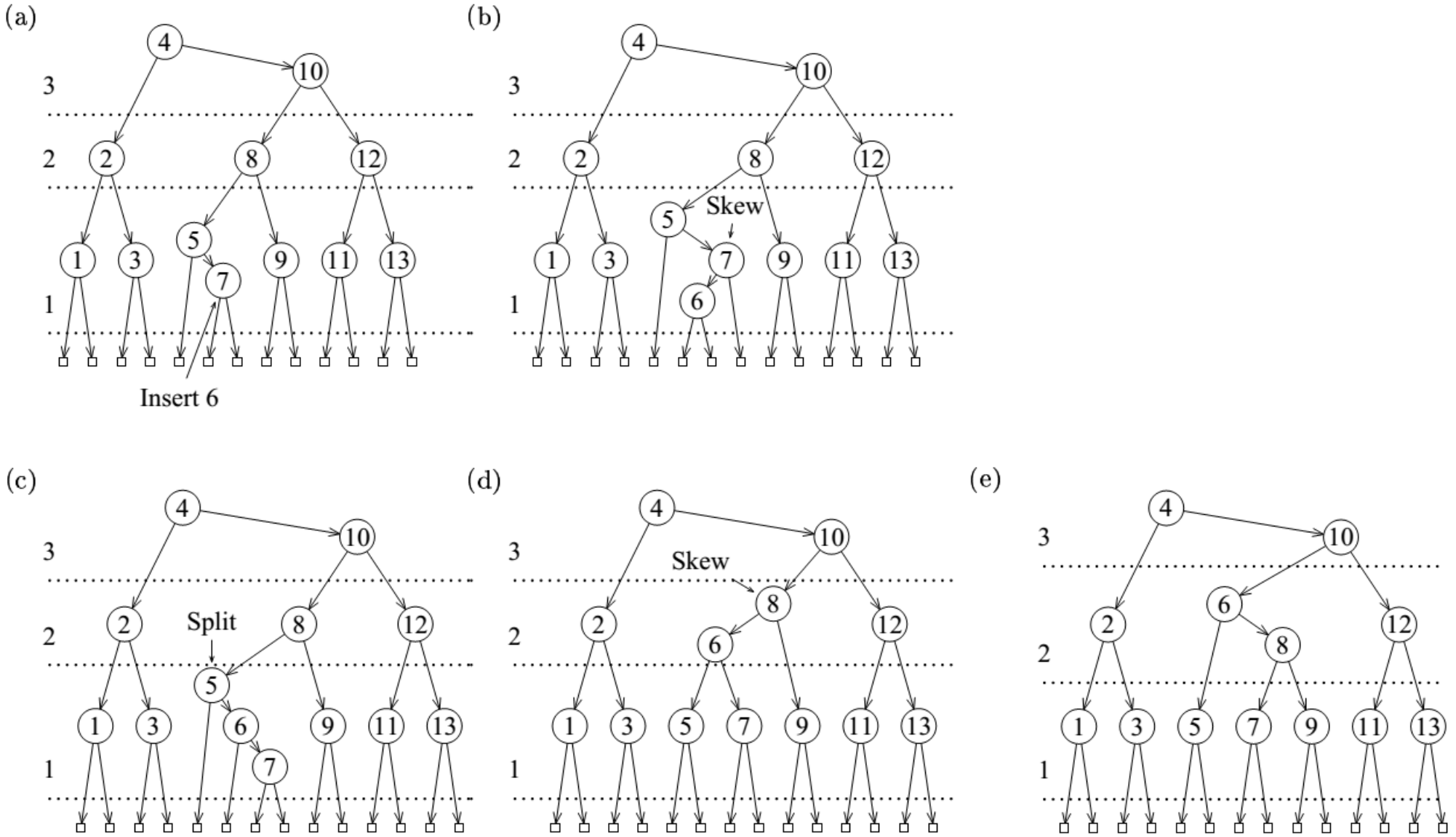


Fig. 1. *Example of insertion into a BB-tree. The levels are separated by horizontal lines.*

Images taken from the original paper

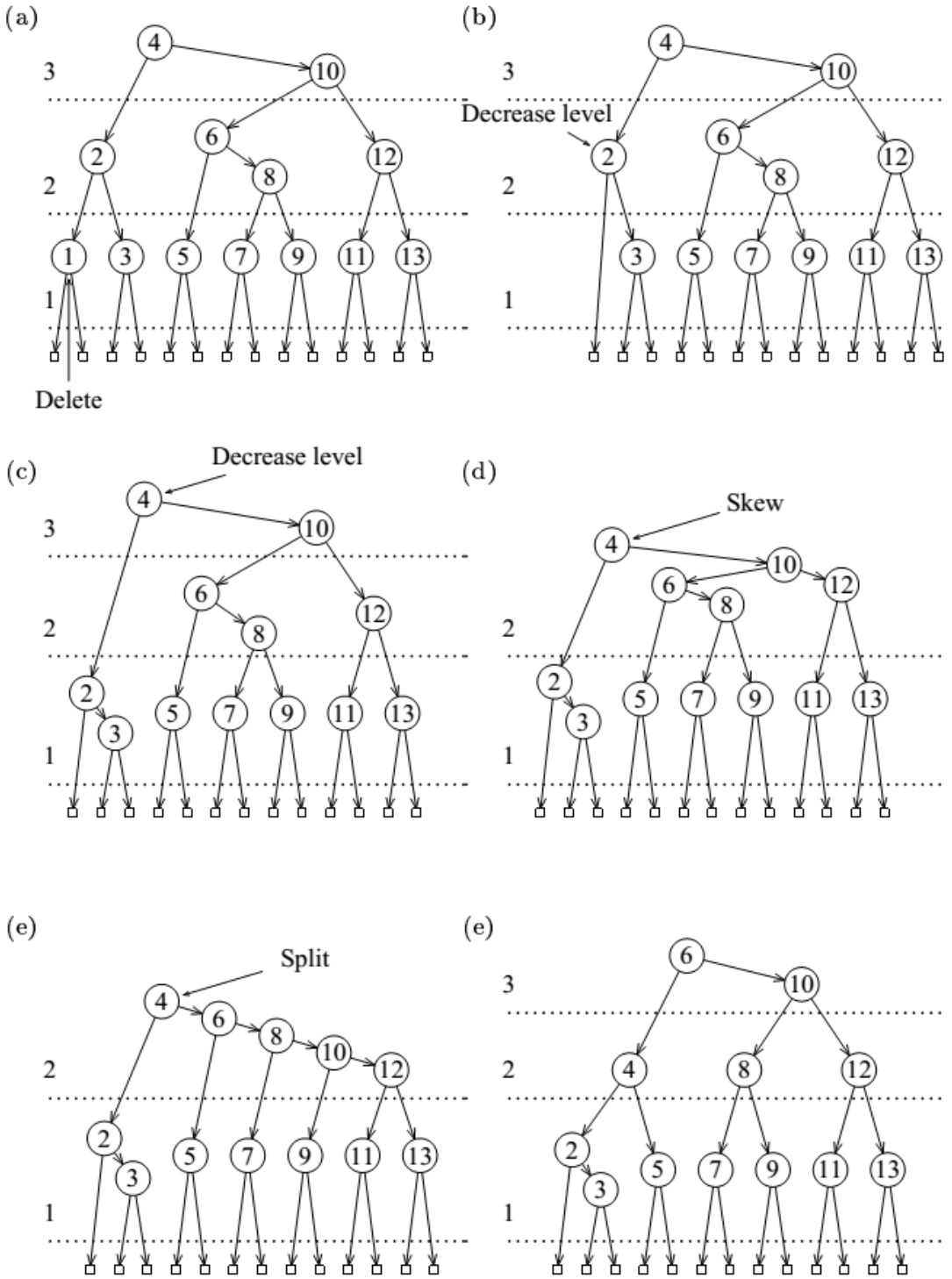


Fig. 2. *Example of deletion.*

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

- Each node has a level, which is 1 for leaves.

  Convention: Have “sentinels” at level 0.

  Every node, even leaves, has two children.

- Rule: The only allowed path with nodes of the same level is a left-right edge.

The end point always has level < starting point + 1.

This implies that height is O(log n).

- Rule of thumb for restructuring:

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

right rotation at x

Split(x): If x.right.right has same level as x

left rotation at x. increase the level of x.

(Think of x as pointer to cell, now occupied by what was x.right.)

Decrease Level(x): If one of x's children is two levels below x, decrese the level of x by one. If x.right had the same level of x, decrease the level of x.right by one too.

```
Insert(x): {
  Search(x).  Insert x as a new leaf where it should have been.
  Follow the path from x to the root and at each node y do:
      Skew(y).
      Split(y).
  }

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);
  }
```

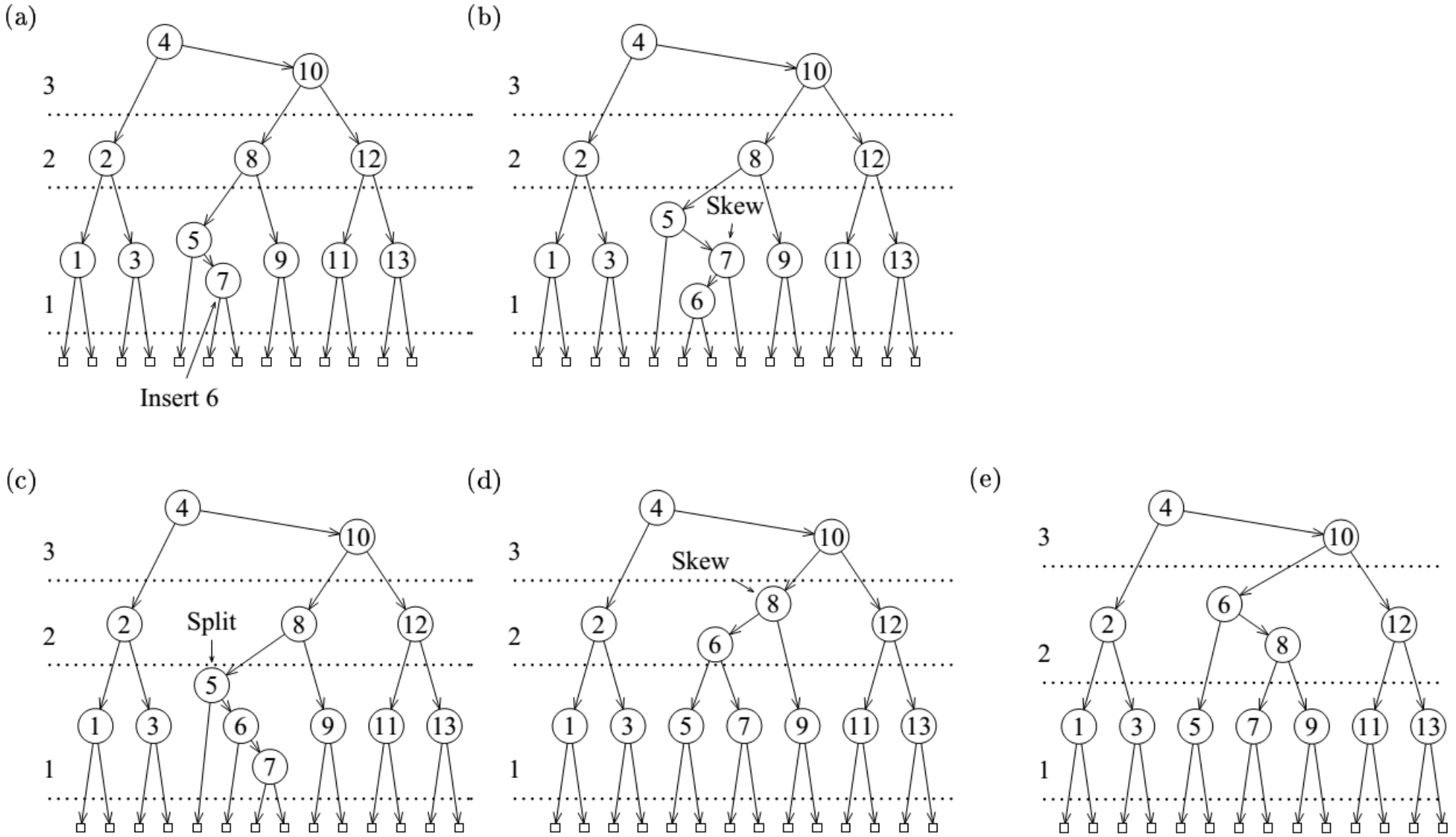


Fig. 1. *Example of insertion into a BB-tree. The levels are separated by horizontal lines.*

Images taken from the original paper

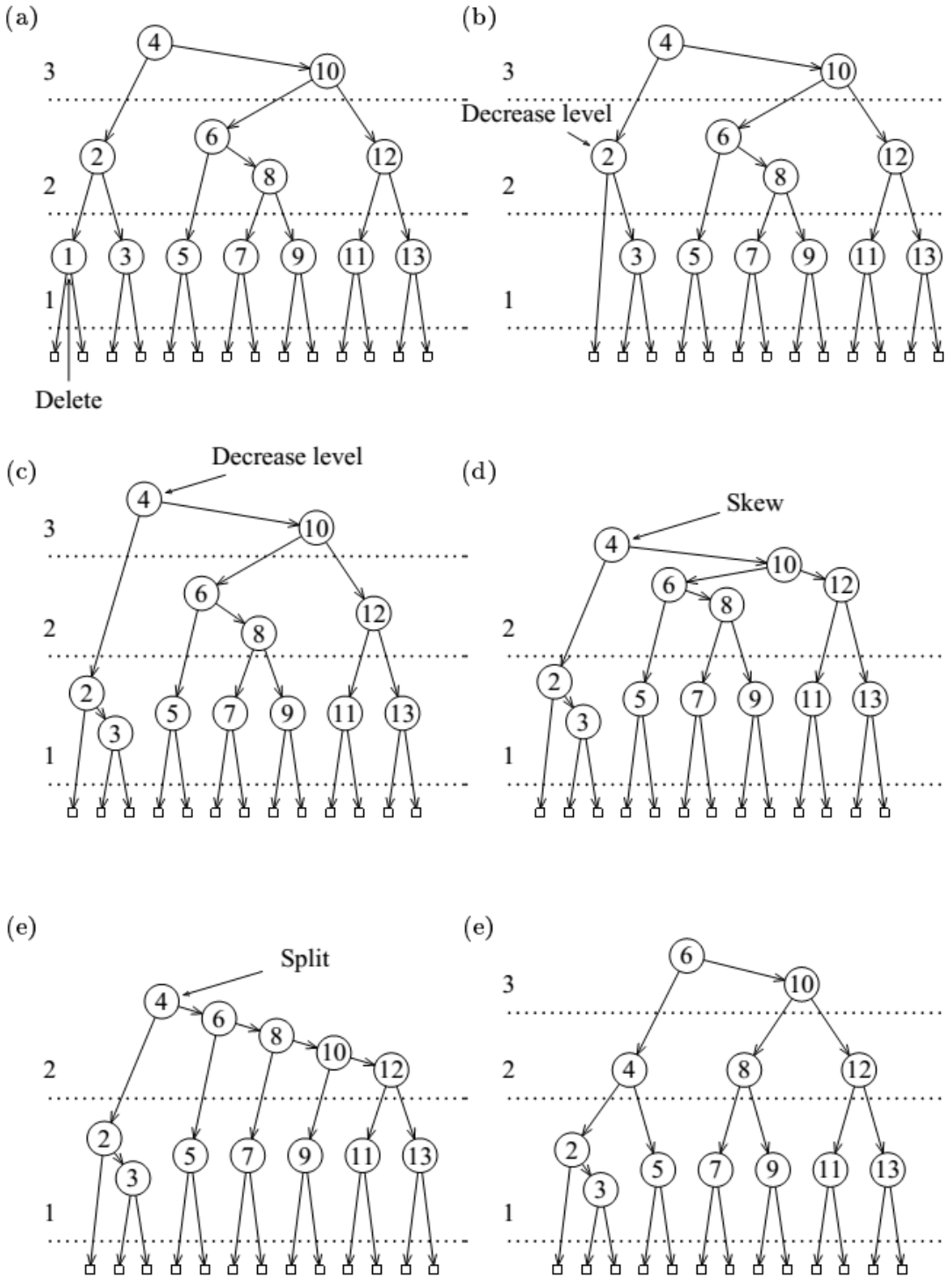


Fig. 2. *Example of deletion.*

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

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, e.g., 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:

You could use new/malloc free/dispose to add and remove nodes.

However, this may cause memory segmentation.

It is possible to implement any tree using an array A in such a way 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.

Running time: O(log n) for Search, insert, and delete.

Space: O(n). For ach key we need to store level and pointers to children, and possibly pointer to parent too.

Can we achieve space n + o(n)?

Surprisingly, this is possible:

Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees

by Franceschini and Grossi

Project: Explain to me how that works.

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 $f : \{0,1\}^u \rightarrow [t]$, 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 $\forall\ x \neq y$, $Pr[f(x)=f(y)] \leq 1/t$	$2^u \log(t)$
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)

Construction of hash function:

Let t be prime. Write u-bit elements in base t.

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

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

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

Claim: $\forall\, x \neq x'$, $Pr_a [f_a(x) = f_a(x')] = 1/t$

Different constructions of hash function:
u-bit keys to r-bit hashes

Classic solution: pick a prime $p>2^u$, and a random a in [p], and

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

Problem: mod p is slow, even with Mersenne primes ($p=2^i-1$)

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

$h_b(x) = ((bx) \bmod 2^u) \operatorname{div} 2^{u-r}$

= bits from u-r to u of integer product bx

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

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

Static search:

Given n elements, want a hash function that gives no collisions.

Probabilistic method: Just hash to [t] = n^2 elements

Pr[$\exists\, x \neq y : \text{hash}(x) = \text{hash}(y)$]

$\leq n^2/2 \Pr[\text{hash}(0) = \text{hash}(1)]$ (union bound)

$\leq n^2 / (2t) = 1/2$

➔ $\exists$ hash : $\forall\, x \neq y$, $\text{hash}(x) \neq \text{hash}(y)$ (probabilistic method)

Can you have no collisions with [t] = O(n)?

Static search:

Given n elements, want a hash function that gives no collisions.

Two-level hashing:

- First hash to t = O(n) elements,
- then hash again using the previous method. That is, if i-th cell in first level has c_i elements, hash to c_i^2 cells at the second level.

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

Note $\sum_{i \le t} c_i^2$ =

Θ(expected number of colliding pairs in first level) =

O(???)

Static search:

Given n elements, want a hash function that gives no collisions.

Two-level hashing:

- First hash to t = O(n) elements,
- then hash again using the previous method. That is, if i-th cell in first level has c_i elements, hash to c_i^2 cells at the second level.

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

Note $\sum_{i \le t} c_i^2$ =

Θ(expected number of colliding pairs in first level) =

$O(n^2 / t)$ =

O(n)

Next:

More about arrays, and heaps.

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.

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

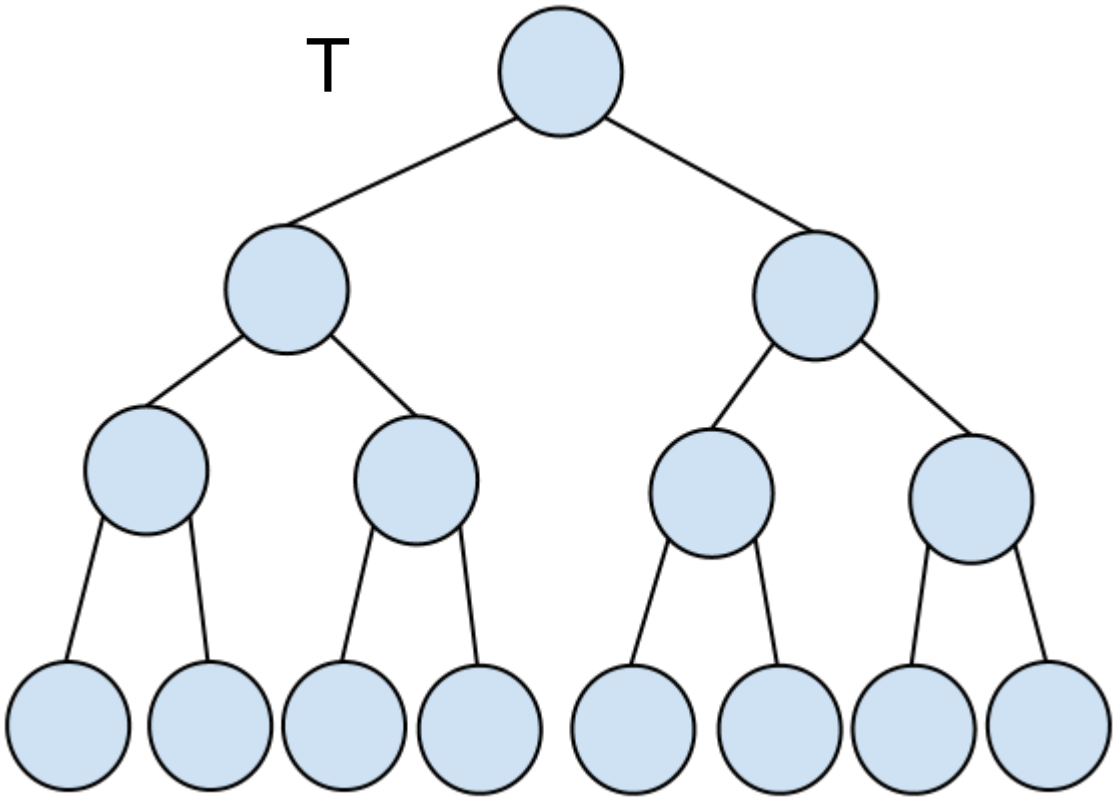
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

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.

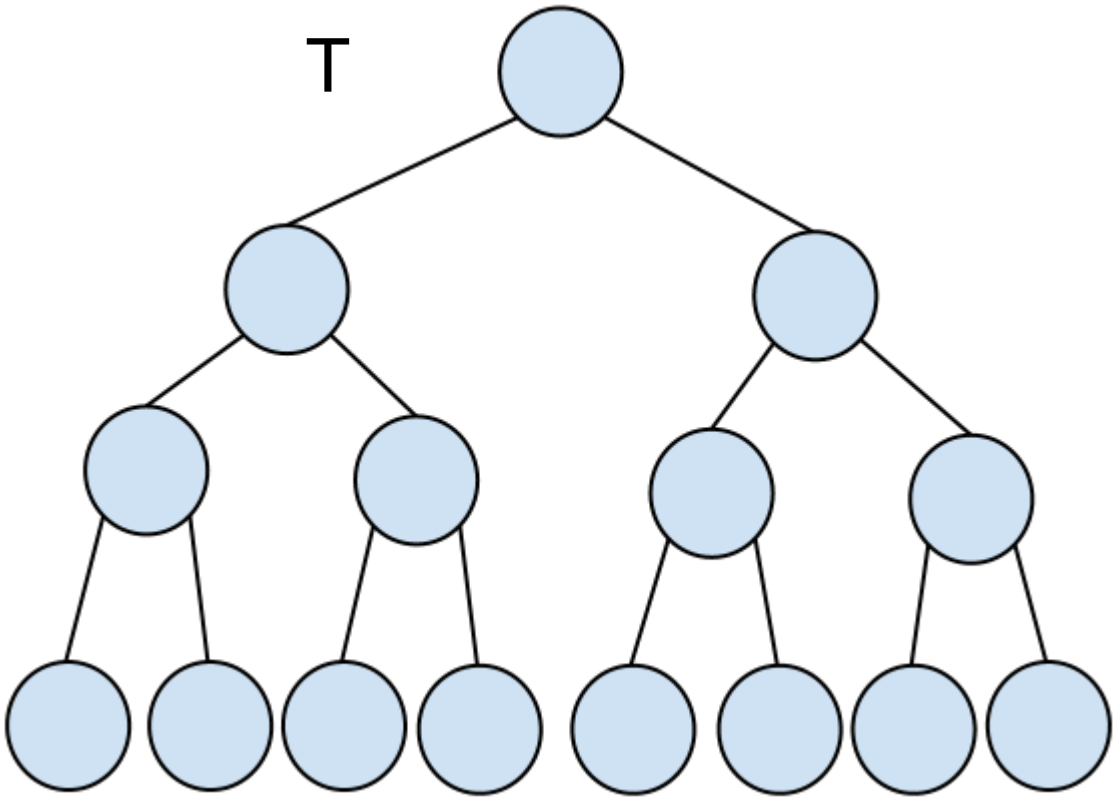
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

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.

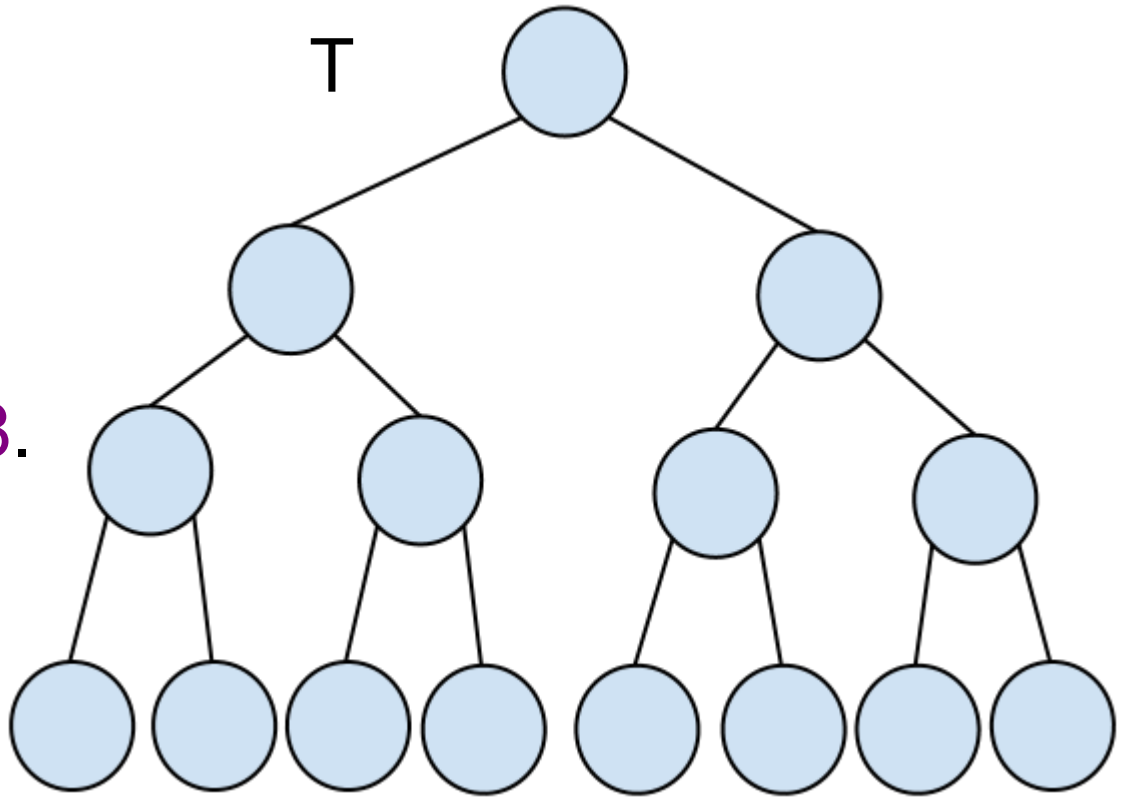
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

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.

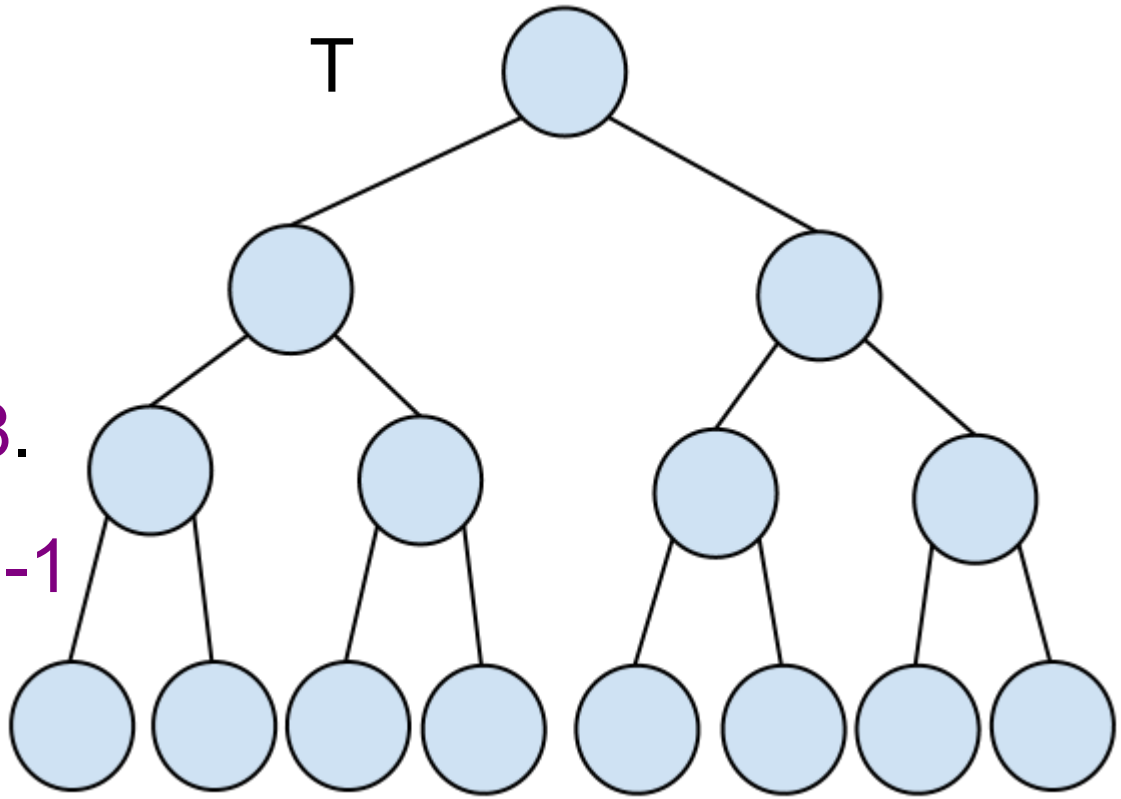
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

Example:

Depth of T=3.

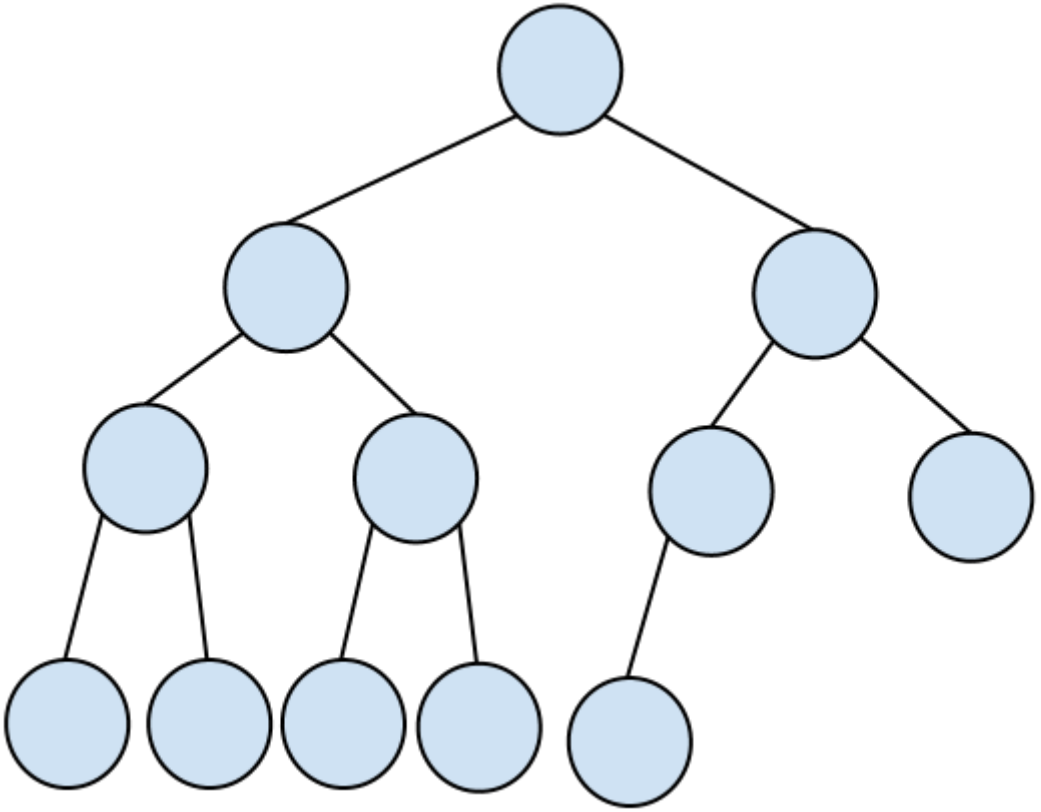
Number of leaves in T=2^3=8.

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:

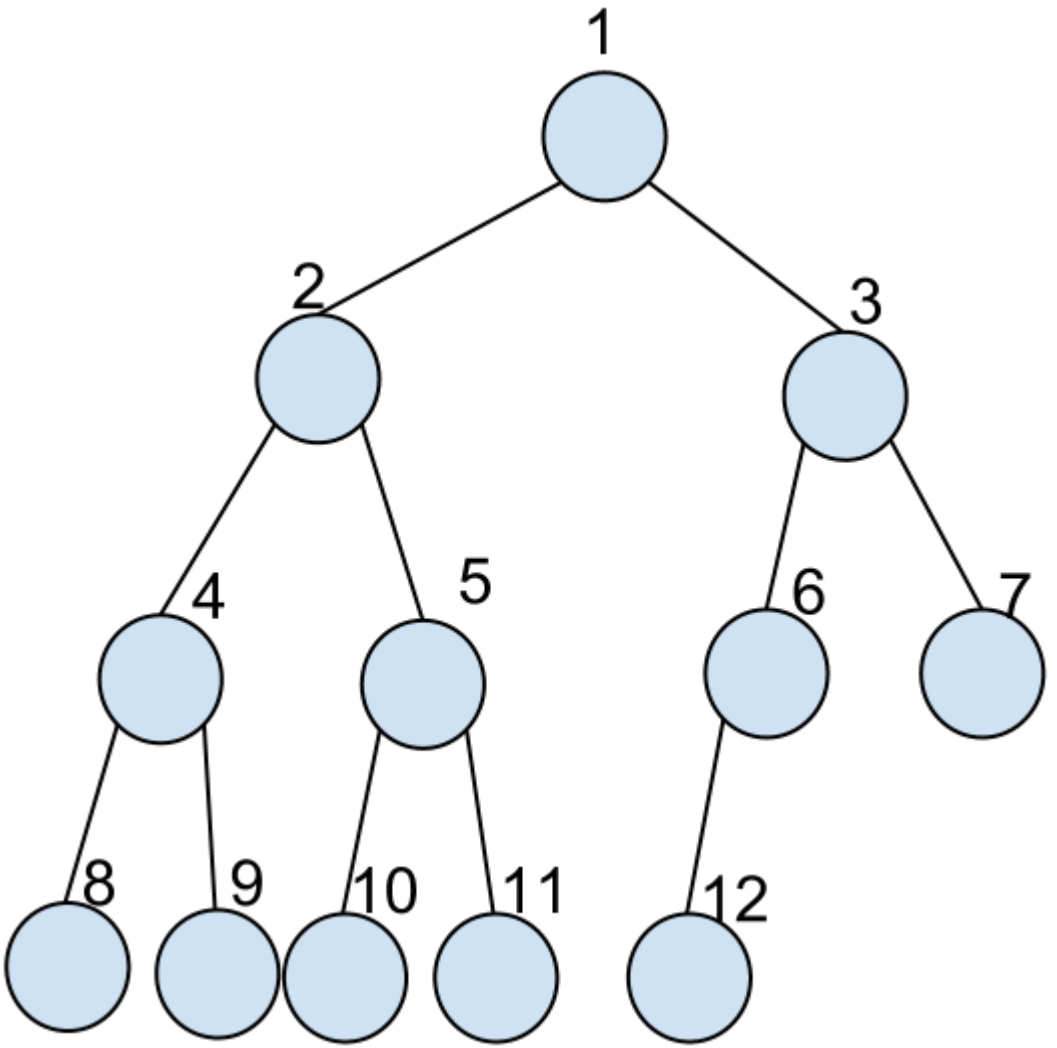
Root is A[1].

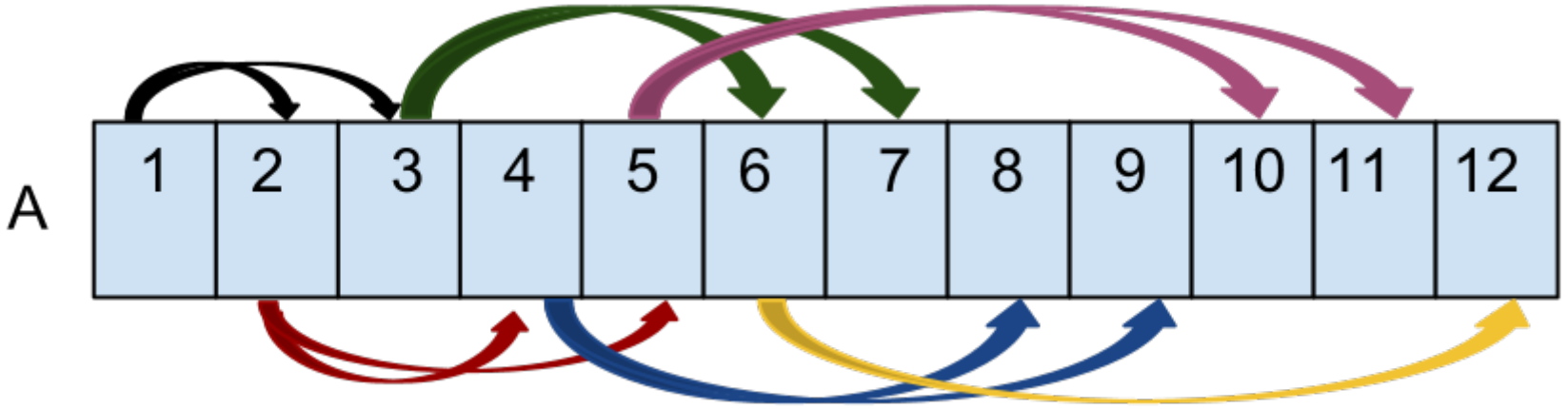
Given index i to a node:

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

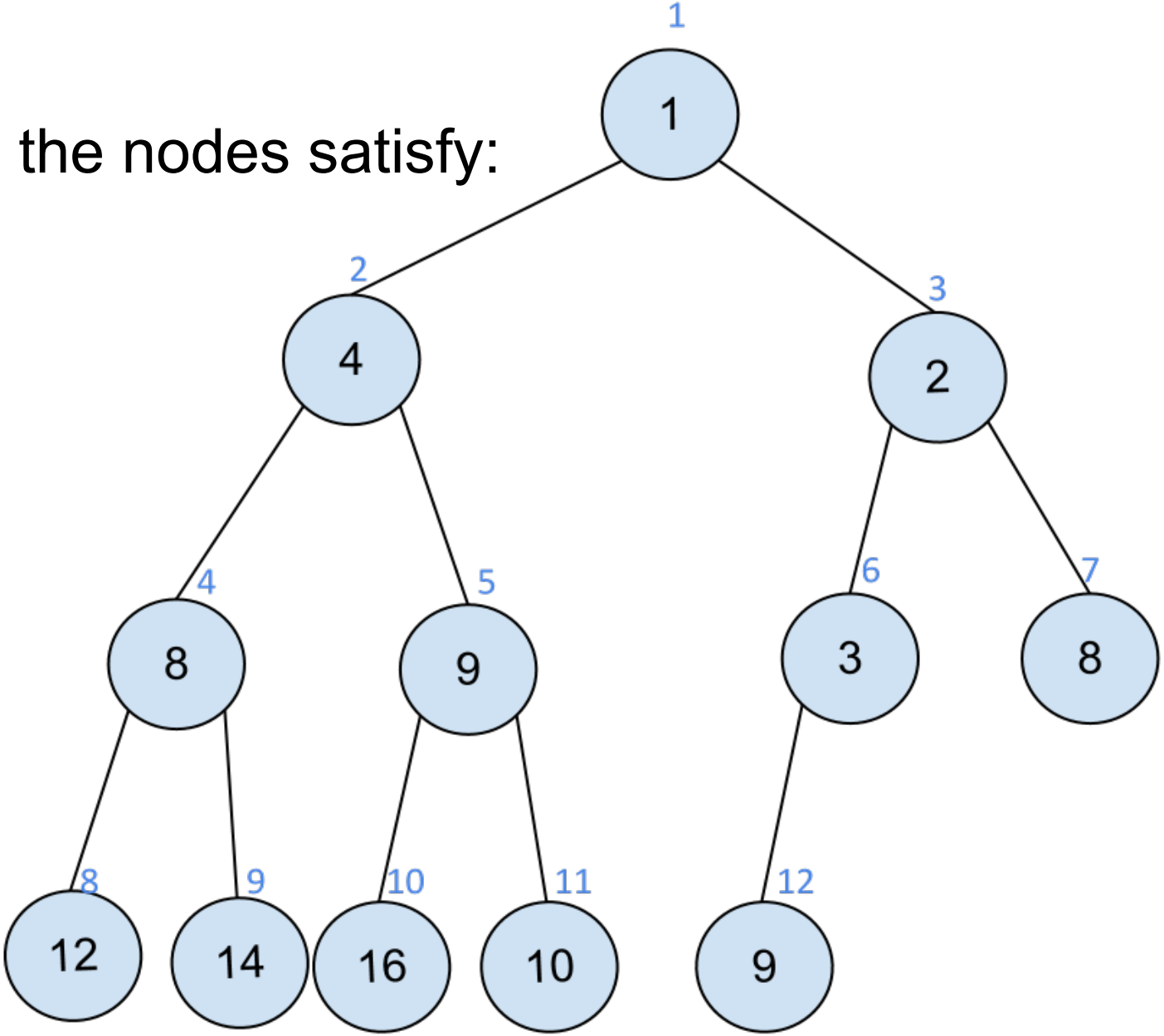
The same results hold for extraction of maximum

We focus on minimum for concreteness.

Min-heap

The values stored in the nodes satisfy:

A[Parent(i)] ≤ A[i].



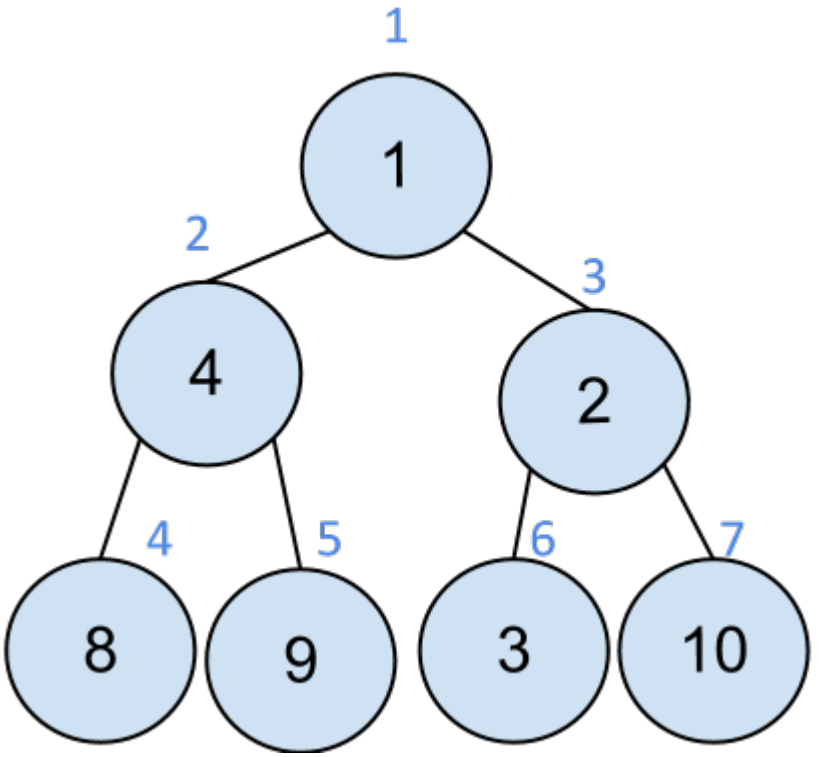
	1	2	3	4	5	6	7	8	9	10	11	12
A	1	4	2	8	9	3	8	12	14	16	10	9

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;
```



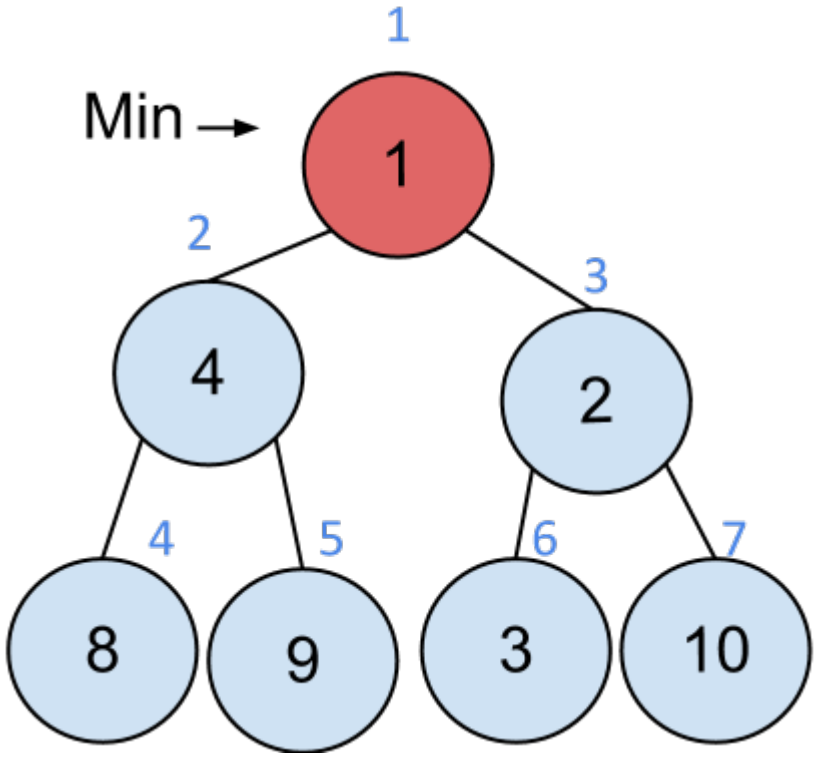
Let's see the steps

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;

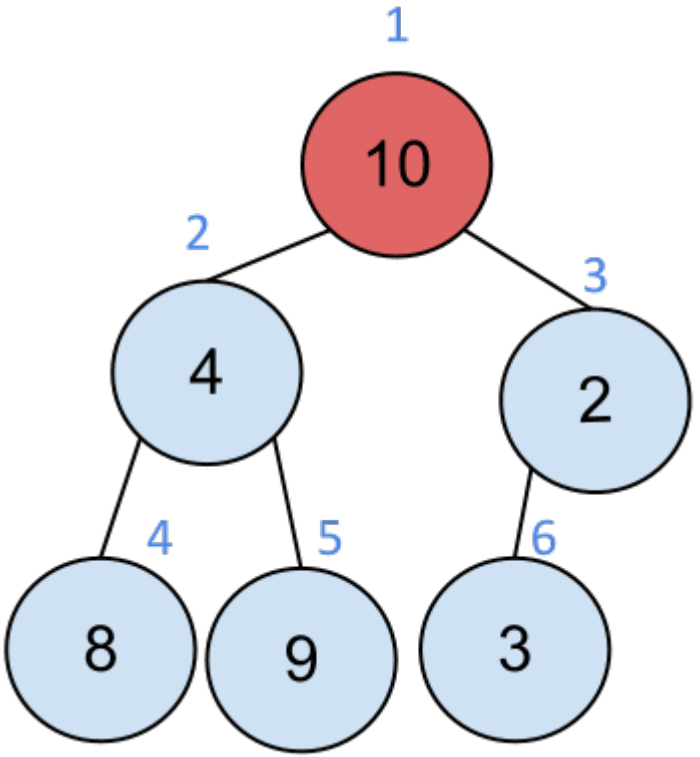


Extracting the minimum element

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;

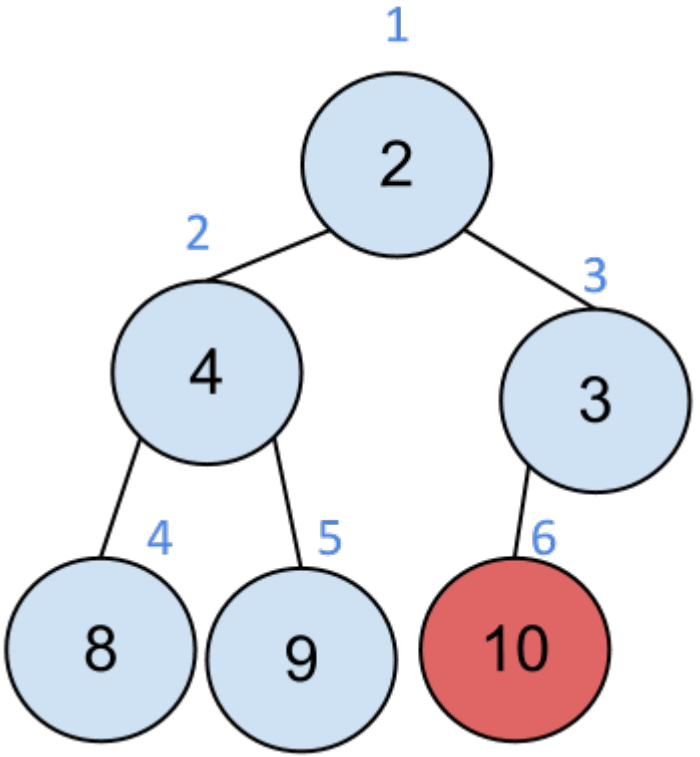


Extracting the minimum element

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;



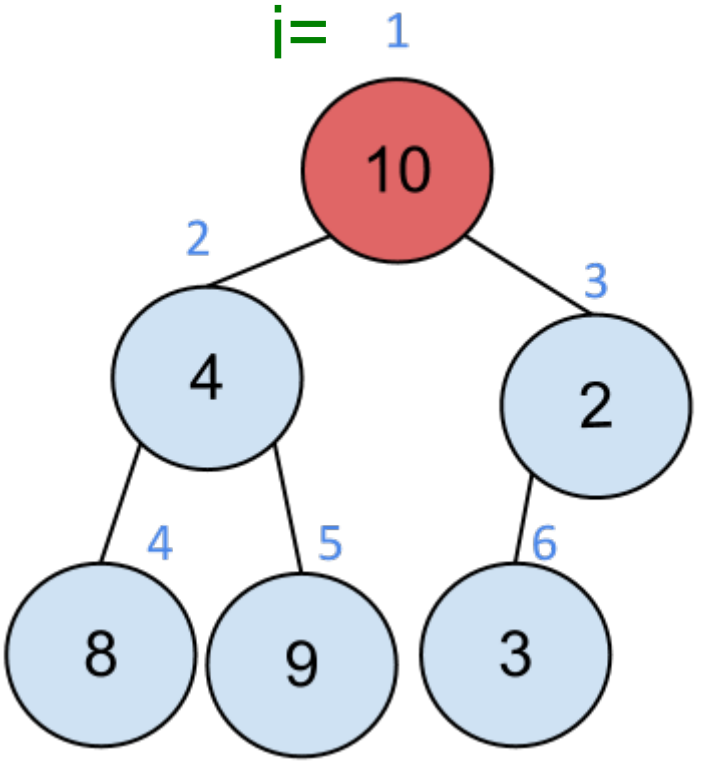
Min-heapify is a function that restores the min property

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)
 }
```

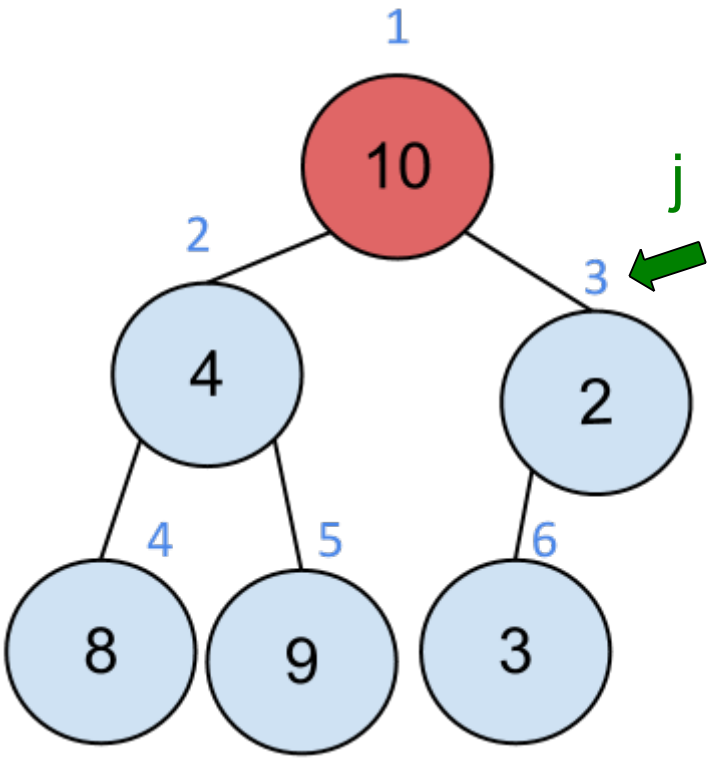


Min-heapify restores the min-heap property

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)
 }
```

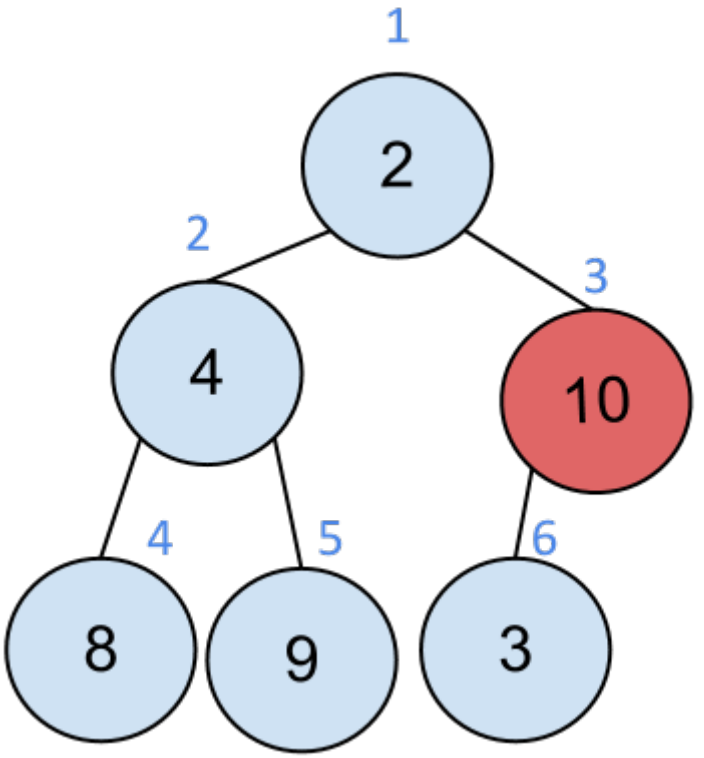


Min-heapify restores the min-heap property

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)
 }
```

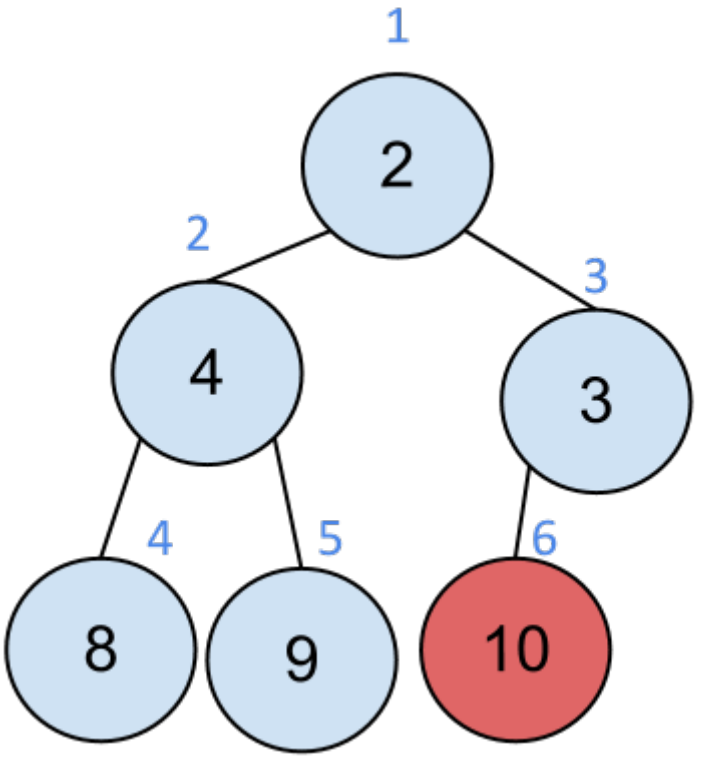


Min-heapify restores the min-heap property

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)
 }
```

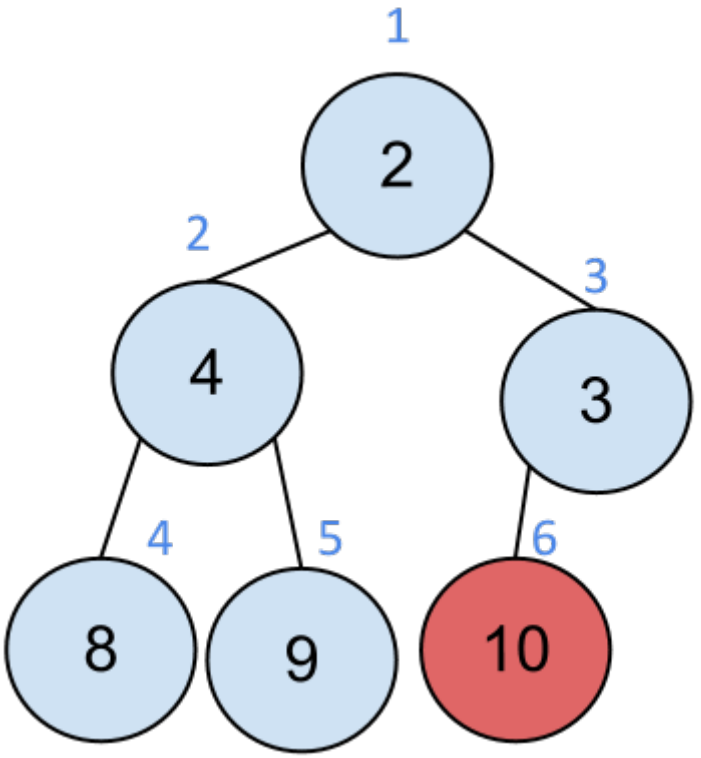


Min-heapify restores the min-heap property

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)
 }
```



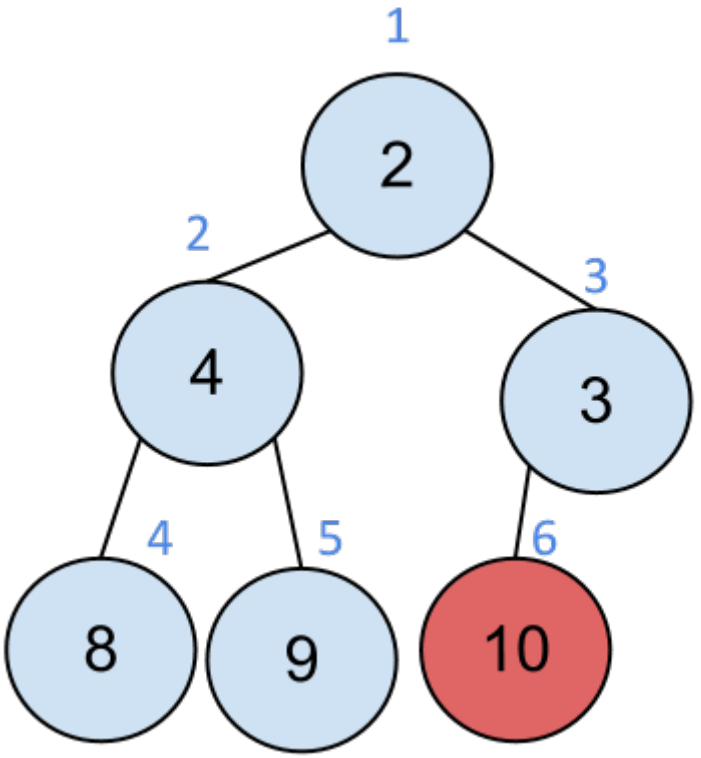
Running time = ?

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

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

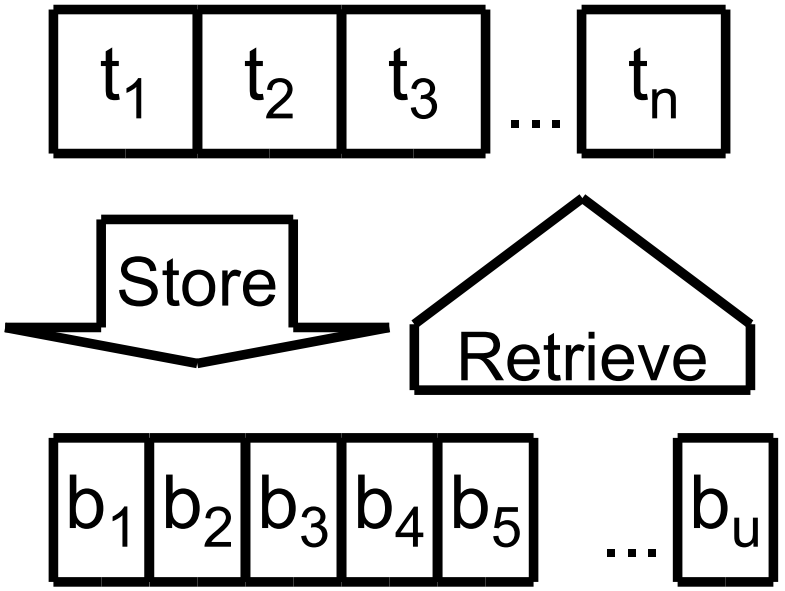
$= O(n)$

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 $\left(\text{optimal} = \lceil n \lg_2 3 \rceil\right)$

  Fast retrieval: Get t_i by probing few bits $\left(\text{optimal} = 2\right)$

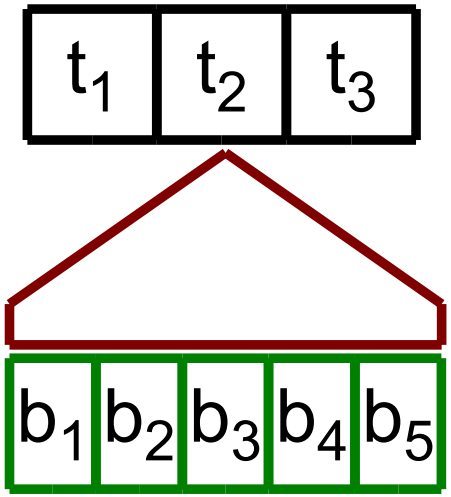
Two solutions

- Arithmetic coding:

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

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

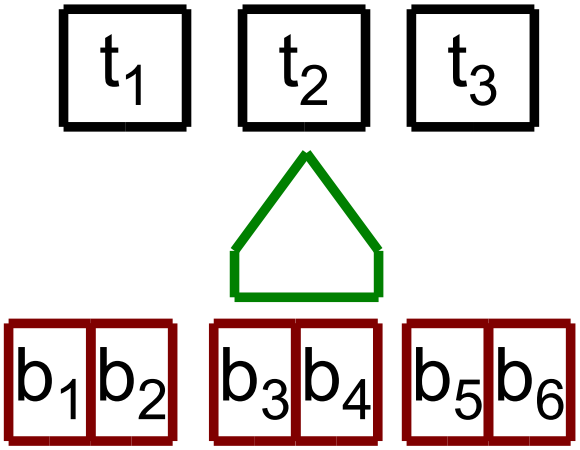
  Bad retrieval: To get t_i probe all > n bits



- Two bits per trit

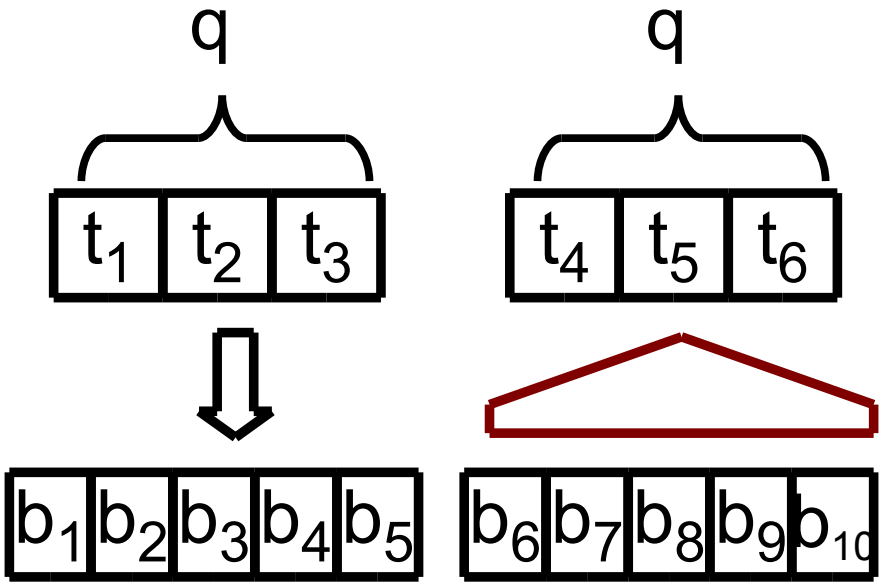
  Bad space: n·2

  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 $O(\lg n)$