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 $\leq$ 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?

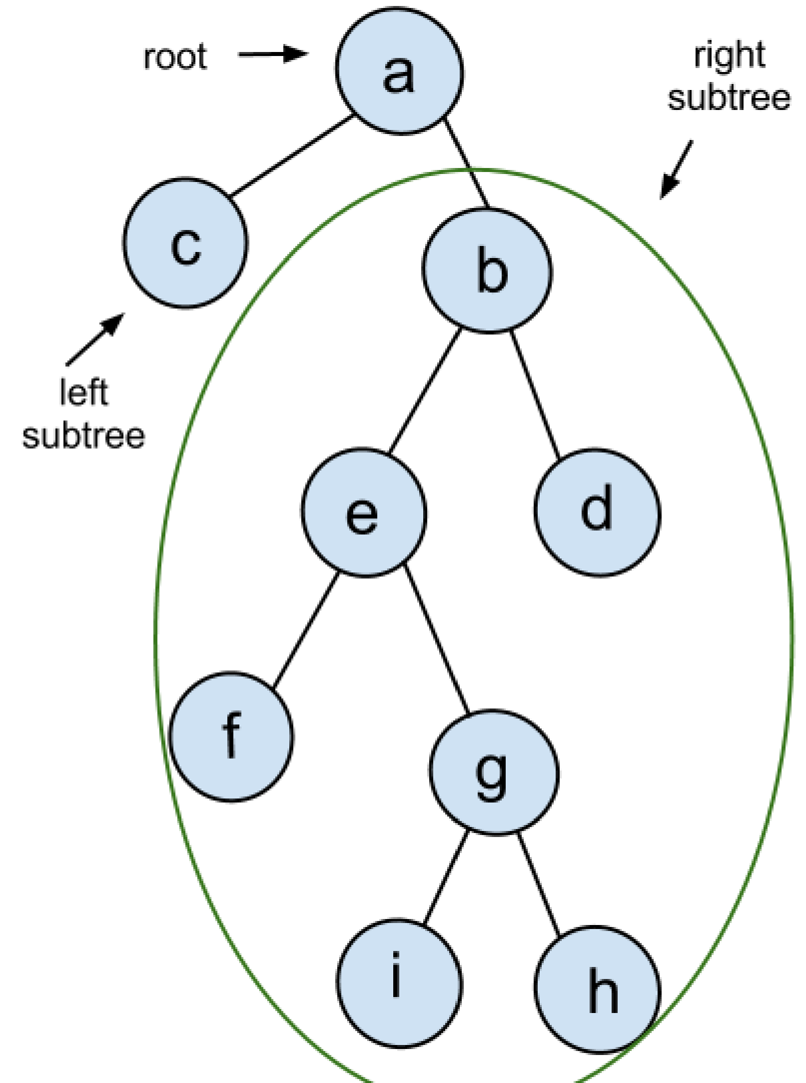
- What if numbers are not small?

- There exist a number of data structure that support each operation in O(log n) time
- AVL tree, red-black tree, etc.

- These data structures organize data in a tree

Binary tree is a graph whose vertices V can be divided in three disjoint sets: root, left sub-tree, and right sub-tree

Alternatively: connected graph without cycles



- Example

V={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

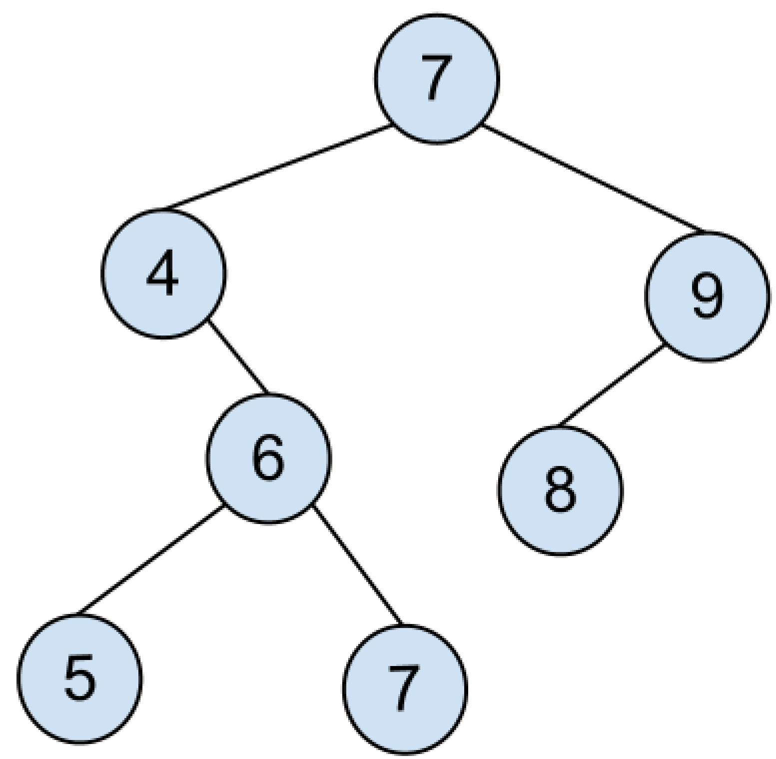
= {c, f, i, h, d}

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 are always stored in such way to 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 $y \Longrightarrow key(x) < key(y)$.

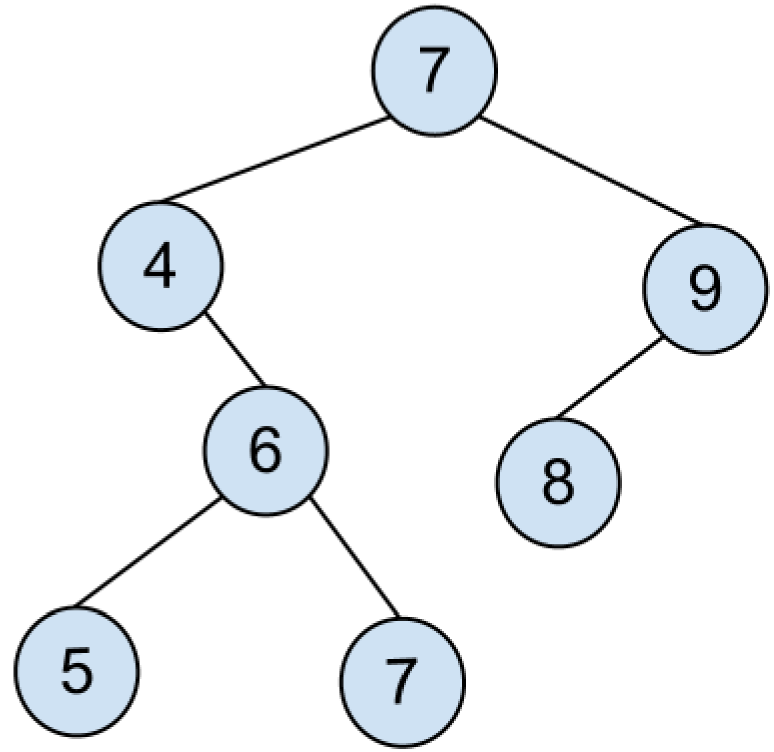
Example:



Binary Search

Looking for k in tree T given root x:

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



Note: NIL stands for empty tree

Running time = the depth of the tree. = $O(n)$, = $\Omega(\log n)$

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 shortly)

Trees make it easier to think about inserting and removing.

Insert x in a tree:

Search (x).

If x not found, create new node with x where x should have been.

To maintain the tree balanced, we perform rotations

Delete x from a tree:

Search (x).

Remove the node with x.

To maintain the tree balanced, we perform rotations

Time O(log n) for both.

DEMO

Next:

More about trees vs. arrays, and heaps.

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

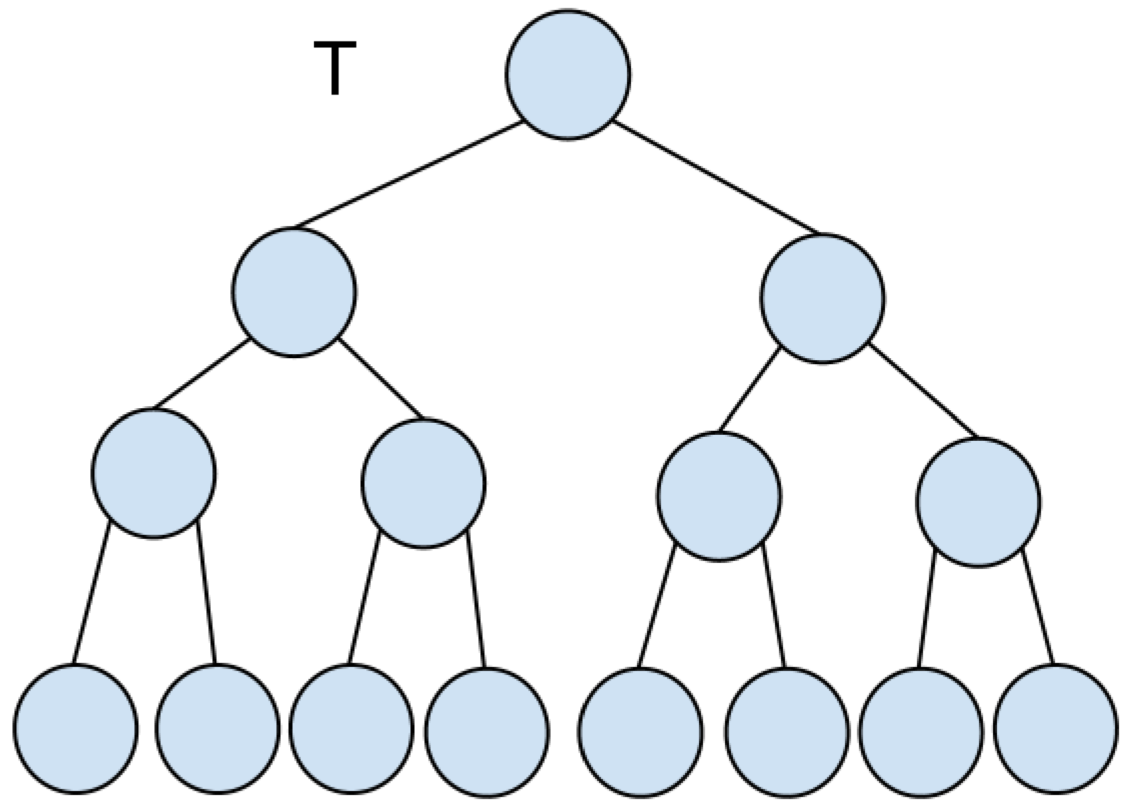
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

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.

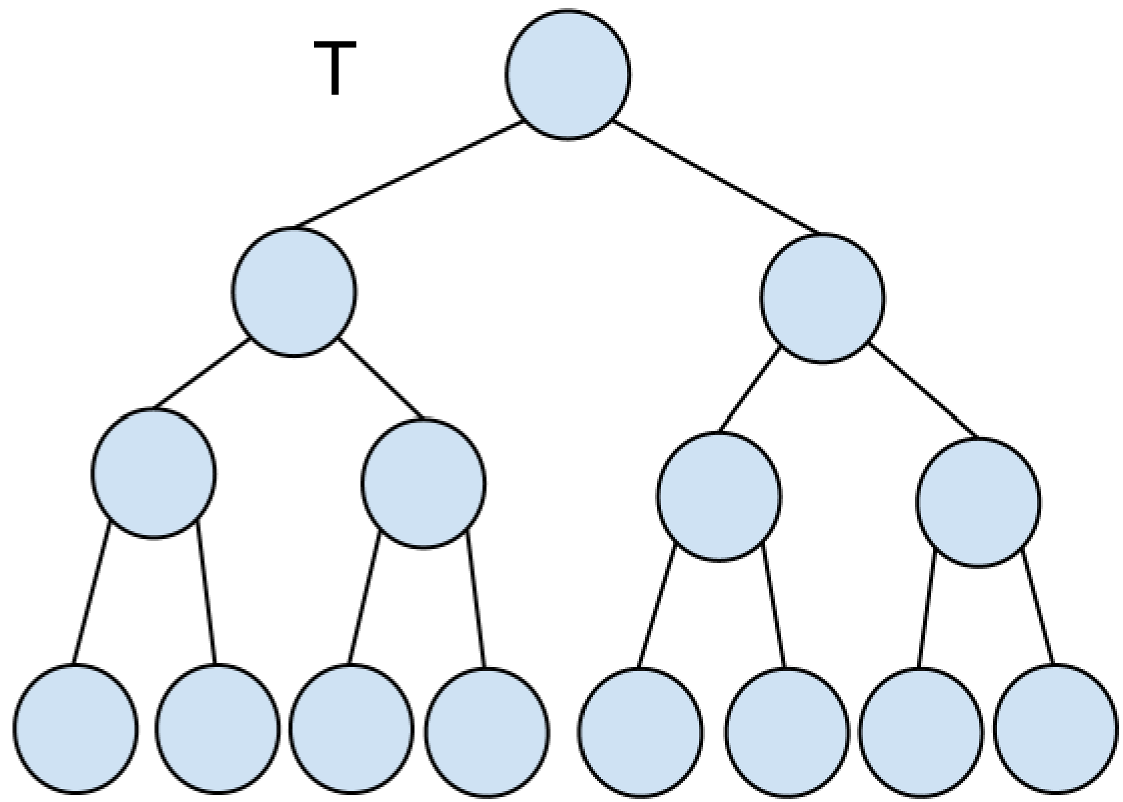
A complete binary tree of depth d has 2^d leaves and $2^{d+1}-1$ nodes.

Example:

Depth of T=3.

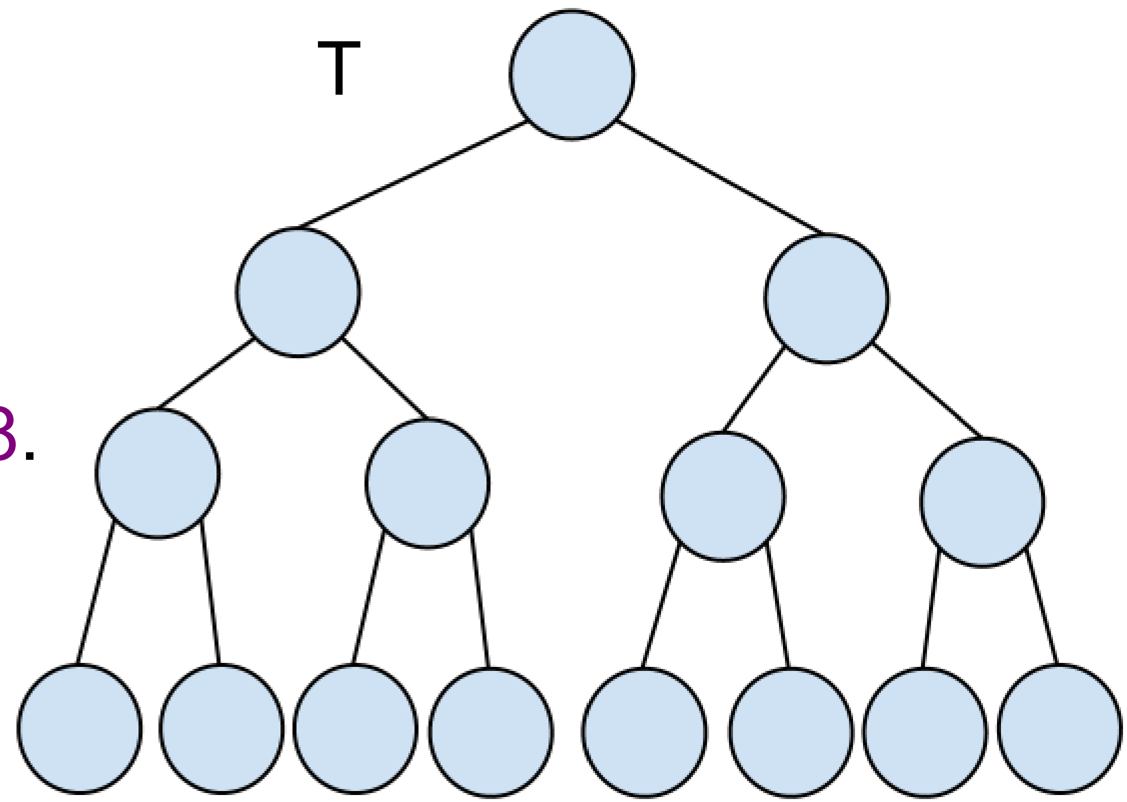
Number of leaves in T=?

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:

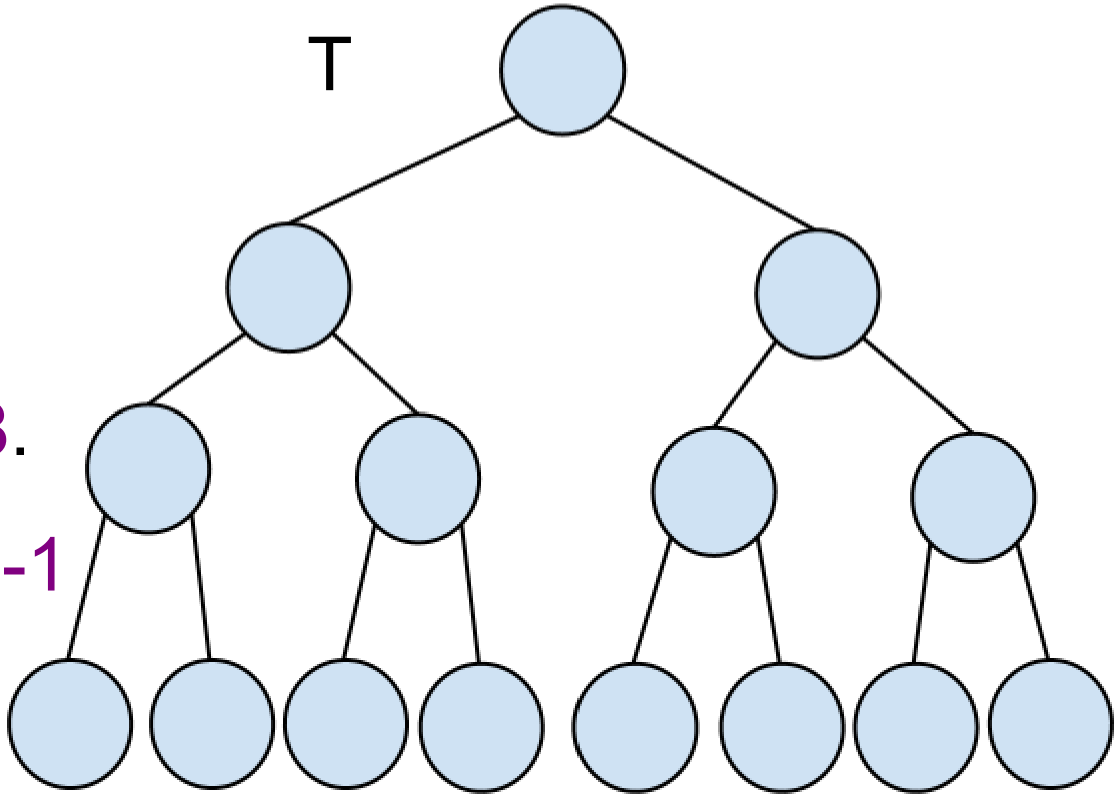
Depth of T=3.

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.

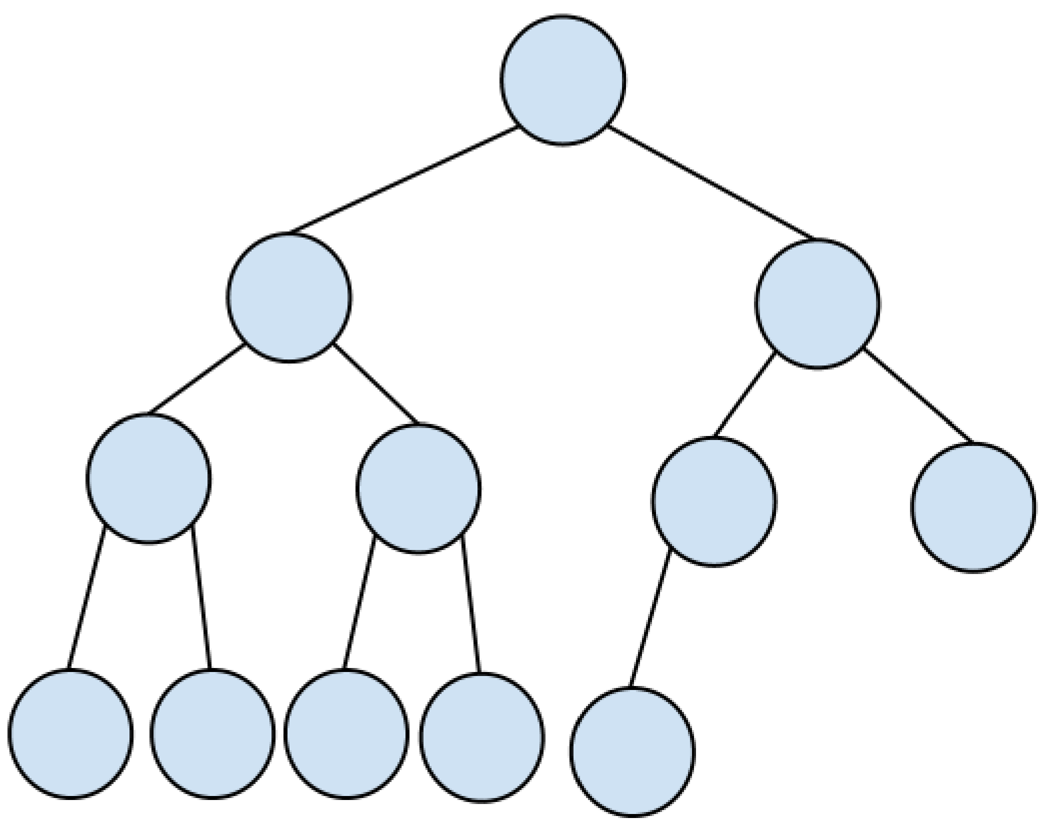
Number of leaves in T=$2^3=8$.

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

Heap is an array that can be view as a nearly complete binary tree, only the last level is missing leaves.

Specifically, the last level must be filled from left to right.

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



Navigating a heap:

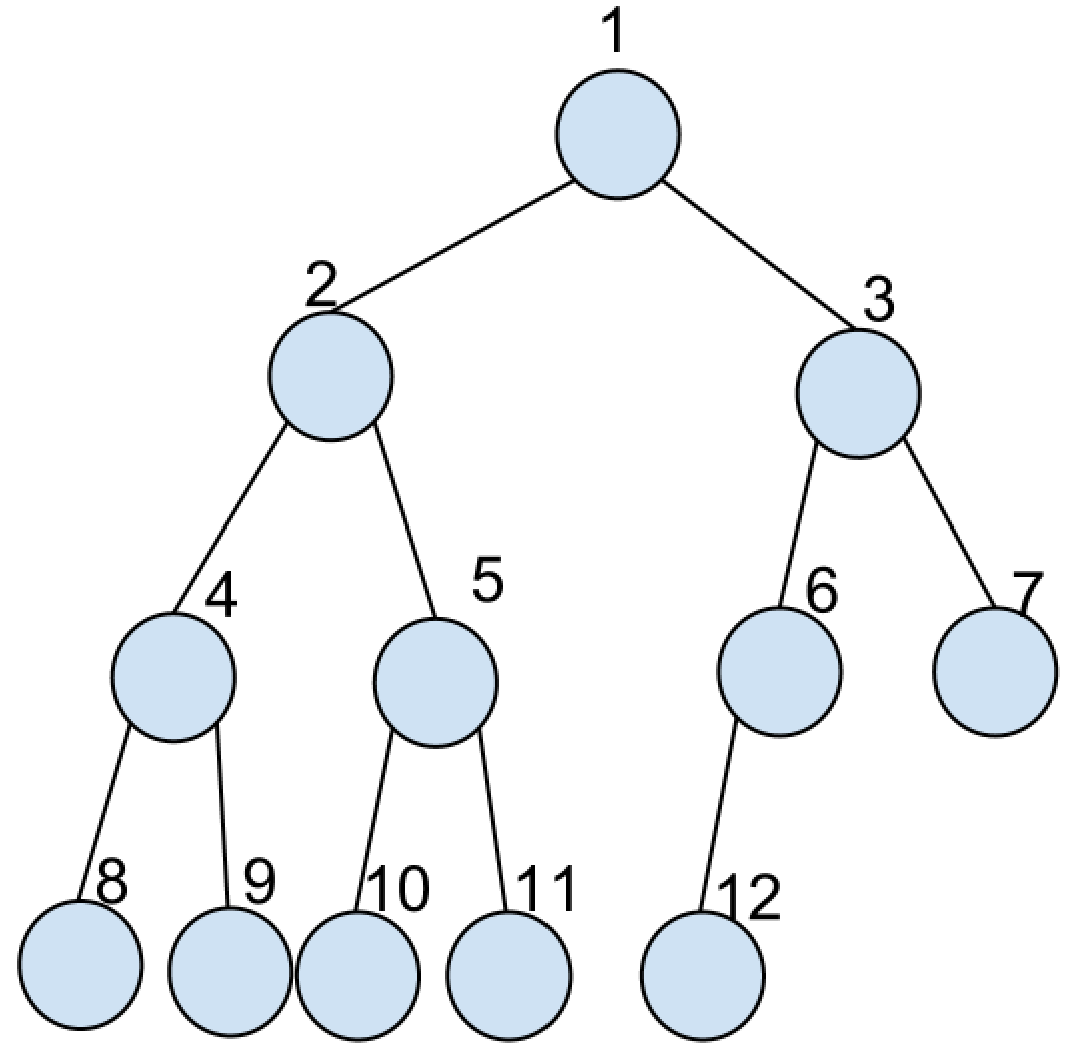
Root is A[1].

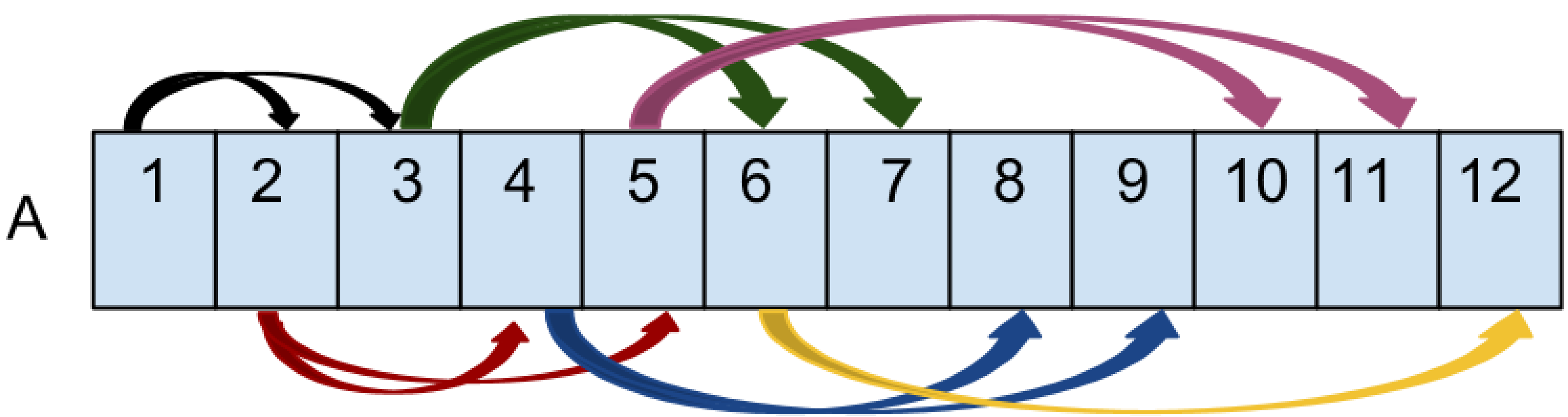
Given index i to a node:

Parent(i) return i/2

Left-Child(i) return 2i

Right-Child(i) return 2i+1



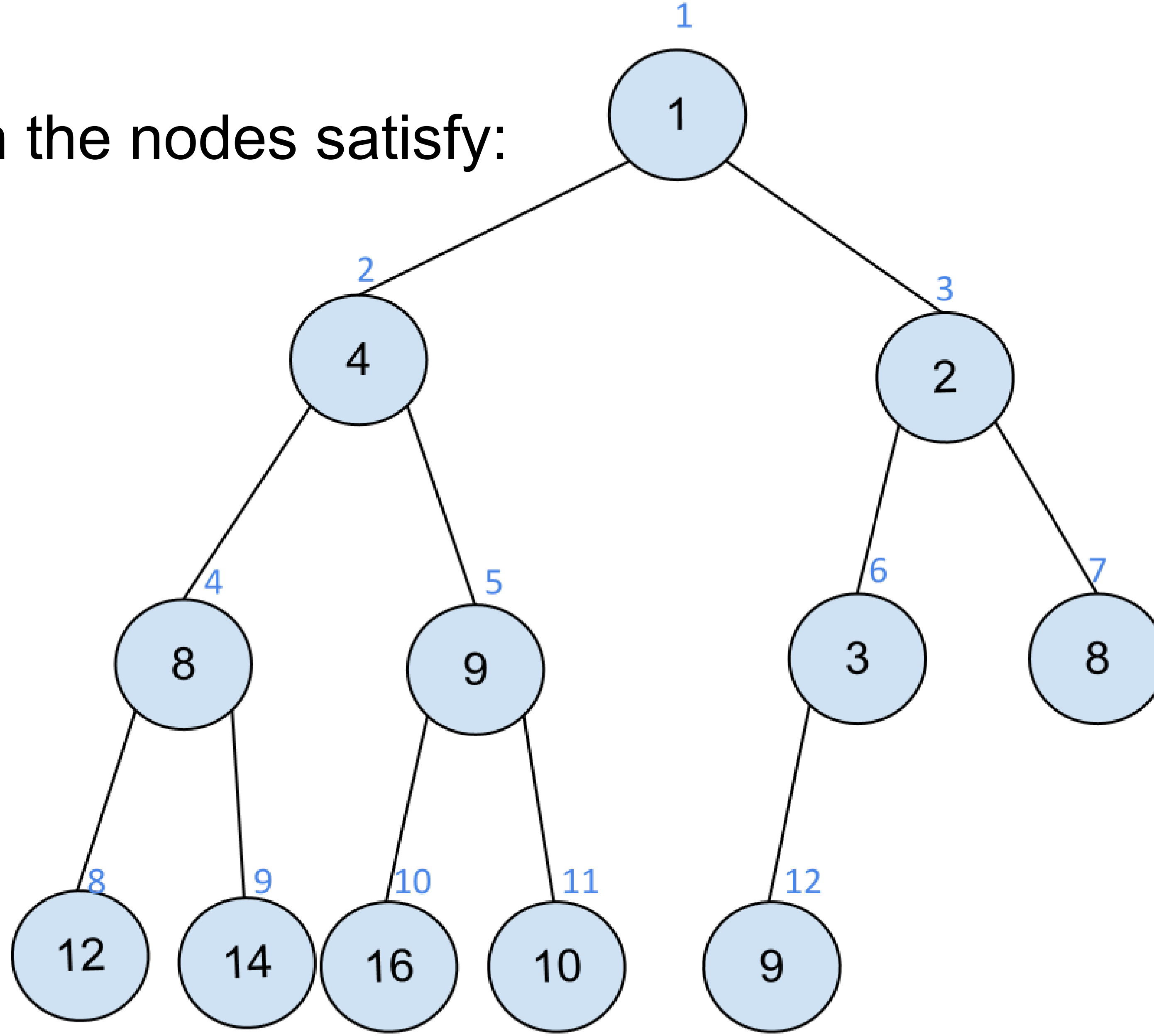


Heaps are useful to dynamically maitain a set of elements while allowing for extraction of Minimum/Maximum element quickly

Min-heap

The values stored in the nodes satisfy:

$A[\text{Parent}(i)] \le A[i]$.

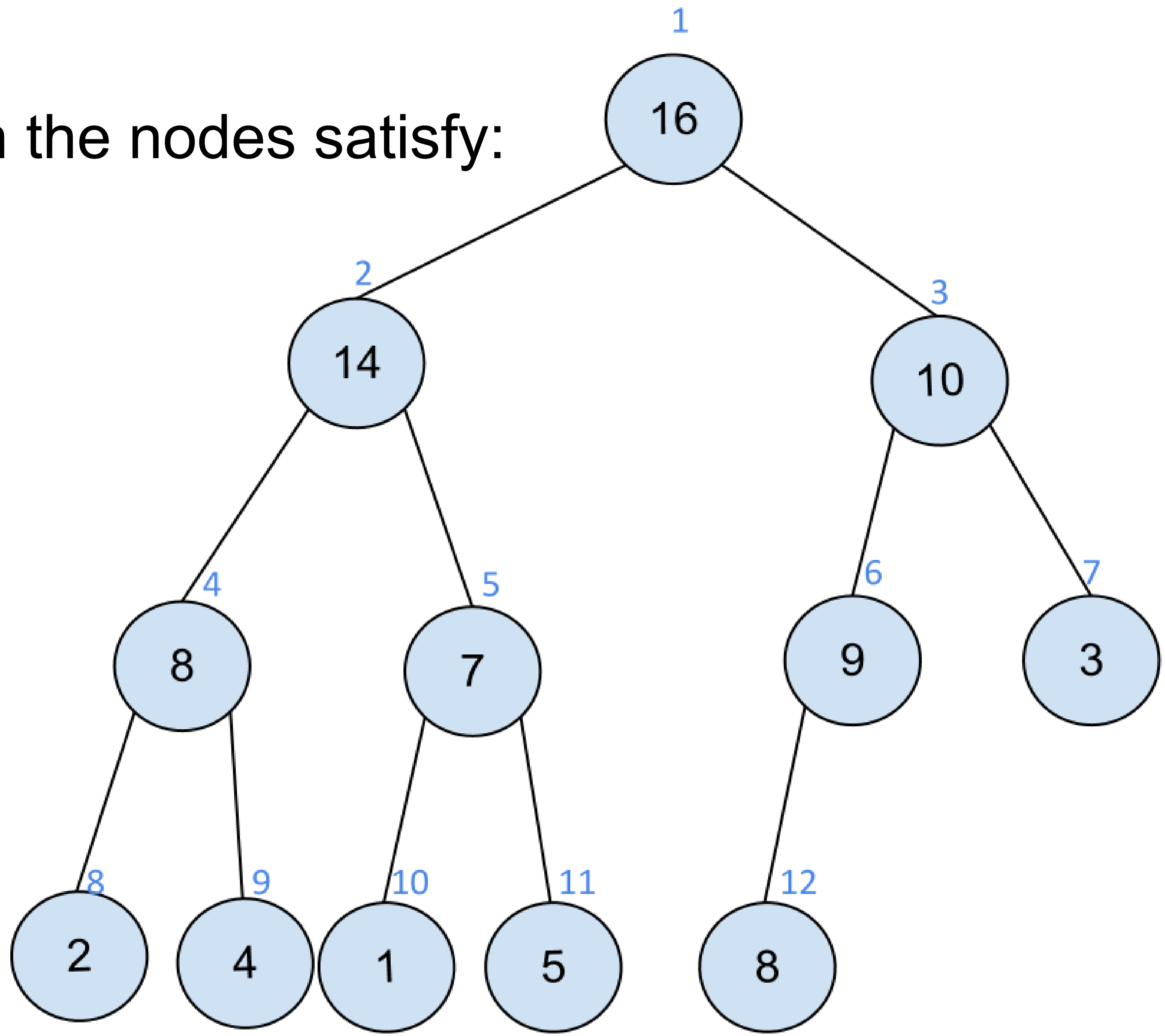


	1	2	3	4	5	6	7	8	9	10	11	12
A	1	4	2	8	9	3	8	12	14	16	10	9

Max-heap

The values stored in the nodes satisfy:

A[Parent(i)] ≥ A[i].



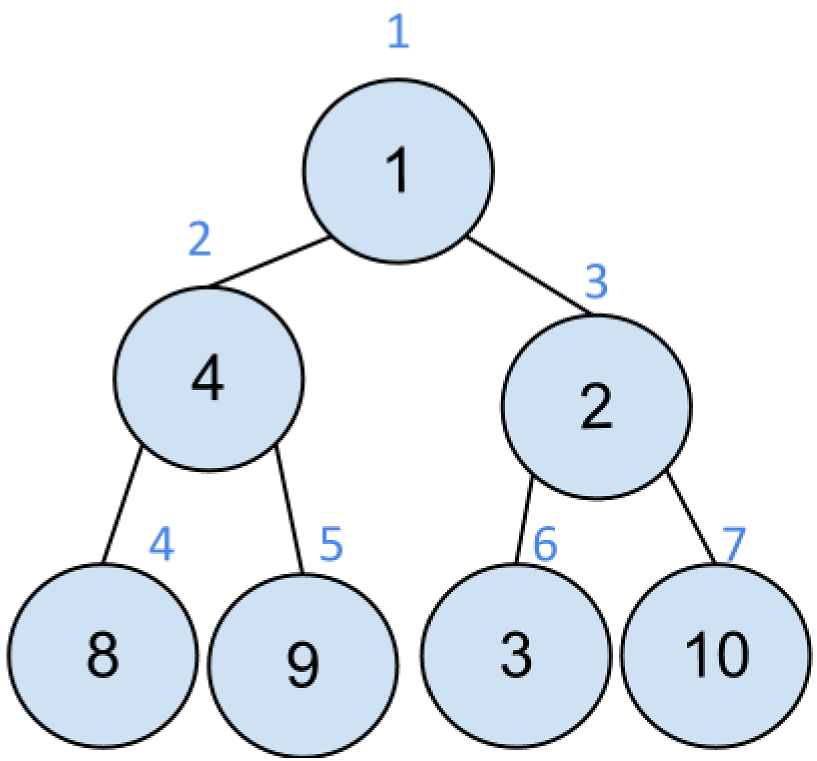
	1	2	3	4	5	6	7	8	9	10	11	12
A	16	14	10	8	7	9	3	2	4	1	5	8

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;
```



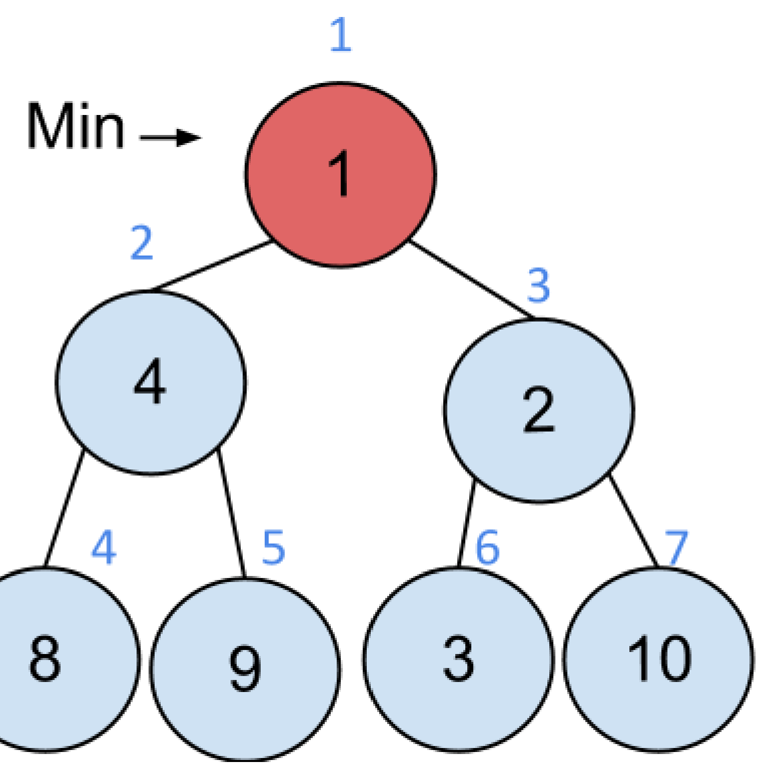
Let's see the steps

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;
```

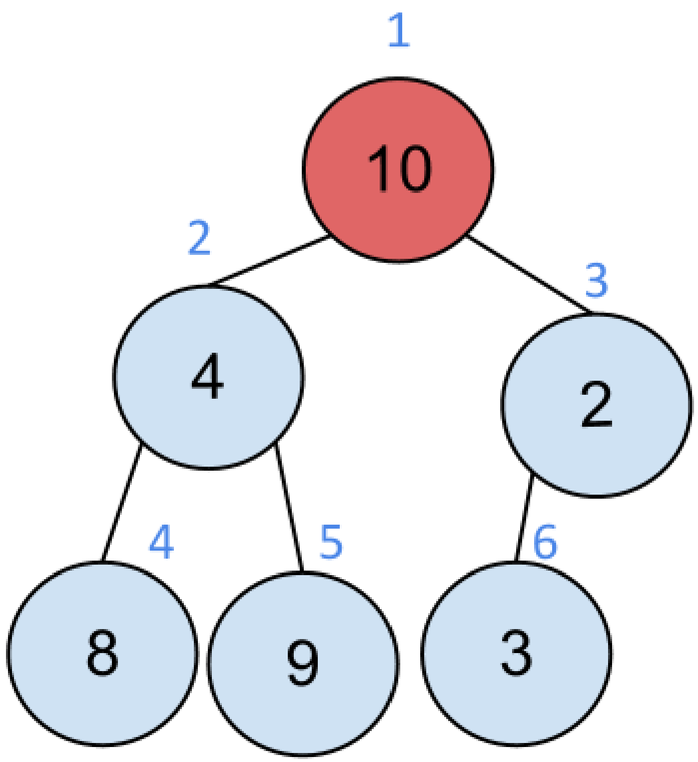


Extracting the minimum element

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;
```

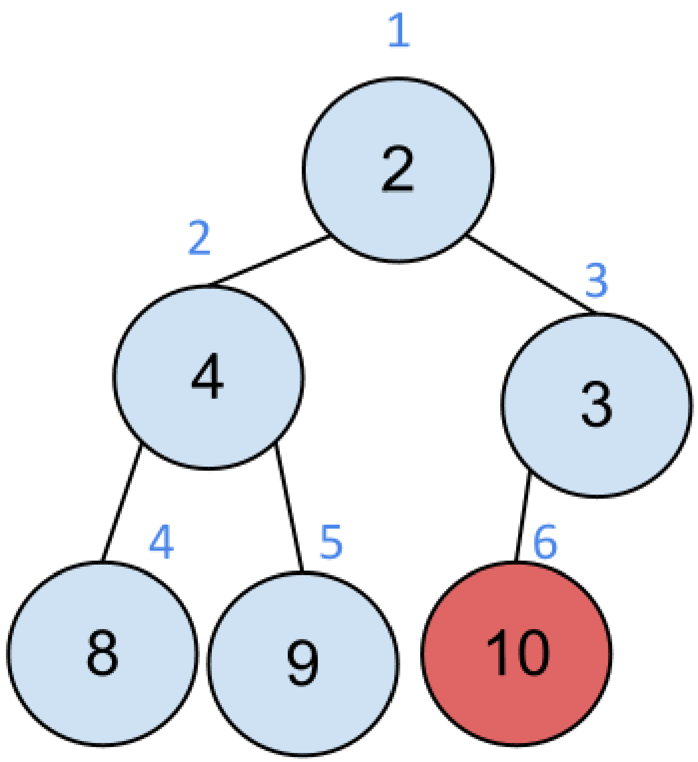


Extracting the minimum element

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 updates the heap so that is maintains the min property

Maintaining a Min-heap

Given array A and index i, the trees rooted at left[i] and Right[i] are Min-heap but A[i] maybe greater than its children.

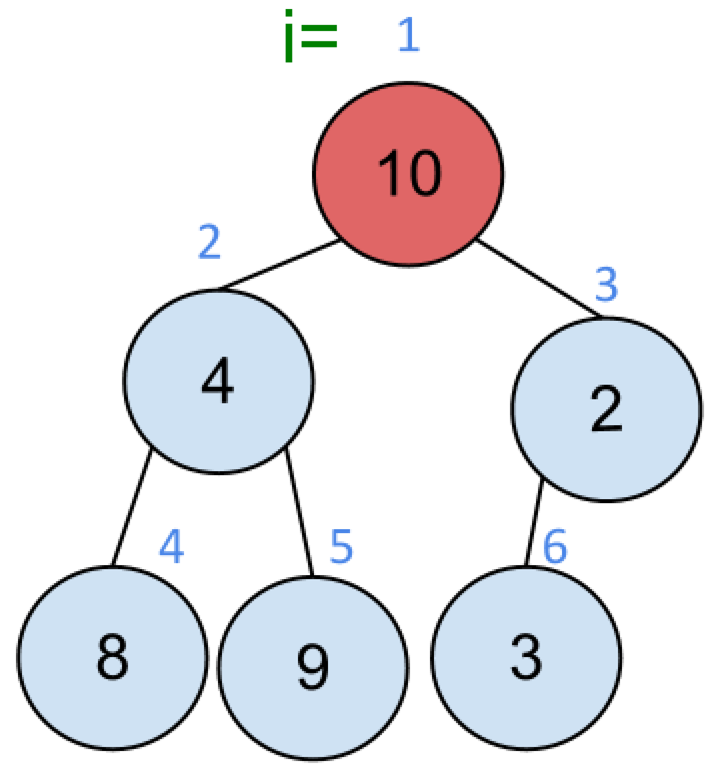
Min-heapify restores the min-heap property.

```
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(A,i)

Let j be the index of smallest node among {A[i], A[Left[i]], A[Right[i]] }

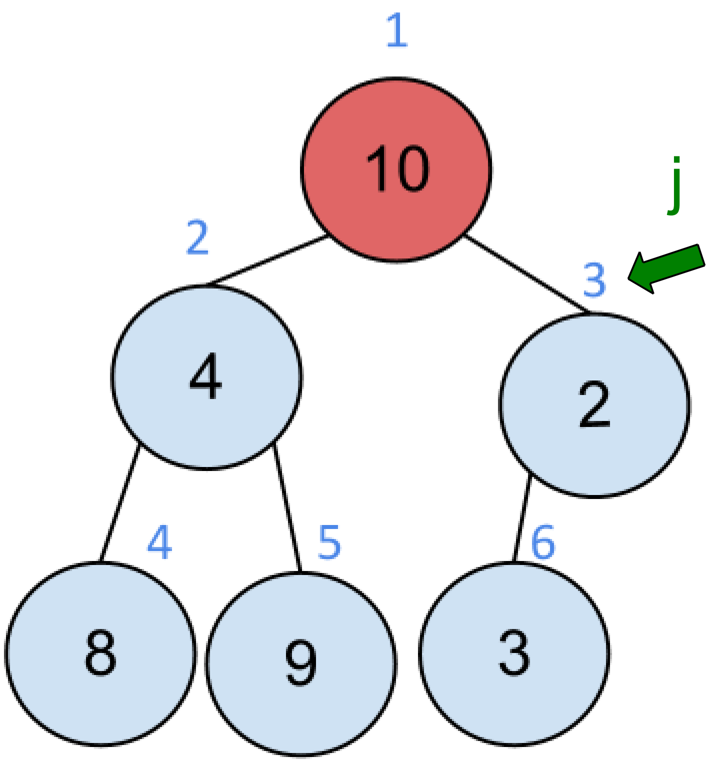
If j ≠ i then {
  exchange A[i] and A[j]
  Min-heapify(A, j)
}



Min-heapify(A,i)

Let j be the index of smallest node among {A[i], A[Left[i]], A[Right[i]] }

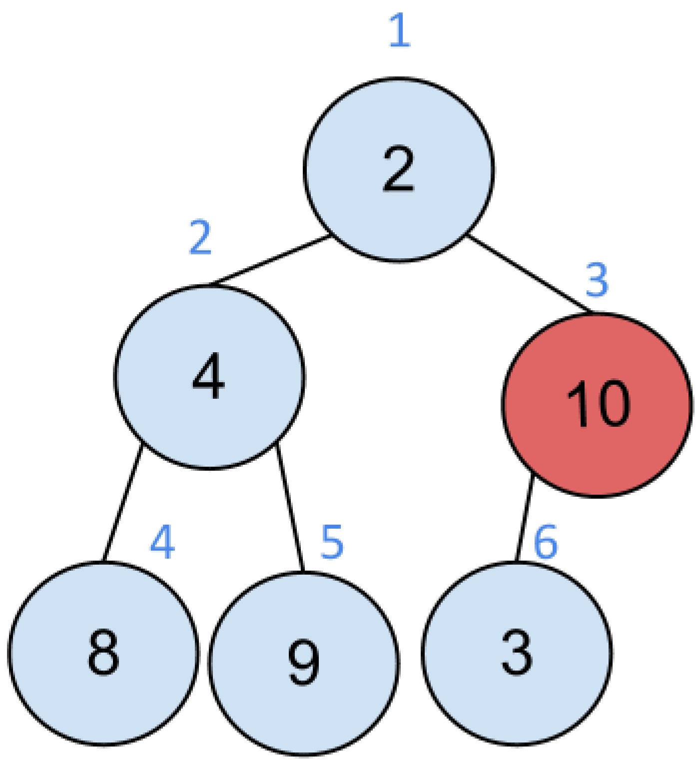
If j ≠ i then {
  exchange A[i] and A[j]
  Min-heapify(A, j)
}



Min-heapify(A,i)

Let j be the index of smallest node
among {A[i], A[Left[i]], A[Right[i]] }

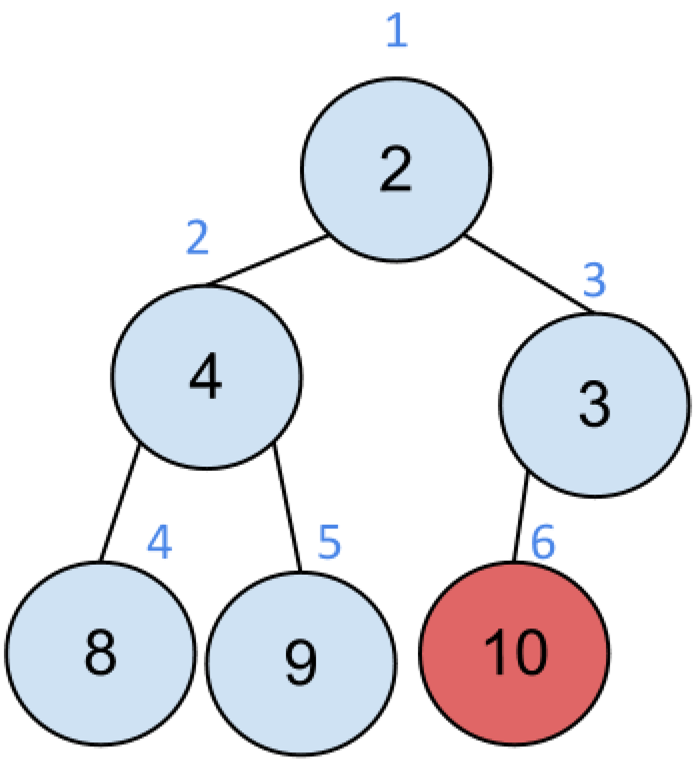
If j ≠ i then {
exchange A[i] and A[j]
Min-heapify(A, j)
}



Min-heapify(A,i)

Let j be the index of smallest node among {A[i], A[Left[i]], A[Right[i]] }

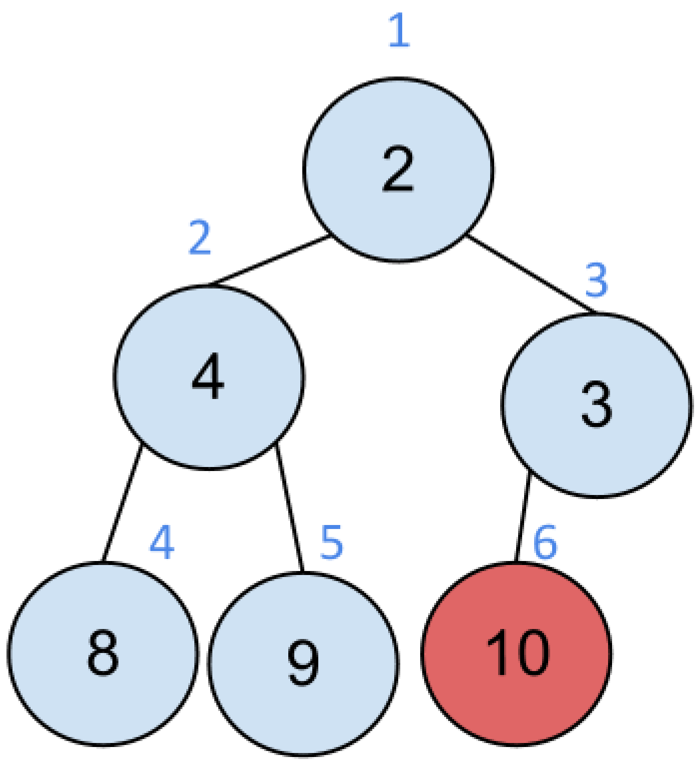
If j ≠ i then {
exchange A[i] and A[j]
Min-heapify(A, j)
}



Min-heapify(A,i)

Let j be the index of smallest node among {A[i], A[Left[i]], A[Right[i]] }

If $j \neq i$ then {
  exchange A[i] and A[j]
  Min-heapify(A, j)
}

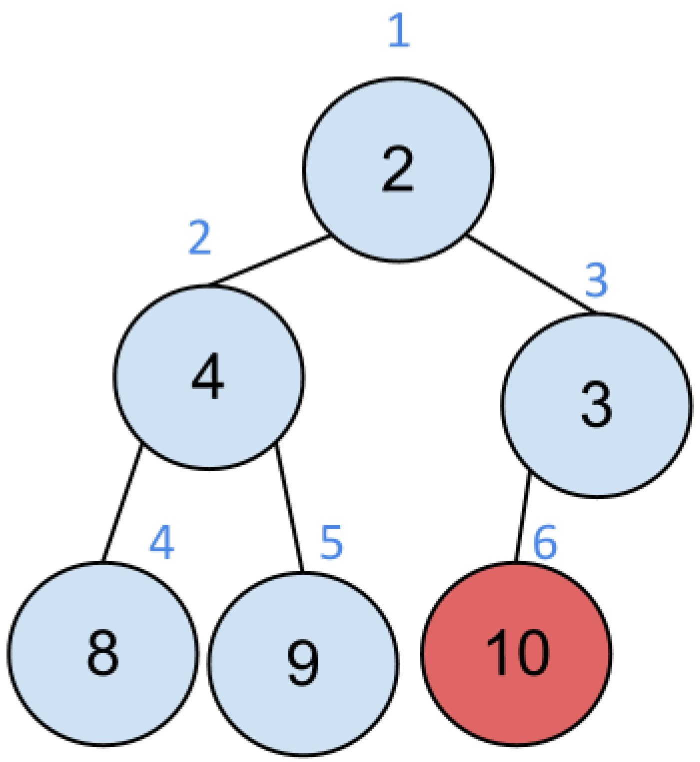


Running time = ?

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)
}



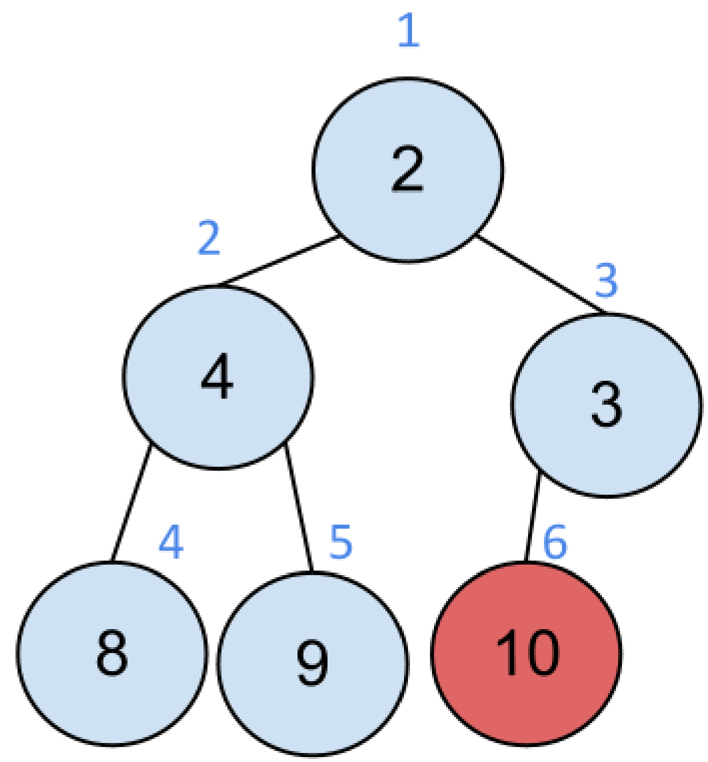
Running time = depth = O(log n)

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;
```



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

Next: Given n elements, how do we initialize a heap?

Building Min-heap

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

Build-Min-heap(A)

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

Building Min-heap

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

Build-Min-heap(A)

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

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

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

$= ?$

Building Min-heap

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

Build-Min-heap(A)

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

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

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

$= O(n)$

Insert-Min-heap

```
insert-Min-heap(A, key)

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

 i:= heap-size[a];

 While i>0 and A[parent(i)] > A[i]) {
     exchange(A[parent(i)], A[i])
      i:= parent[i]
 }
```

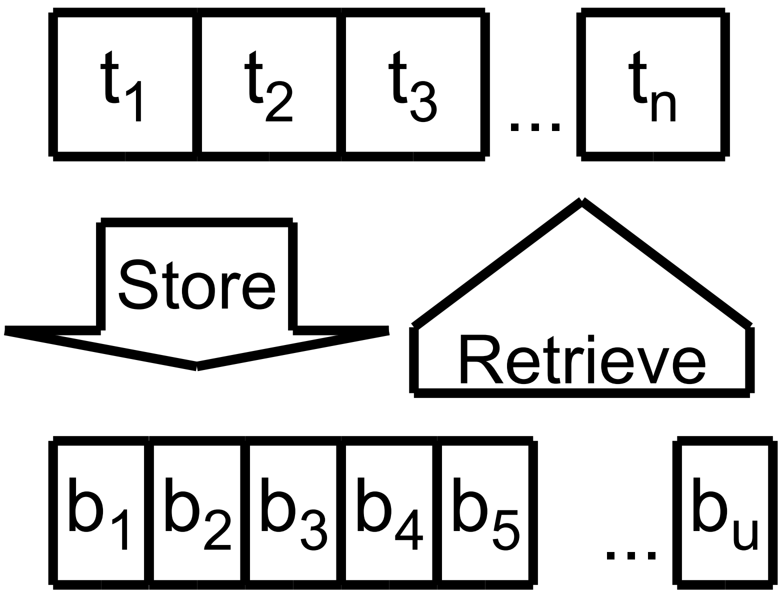
Running time = O(log n).

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 $\left(\text{optimal} = 2\right)$

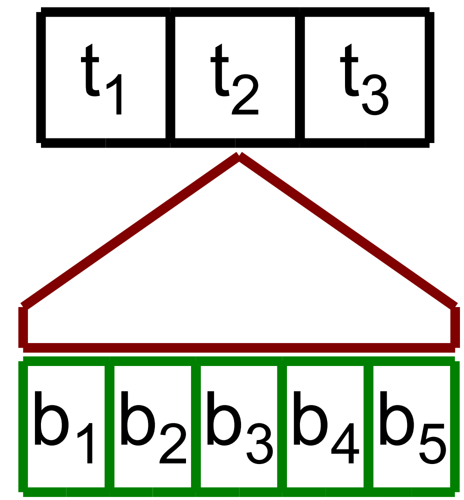
Two solutions

- Arithmetic coding:

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

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

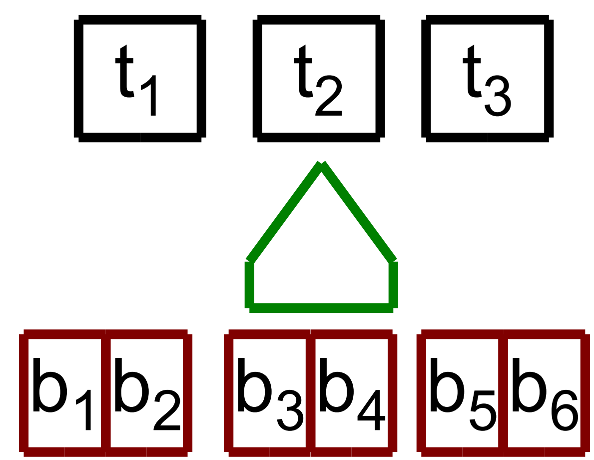
  Bad retrieval: To get t_i probe all > n bits



- Two bits per trit

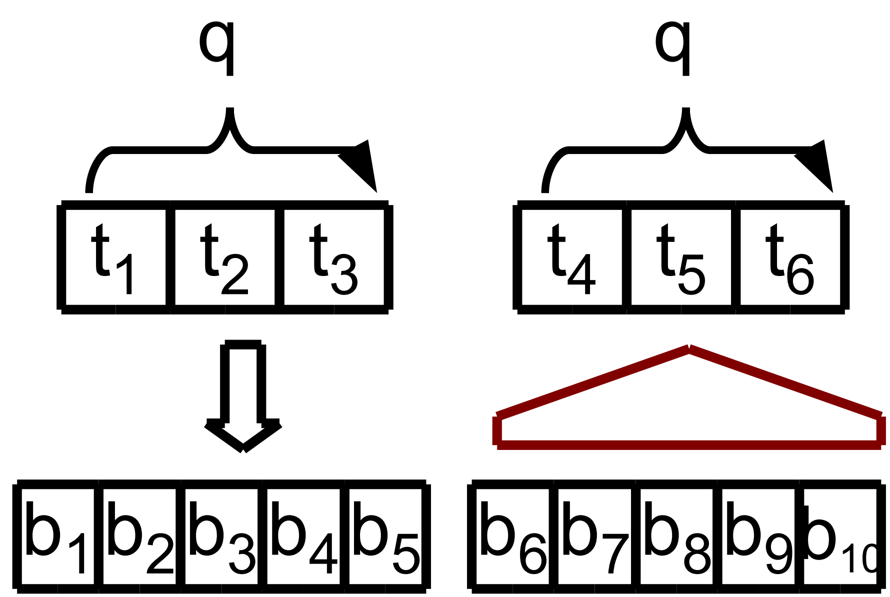
  Bad space: $n \cdot 2$

  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