Trees

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships"

-Linus Torvalds

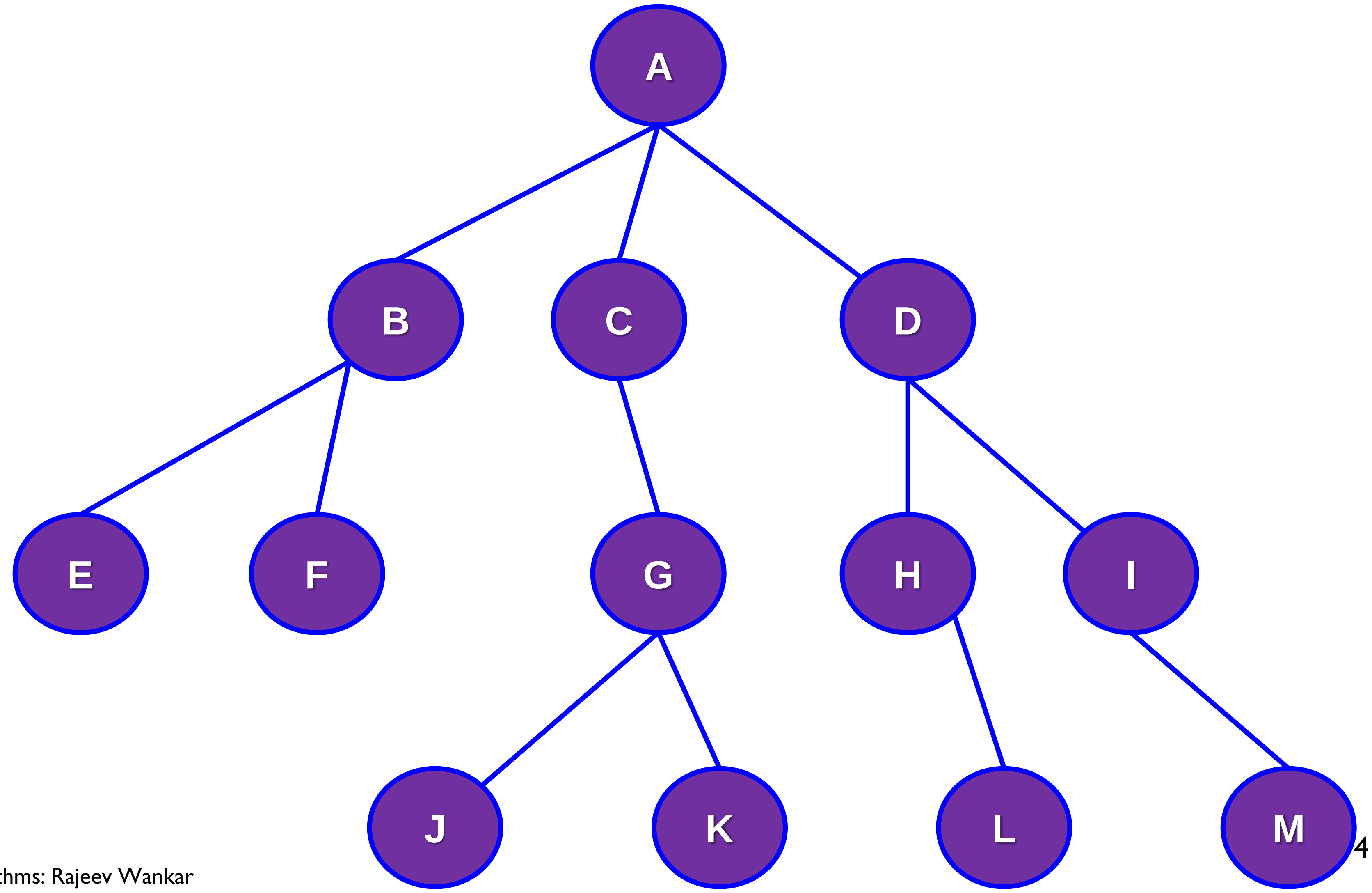
What you have to start from today

- Striver's DSA Sheet (https://takeuforward.org/strivers-a2z-dsa-course/strivers-a2z-dsa-course-sheet-2)
- https://cses.fi/problemset/
- https://leetcode.com/problemset/
- https://codeforces.com/

Tree - Definition

- A ***tree*** T is a finite set of one or more nodes such that there is one specially designated node r called the root of T, and the remaining nodes in $(T - \{ r \})$ are partitioned into $n \geq 0$ disjoint subsets $T_1, T_2, \ldots, T_n$, each of which is a sub tree, and whose roots $r_1, r_2, \ldots, r_n$, respectively, are children of r.

Tree

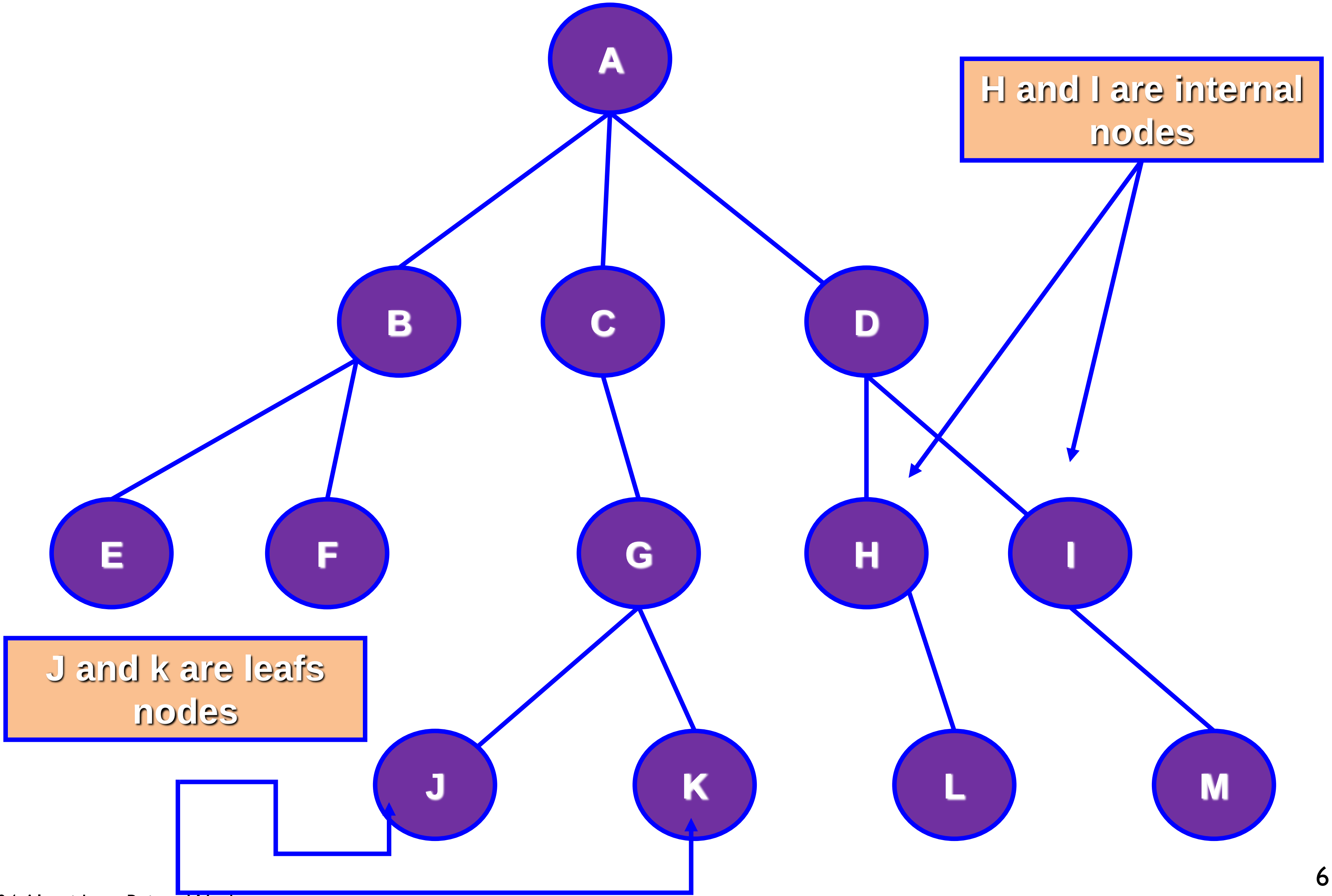


Degree: Number of subtrees of a node

Leaf of terminal: Degree is zero

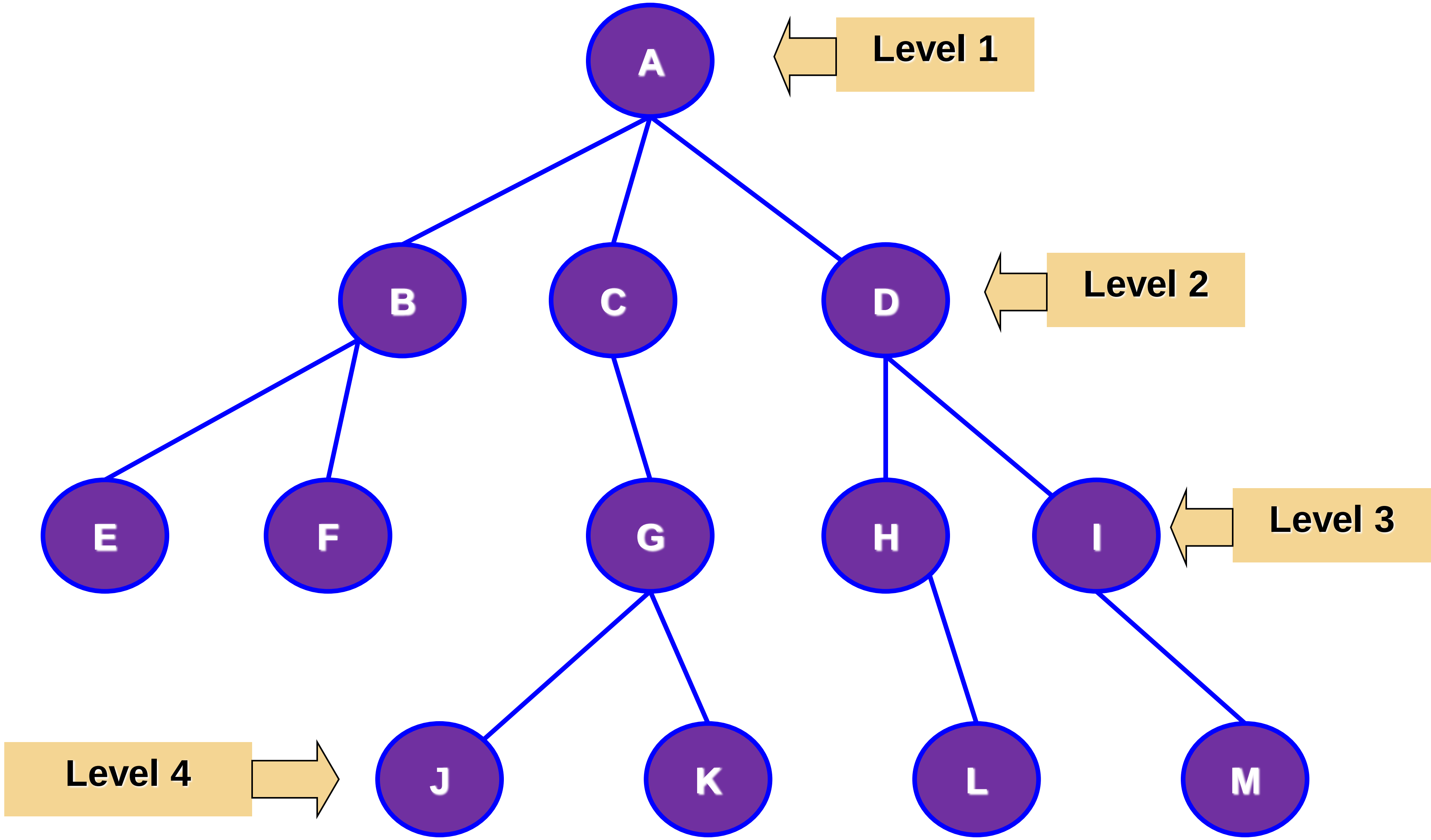
Siblings: Children of the same parent

Forest: It is a set of $n \geq 0$ disjoint trees (after removing root we get forest)

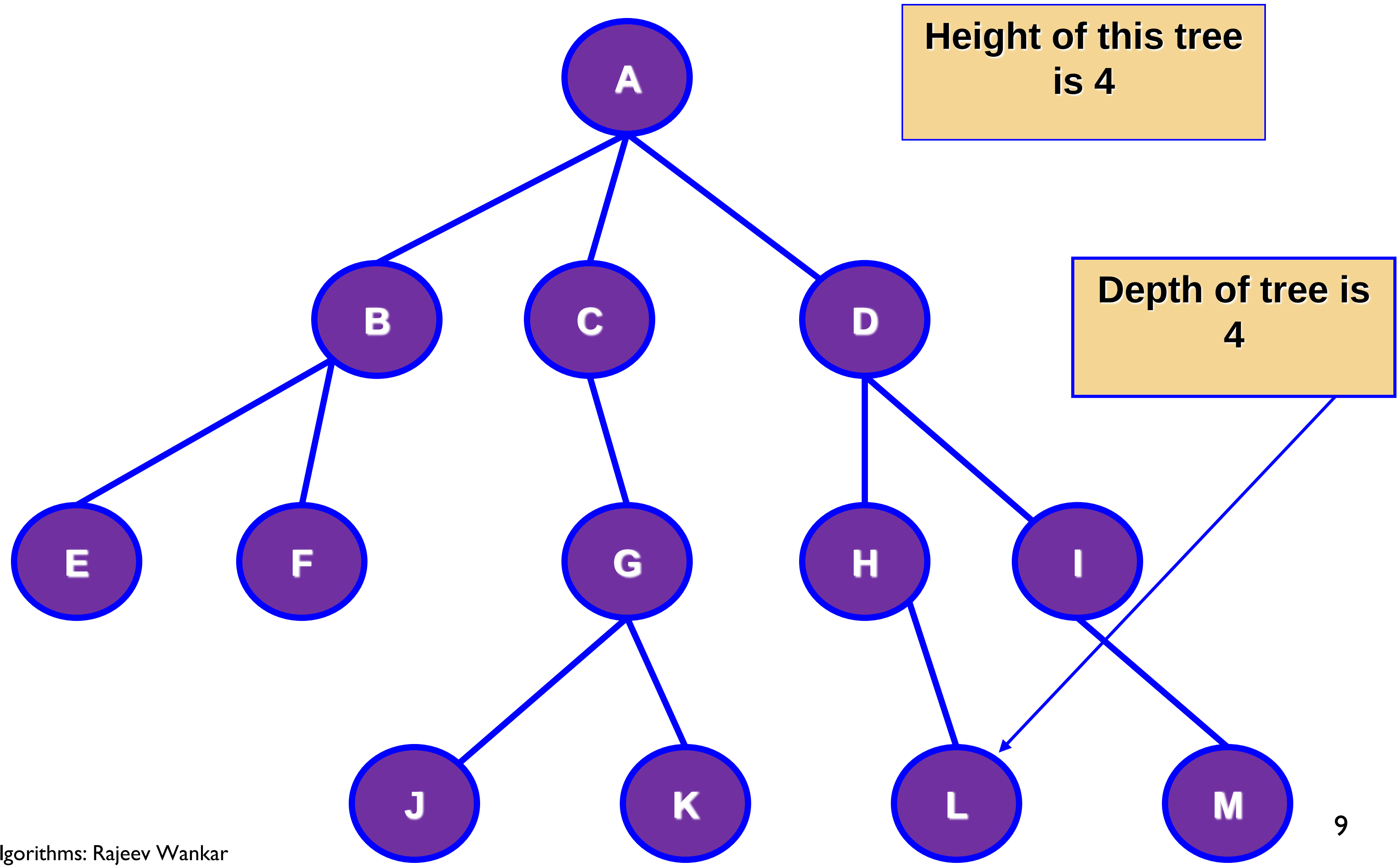

A
B
C
D
E
F
G
H
I
J
K
L
M
H and I are internal nodes
J and k are leafs nodes

Levels

- The ***level*** of a particular node refers to how many generations the node is from the root.
- If the root is assumed to be on level 1, then its children will be on level 2, its grandchildren will be on level 3, and so on.


A
B
C
D
E
F
G
H
I
J
K
L
M
Level 1
Level 2
Level 3
Level 4

Tree – Depth(Max. level of any node) and Height

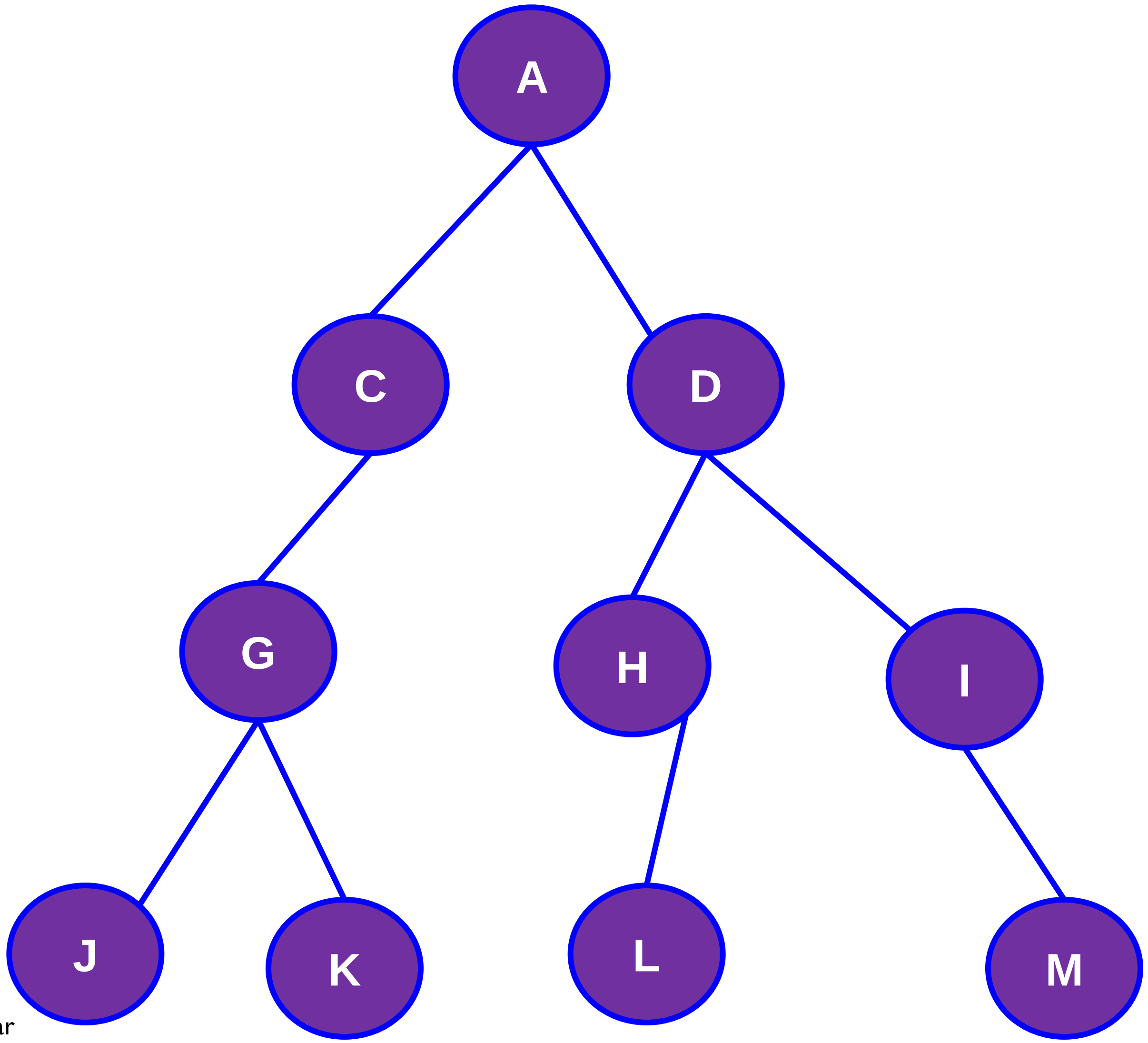


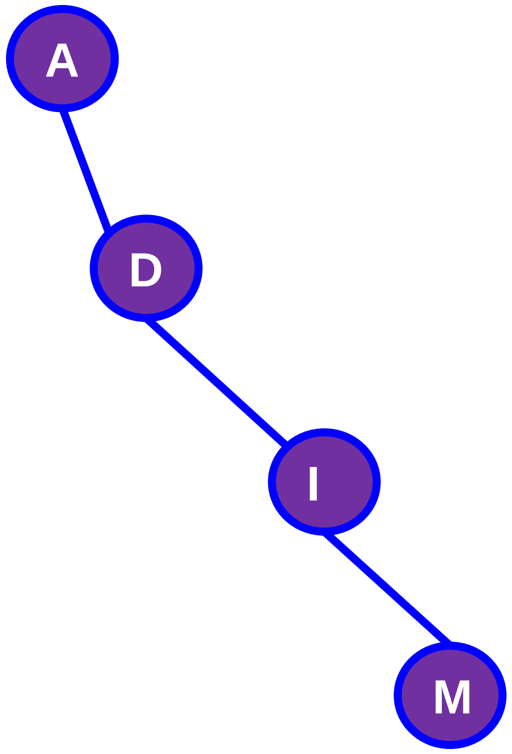
Binary tree: It is a finite set of nodes which is either empty or consist of root and two disjoint binary trees called left and right subtrees.

***Lemma*:** Maximum number of nodes on level i of a binary tree is 2^{i-1}, maximum number of nodes in a binary tree of depth k is 2^k-1, $k > 0$. (levels are numbered from 1 to i)

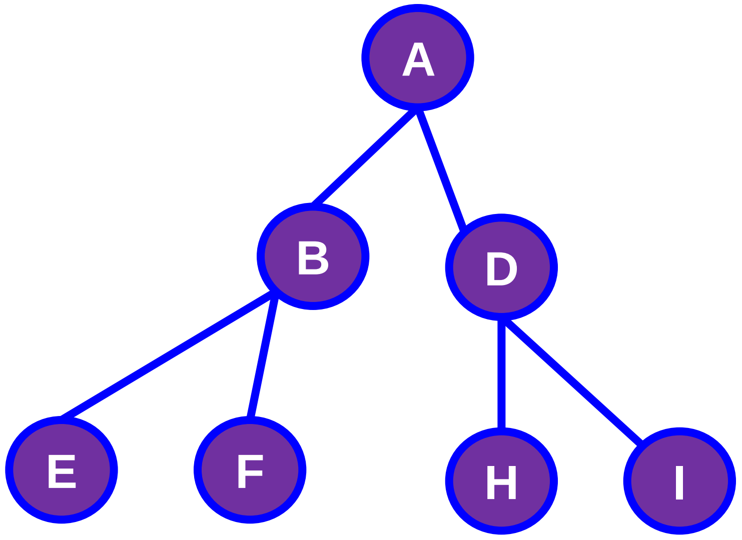
The binary tree of depth k, which has exactly 2^k-1 nodes is called ***full binary tree.*** A node is allowed to have either none or two children **[but not one!]**

Binary tree

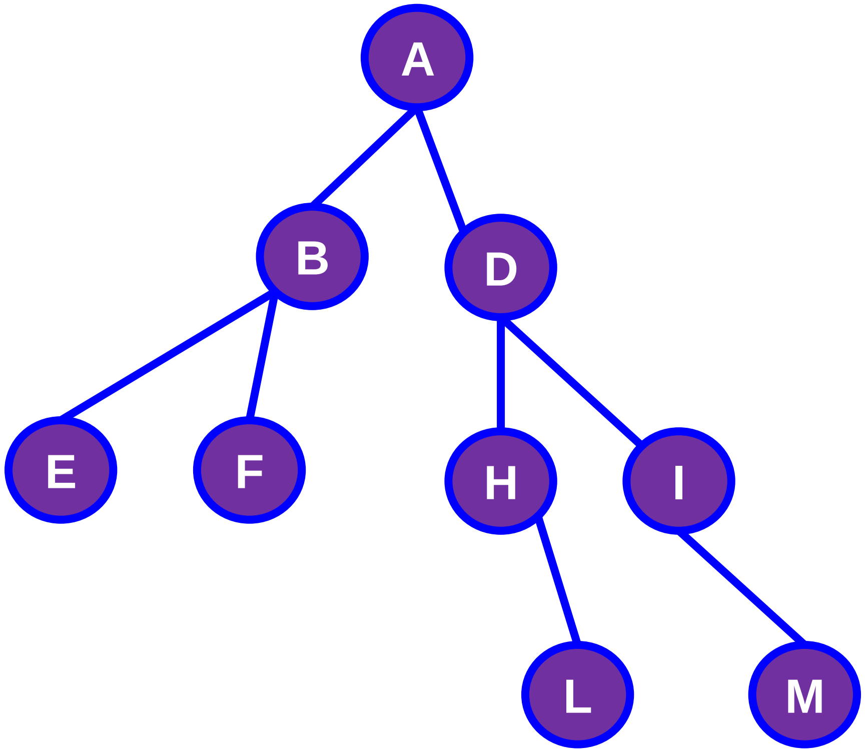




Skewed binary tree



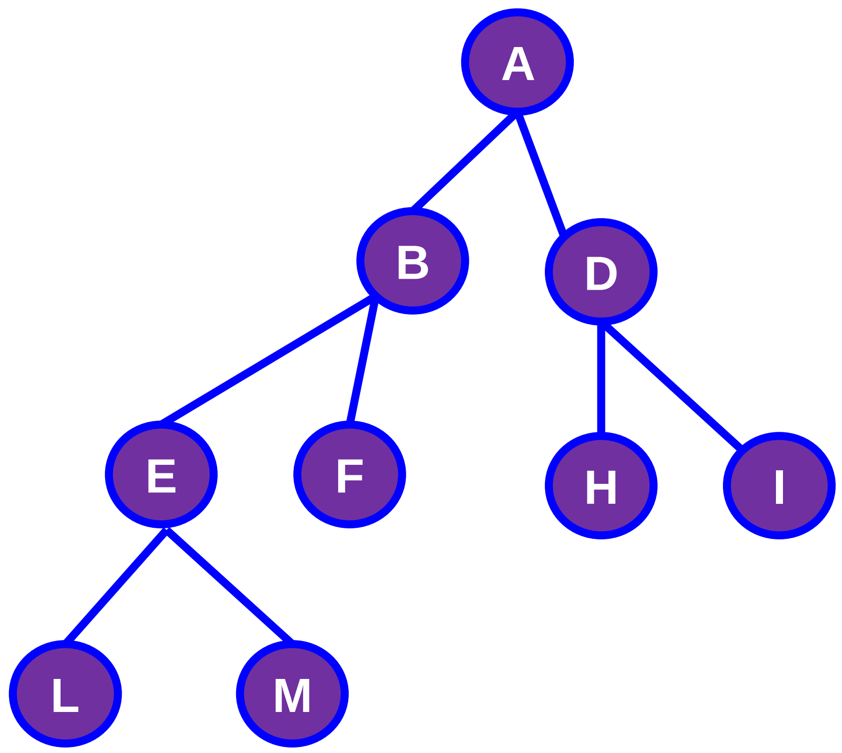
Full binary tree



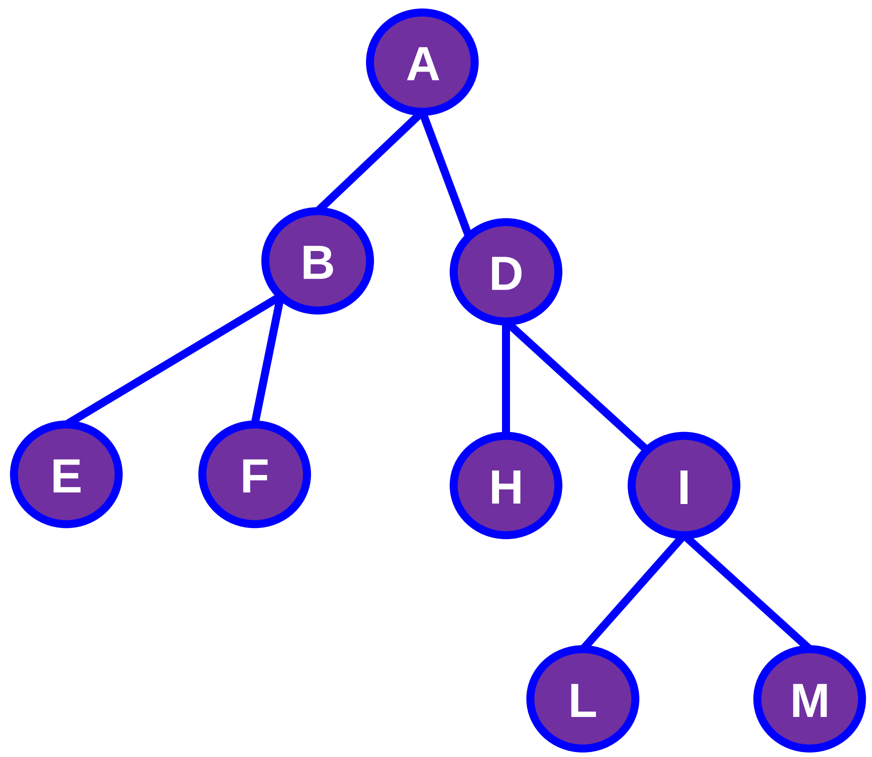
Not a full binary tree

Complete: A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes which are numbered 1 to n in the full binary tree of depth k.

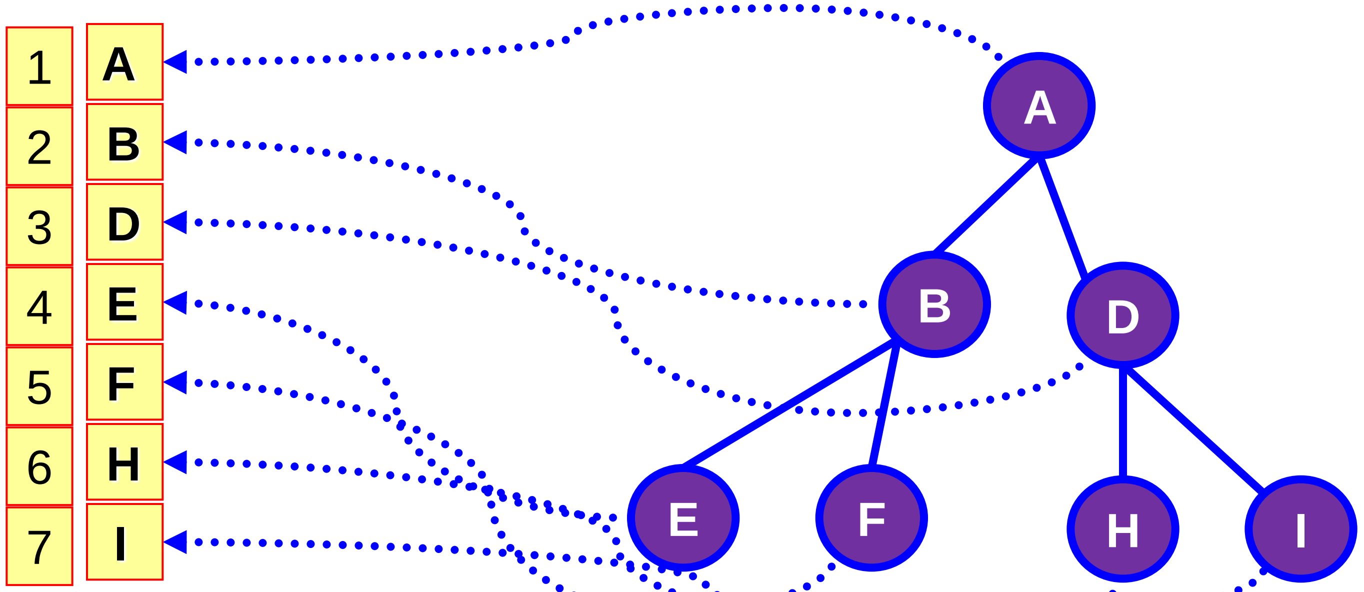
That is the tree has been filled in the level order from left to right.



Complete binary tree



Not a complete binary tree



left = 2*index right = 2*index+1 parent = index/2

Lemma: If a complete binary tree with n nodes is represented sequentially using array then for any node with index *i*, $1 \le i \le n$ we have:

PARENT(*i*) is at $\lfloor i/2 \rfloor$ if $i \neq 1$. When *i* = 1, *i* has no parent.

LCHILD(i) is at 2*i*, if $2i \le n$. If 2*i* > *n* then *i* has no left child.

RCHILD(i) is at 2*i*+1, if $2i+1 \le n$. If 2*i*+1 > *n* then *i* has no right child.

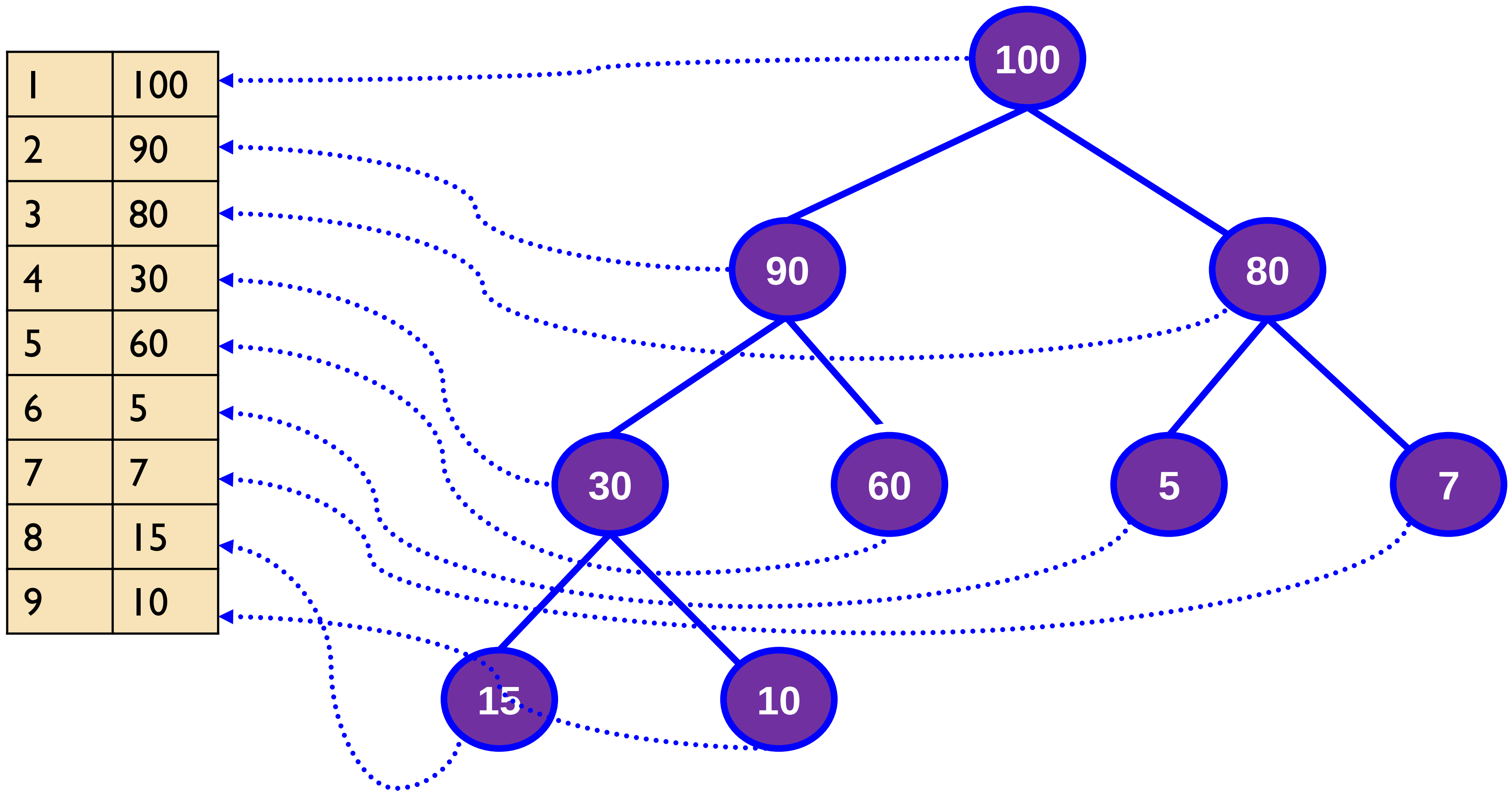
Heap (max): It is a complete binary tree with the property that value at each node is at least as large as the value of its children, if they exist.

Lemma:

If a complete binary tree with n nodes is represented sequentially using a 0-indexed array, then for any node with index iii, where $0 \leq i < n$:

- **PARENT(i)** is at index $\lfloor i-1 \rfloor /2$, if $i \neq 0$. When $i=0$, the node has no parent.
- **LCHILD(i)** is at index $2i+1$, if $2i+1<n$. If $2i+1 \geq n$, the node has no left child.
- **RCHILD(i)** is at index $2i+2$, if $2i+2<n$. If $2i+2 \geq n$, the node has no right child.

Max-heap – Example



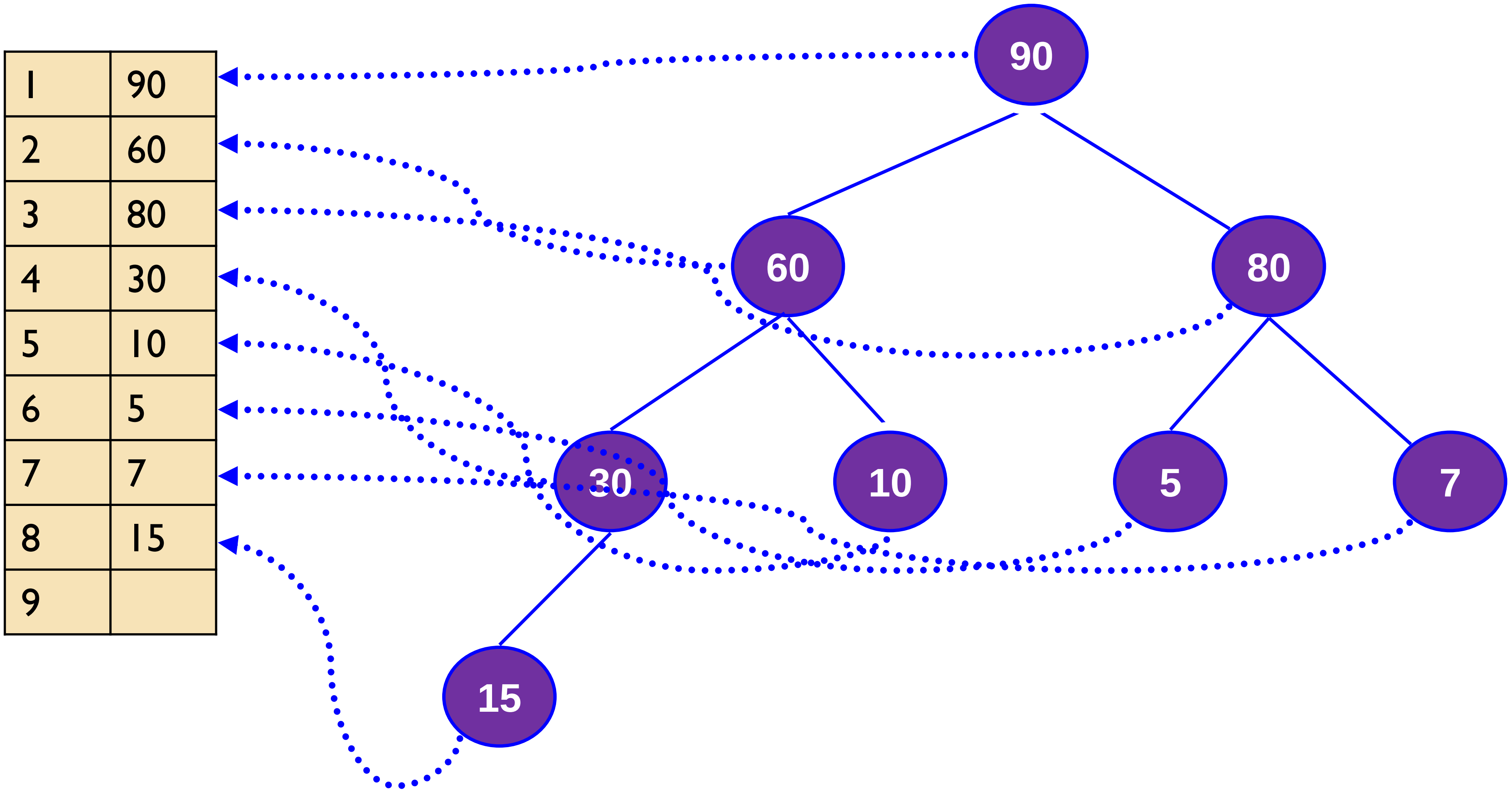
1	100
2	90
3	80
4	30
5	60
6	5
7	7
8	15
9	10

How to form a heap from given n integers stored in A[1..n]

One strategy is to determine how to insert an element at a time in to already existing heap and apply the algorithm **n** times.

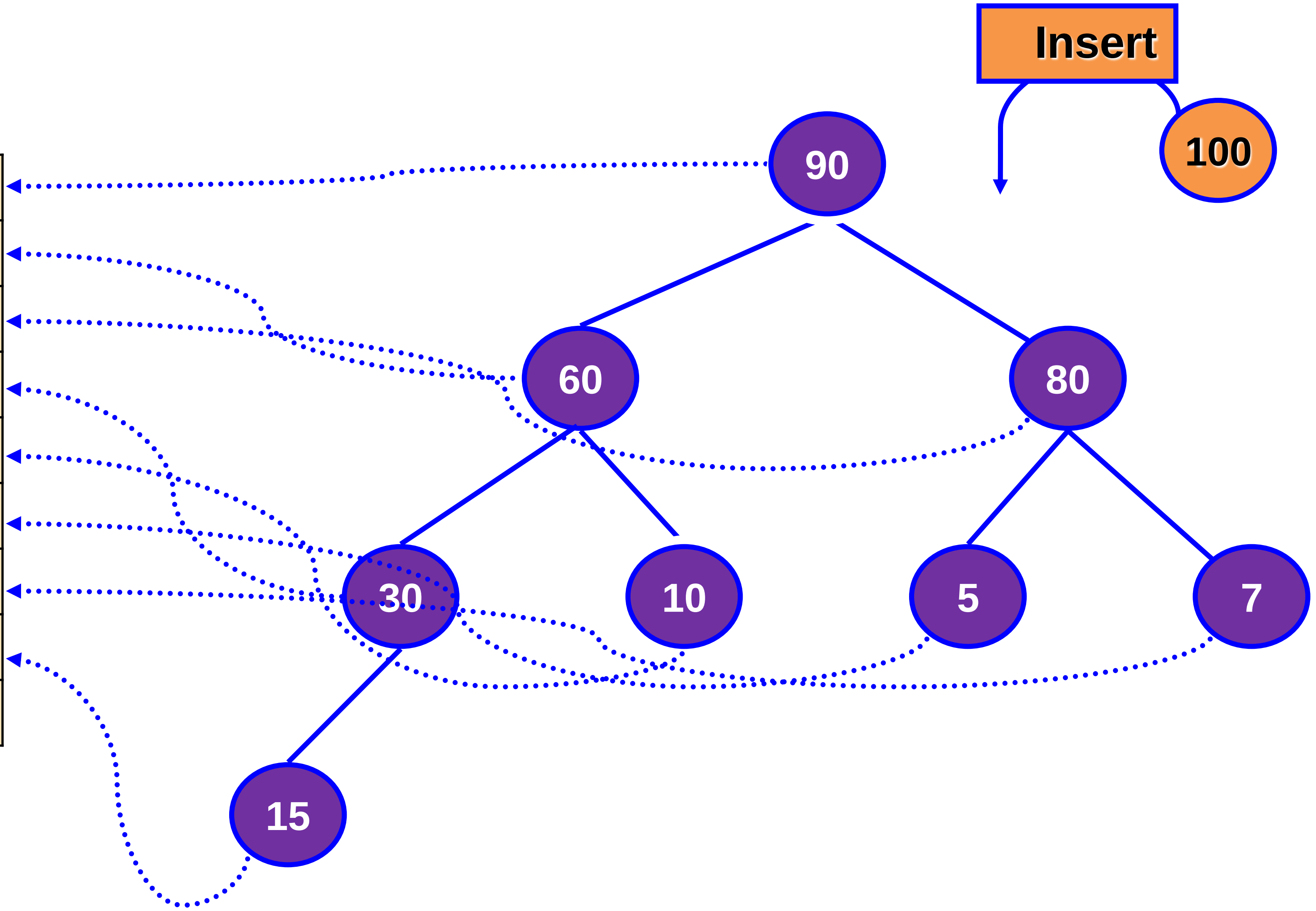
The solution is very simple: add new element "**at the bottom**" of the heap then compare it with its parent, grant parent, great grant parent etc., until it is less than or equal to one of these values.

Max-heap – Insert

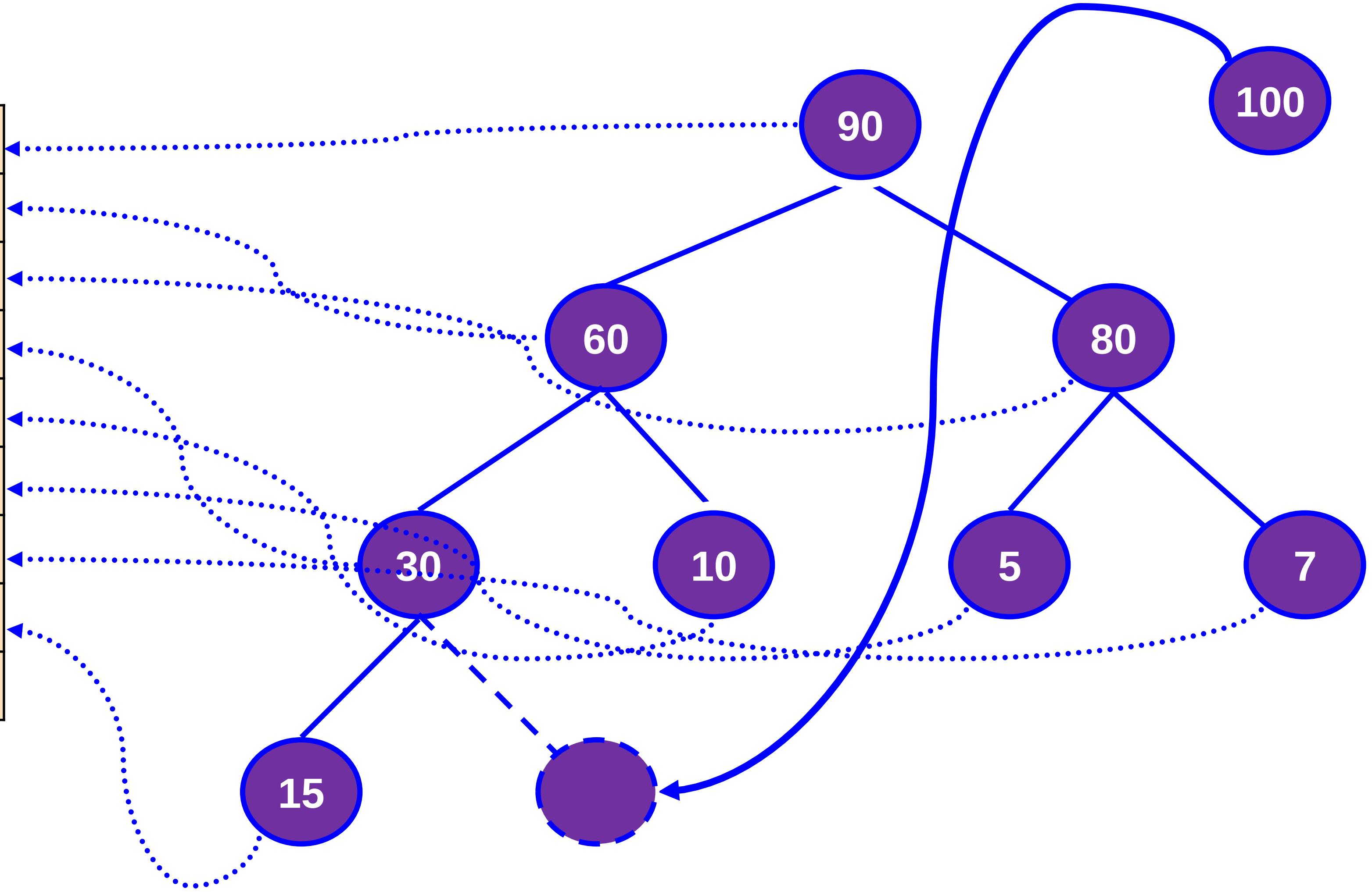


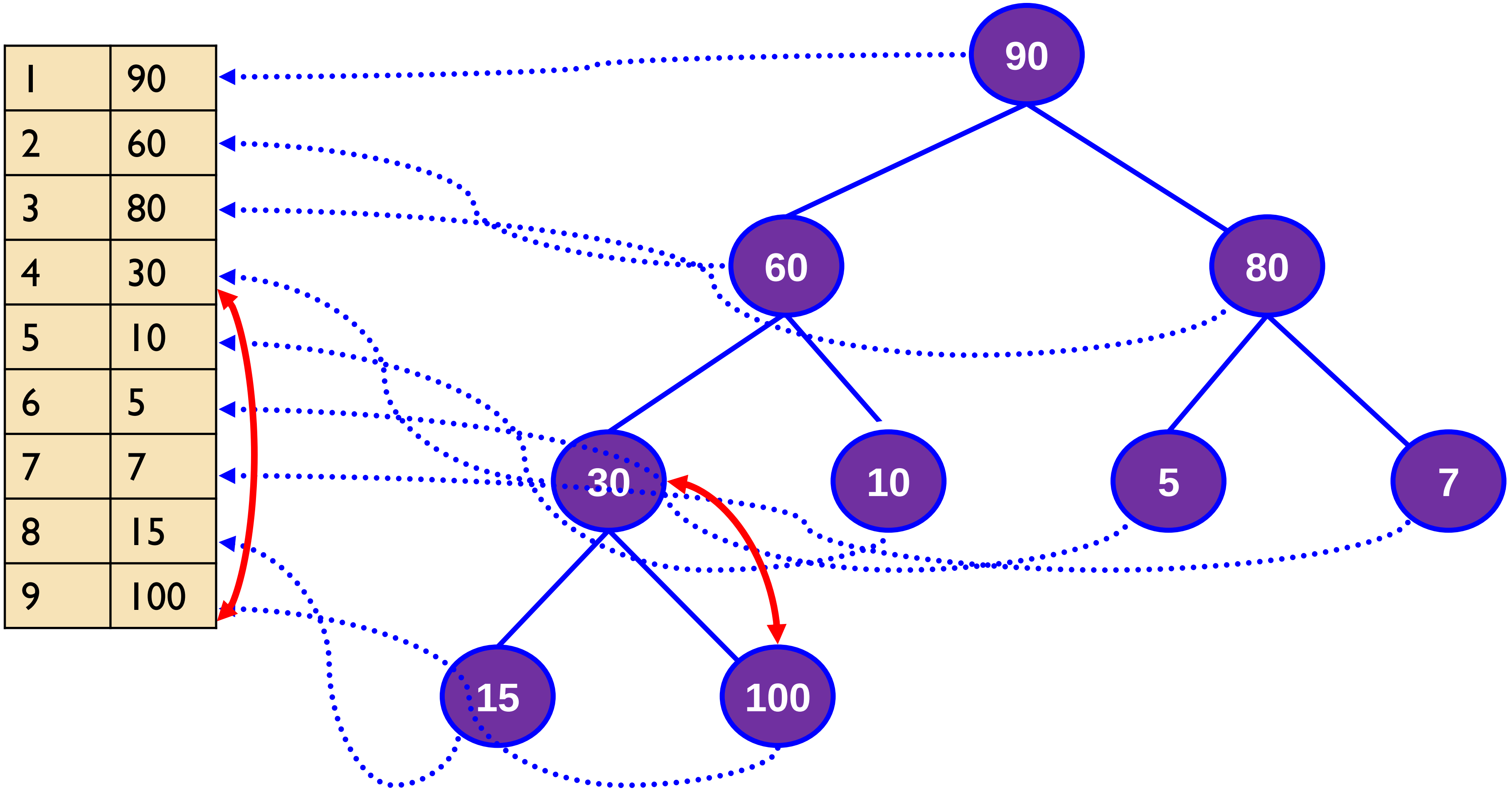
1	90
2	60
3	80
4	30
5	10
6	5
7	7
8	15
9	

1	90
2	60
3	80
4	30
5	10
6	5
7	7
8	15
9	



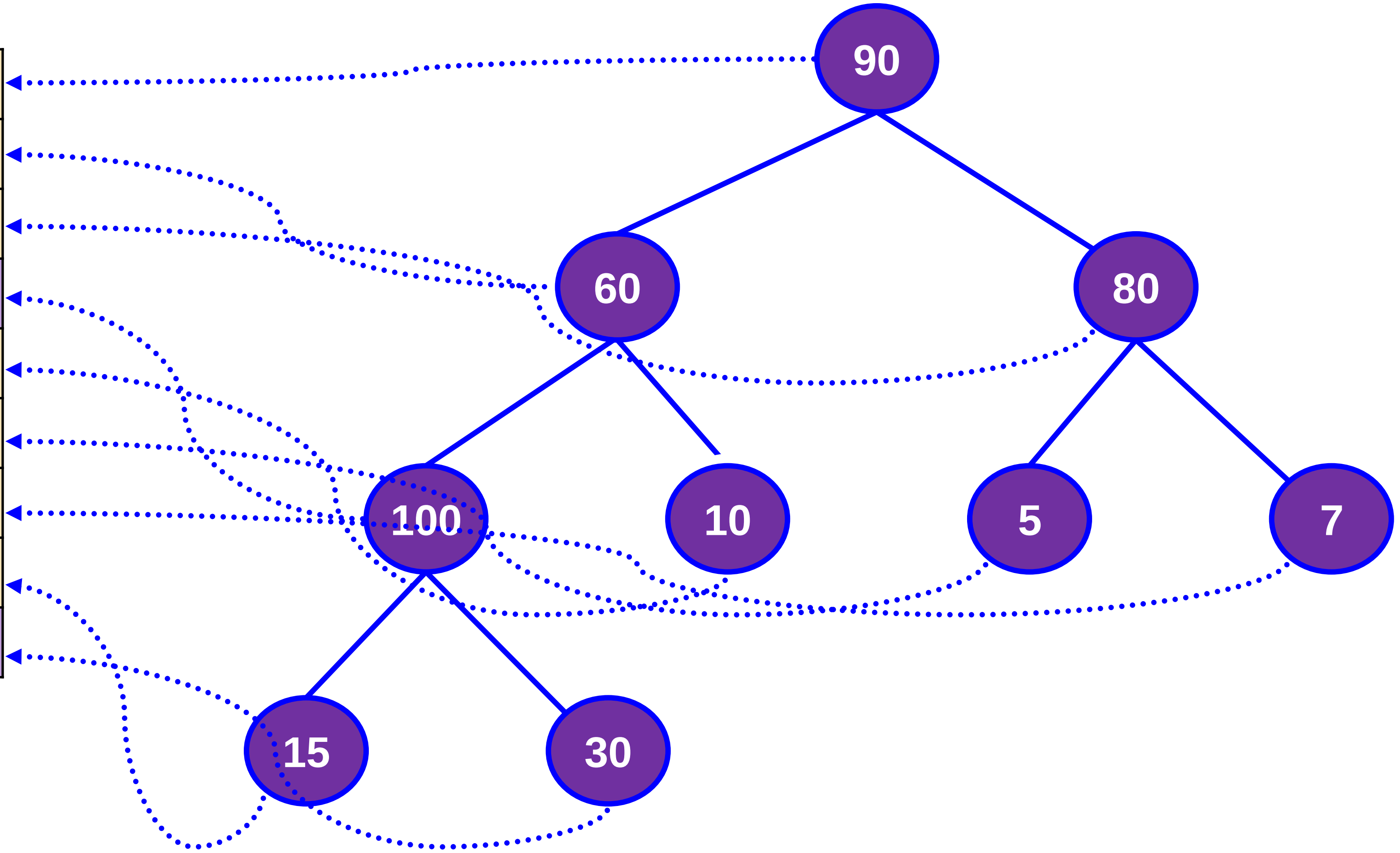
1	90
2	60
3	80
4	30
5	10
6	5
7	7
8	15
9	

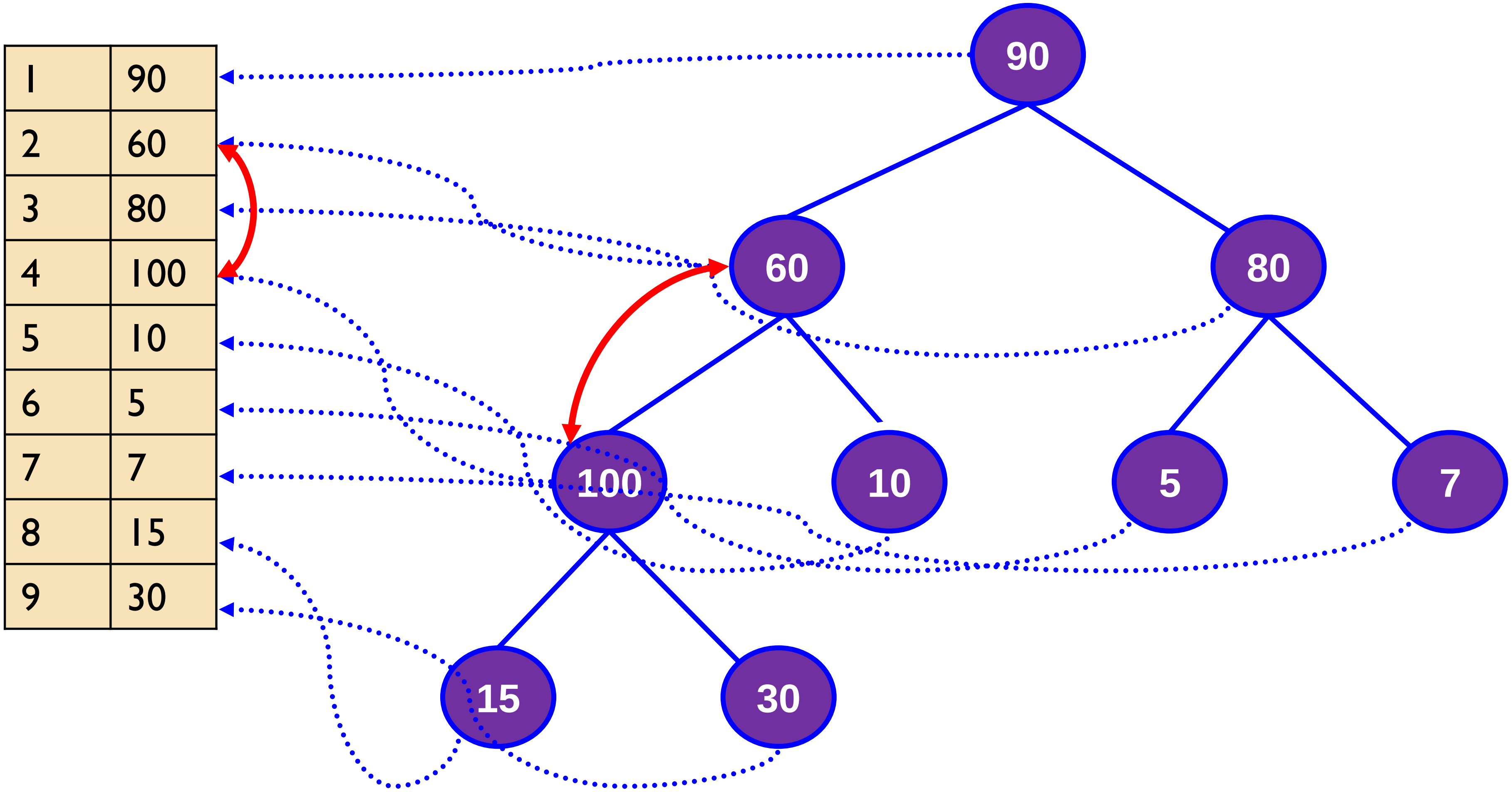




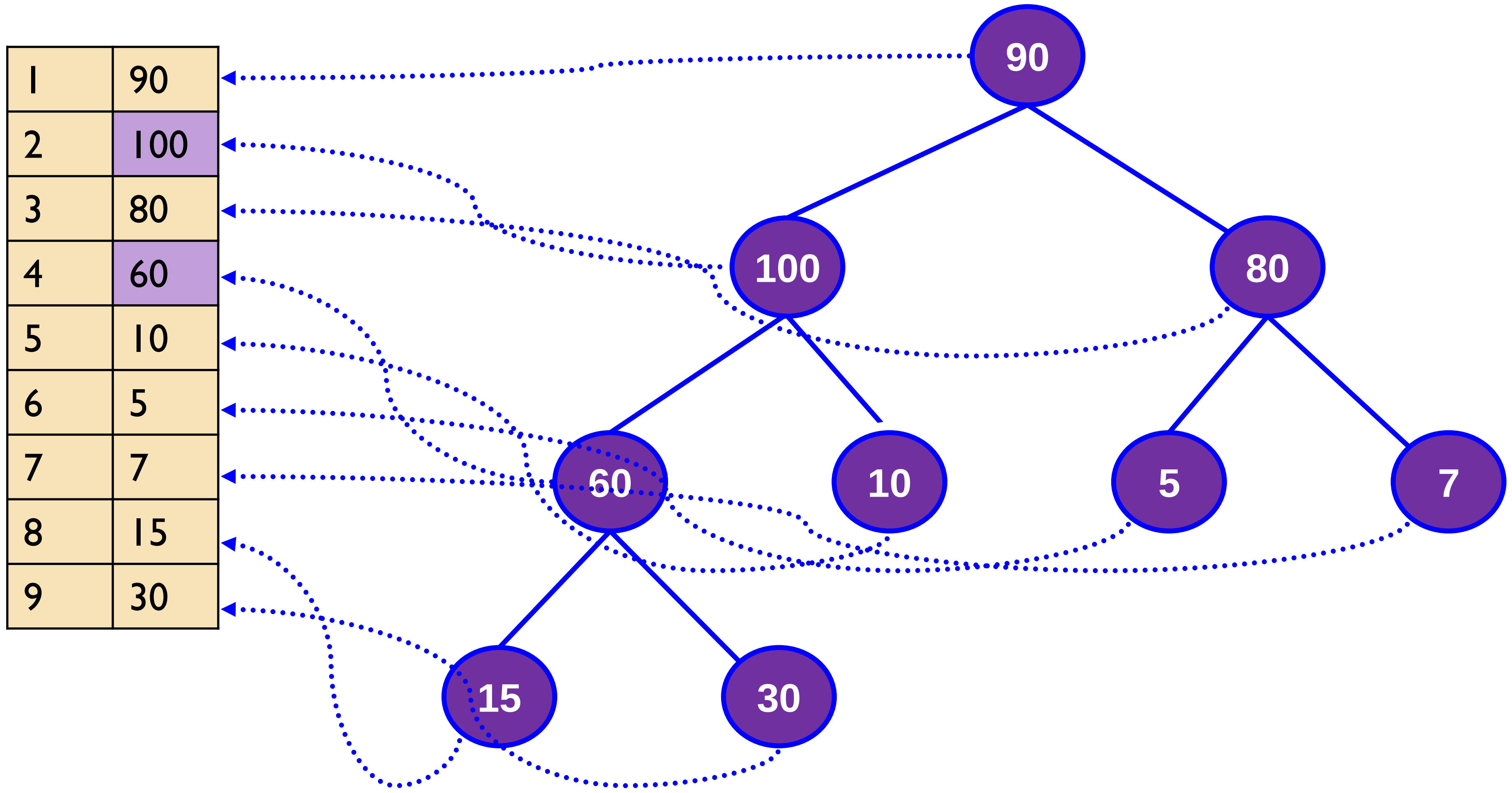
1	90
2	60
3	80
4	30
5	10
6	5
7	7
8	15
9	100

1	90
2	60
3	80
4	100
5	10
6	5
7	7
8	15
9	30



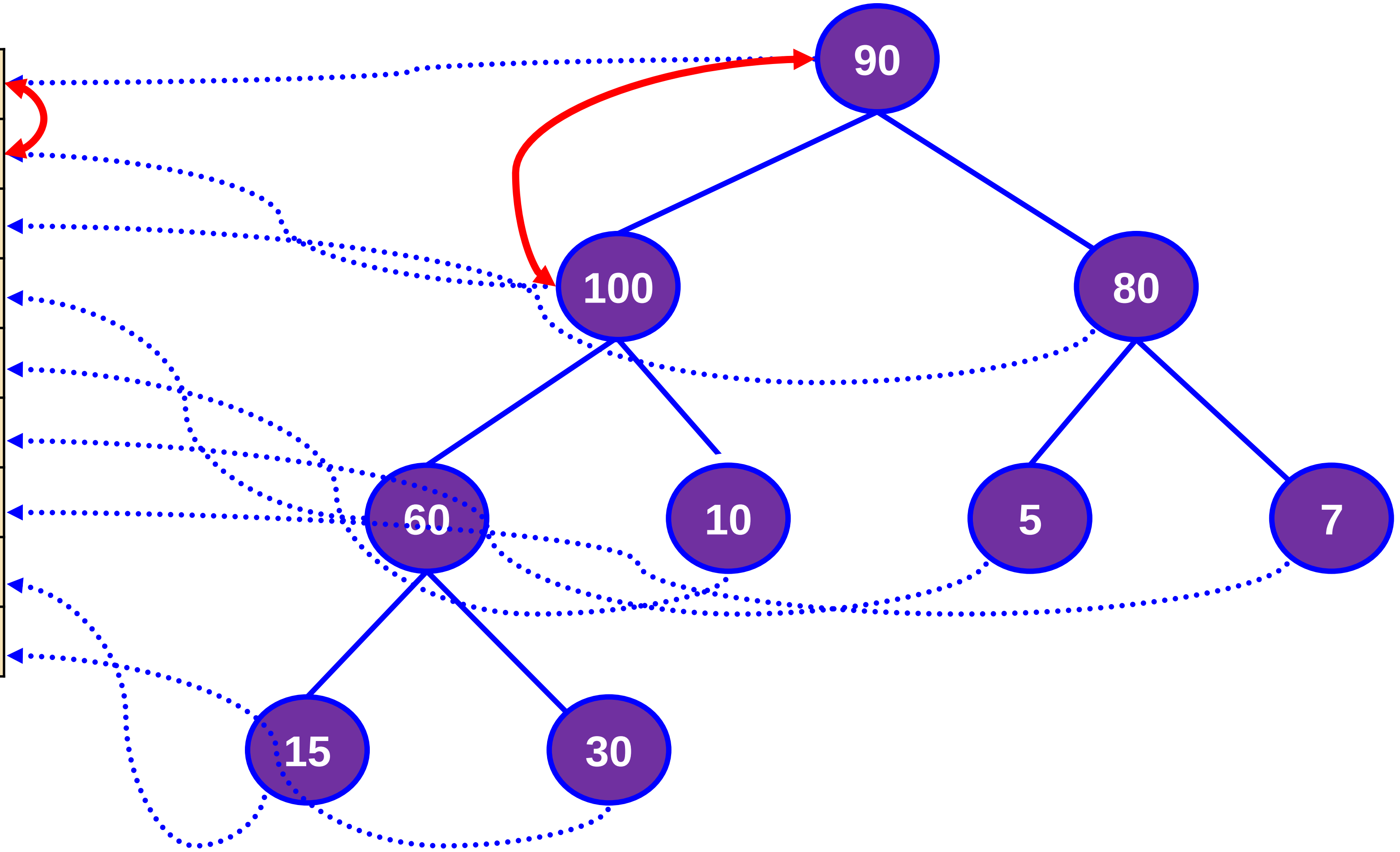


1	90
2	60
3	80
4	100
5	10
6	5
7	7
8	15
9	30

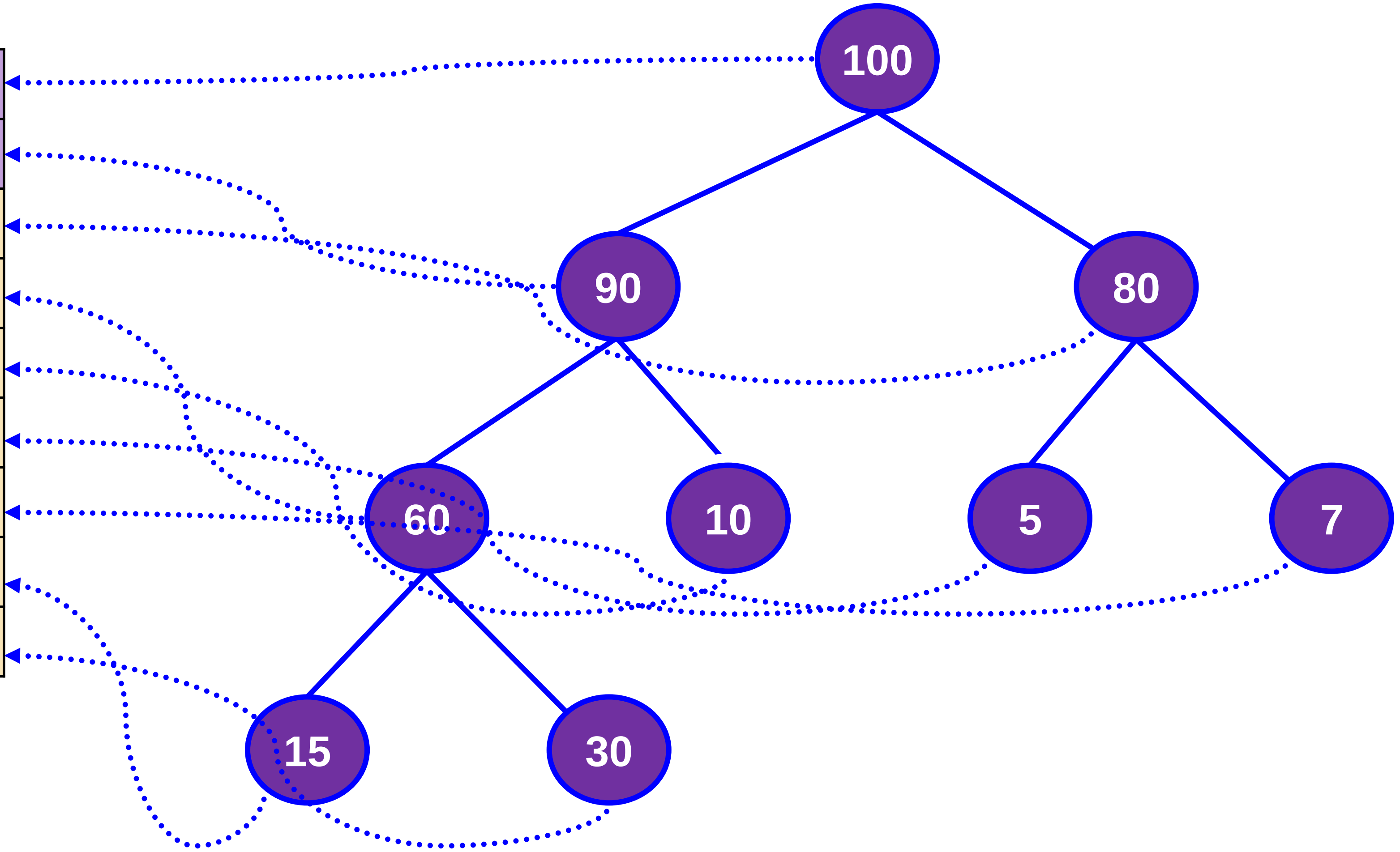


1	90
2	100
3	80
4	60
5	10
6	5
7	7
8	15
9	30

1	90
2	100
3	80
4	60
5	10
6	5
7	7
8	15
9	30



1	100
2	90
3	80
4	60
5	10
6	5
7	7
8	15
9	30



```
procedure INSERT(var A: arraytype; n : integer);
//Insert the value in A[n] into the heap which is stored at A[1]
to A[n-1]//
var i,j: integer; item: itemtype;
begin {INSERT}
   j ← n; i ← $\lfloor n/2 \rfloor$ ; item ← A[n];
        while ((i > 0) and (A[i] < item)) do
         begin
             A[j] ← A[i]                  // move the parent down//
              j ← i; i ← $\lfloor i/2 \rfloor$             //parent of A[i] is at A[ $\lfloor i/2 \rfloor$ ]//
         end;
  A[j] ← item;
end {INSERT}.
```

Worse case analysis of INSERT

The data set causes the heap creation using INSERT procedure to behave in the worse way when the elements are inserted in the ascending order. Every new value is rise to become a new root.

There are at most 2^{i-1} nodes on level i of a complete binary tree,

$1 \leq i \leq \lceil \log(n+1) \rceil$. For a node at level i, the **distance** to the root is (*i*-1). Thus the time required for the heap creation using INSERT procedure is:

$$\sum_{i=1}^{\lceil \log(n+1) \rceil} 2^{i-1}(i-1) \qquad \text{.........(A)}$$

In general we know that $\sum_{i=1}^{m} x^{i-1}.(i-1) = \frac{mx^m+1}{x-1} - \frac{x^{m+1}-1}{(x-1)^2}$

for $x > 1$ ....(B)

Choosing $m = \lceil \log(n+1) \rceil$ and x = 2, we get,

$$\sum_{i=1}^{\lceil \log(n+1) \rceil} 2^{i-1}.(i-1) = \lceil \log(n+1) \rceil . 2^{\lceil \log(n+1) \rceil} - 2^{\lceil \log(n+1) \rceil + 1} + 2$$

$< \lceil \log(n+1) \rceil . 2^{\lceil \log(n+1) \rceil}$ for all $n \geq 1$

<((1+ log n).2n)

= O(n log n)

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}, \quad \forall n > 1$$ ……………………(A)

$$\frac{d}{dx}\left(\sum_{i=0}^{n-1} x^i\right) = \frac{d}{dx}\left(\frac{x^n - 1}{x - 1}\right)$$ ……………………(B)

$$\sum_{i=1}^{n-1} i.x^{i-1} = \frac{nx^{n-1}}{x-1} - \frac{x^n - 1}{(x-1)^2}$$ ……………………(C)

we have from (A) $\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$ Put j=i+1 or i=j-1

$$\sum_{j=1}^{n} x^{j-1} = \frac{x^n - 1}{x - 1}$$ ……………………(D)

therefore $\sum_{i=1}^{n-1} x^{i-1} = \frac{x^{n-1}-1}{x-1}$ ……………………..(E)

Subtracting (E) from (C) we get

$$\sum_{i=1}^{n-1} i.x^{i-1} - \sum_{i=1}^{n-1} x^{i-1} = \frac{nx^{n-1}}{x-1} - \frac{x^n-1}{(x-1)^2} - \frac{x^{n-1}-1}{x-1}$$

$$\sum_{i=1}^{n-1} (i-1).x^{i-1} = \frac{(n-1)x^{n-1}+1}{x-1} - \frac{x^n-1}{(x-1)^2}$$

$$\sum_{i=1}^{n} (i-1).x^{i-1} = \frac{nx^n+1}{x-1} - \frac{x^{n+1}-1}{(x-1)^2}$$

Average case analysis of INSERT: (Istvan and Porter's analysis)

"Surprisingly it is faster than the worse case, O(n) rather than O(n log n)".

On an average each new value rise to a constant number of levels in the tree.

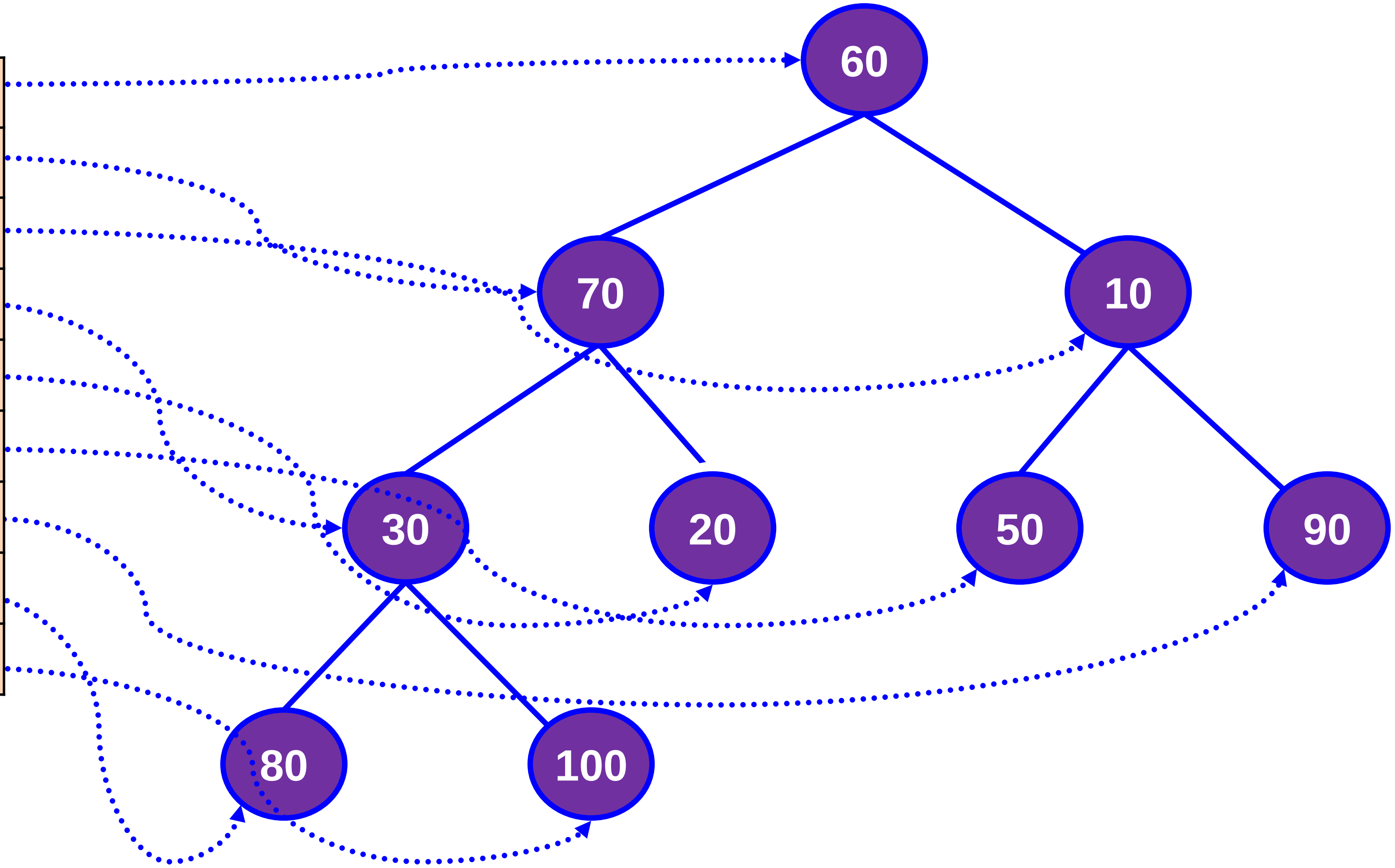
"There is another algorithm for creation of the heap which has the nice property that its worse case time is an order of magnitude faster than n-1 calls."

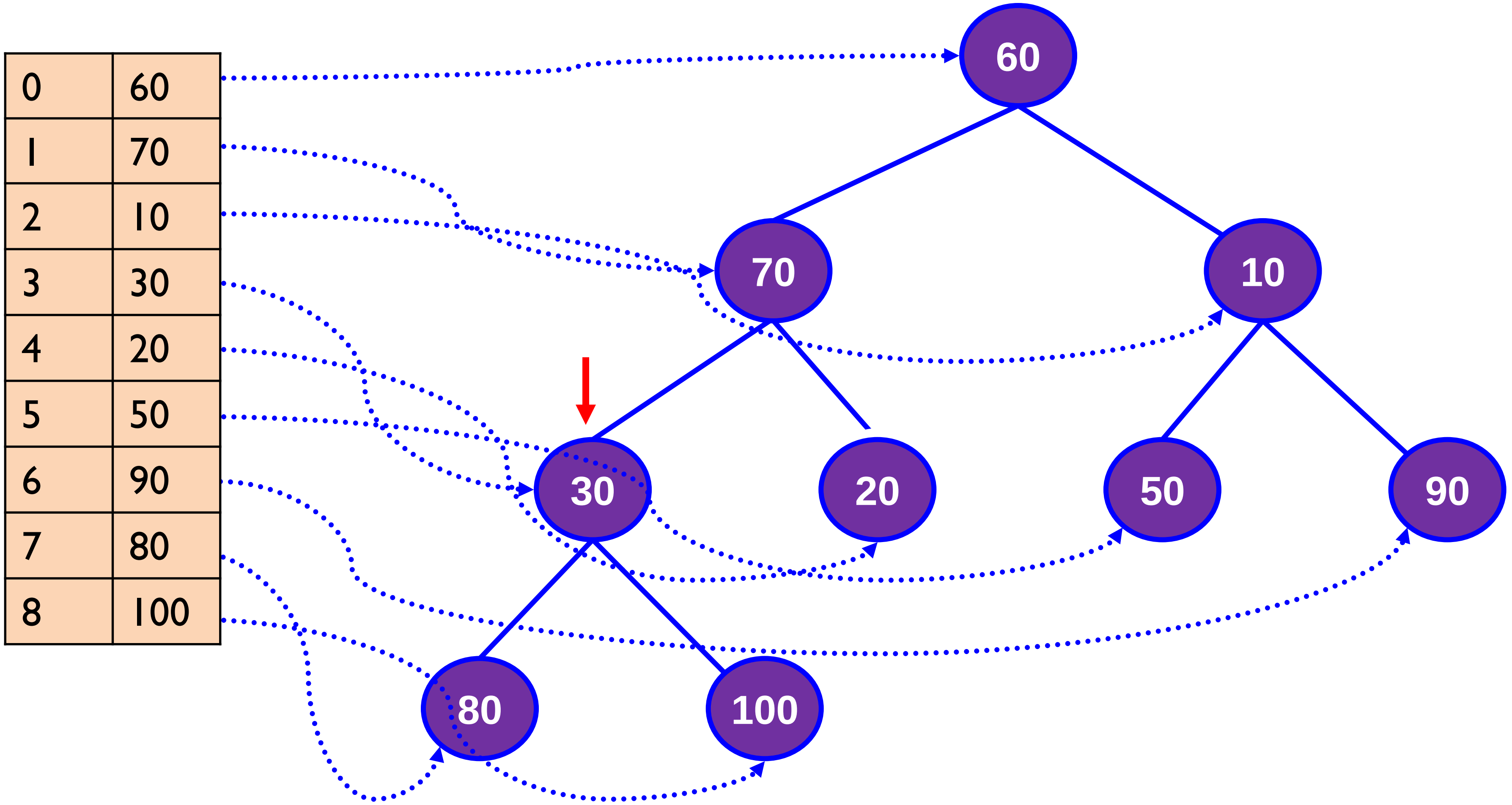
- **It works level by level**. At each level it will be the case that the left and right subtrees of any node are heaps, only value at the root is violating the heap property.

- So it will be sufficient to device a method which converts a binary tree in to heap, in which only root is violating the heap property.

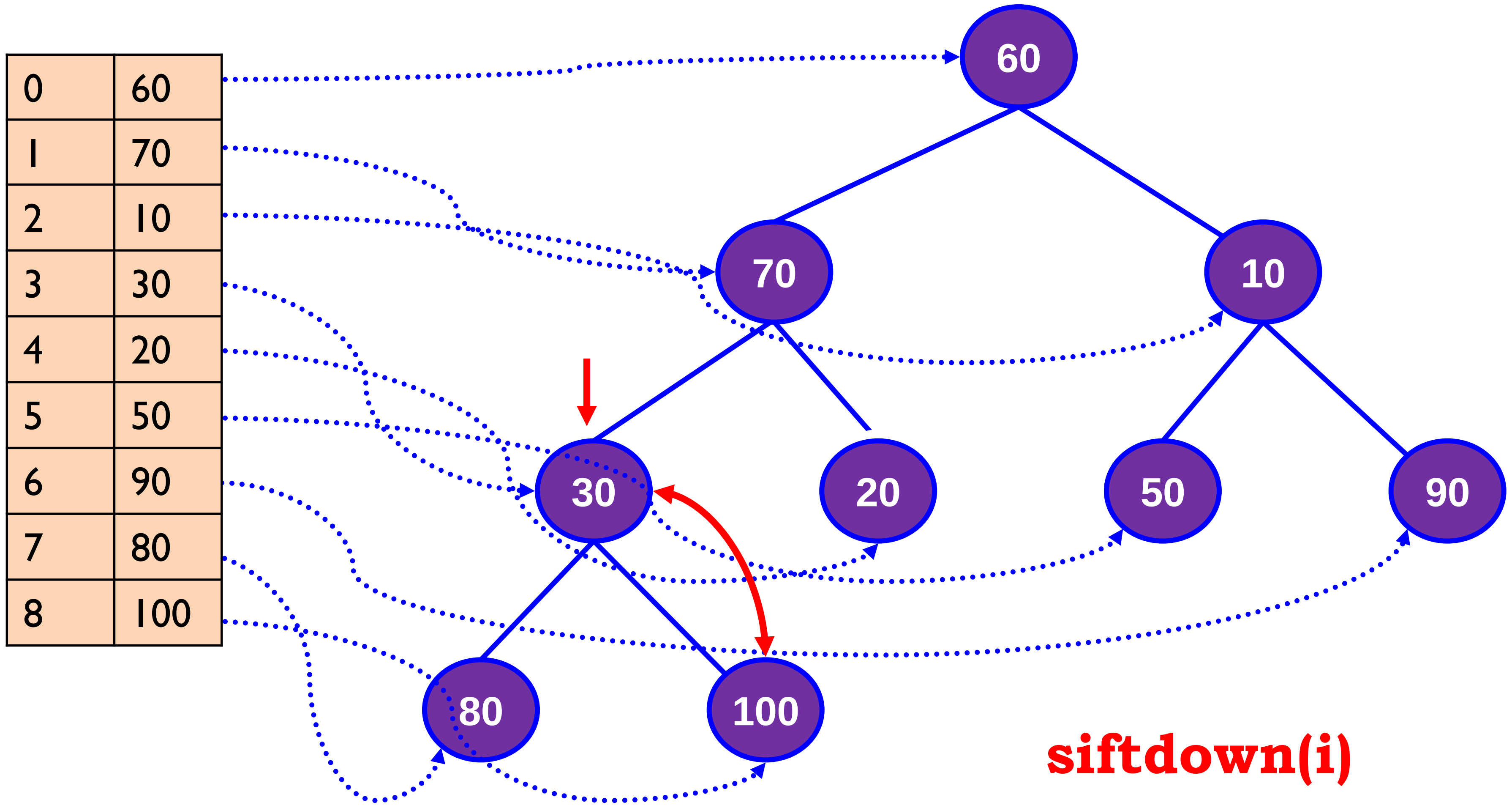
Building a Max-heap

0	60
1	70
2	10
3	30
4	20
5	50
6	90
7	80
8	100



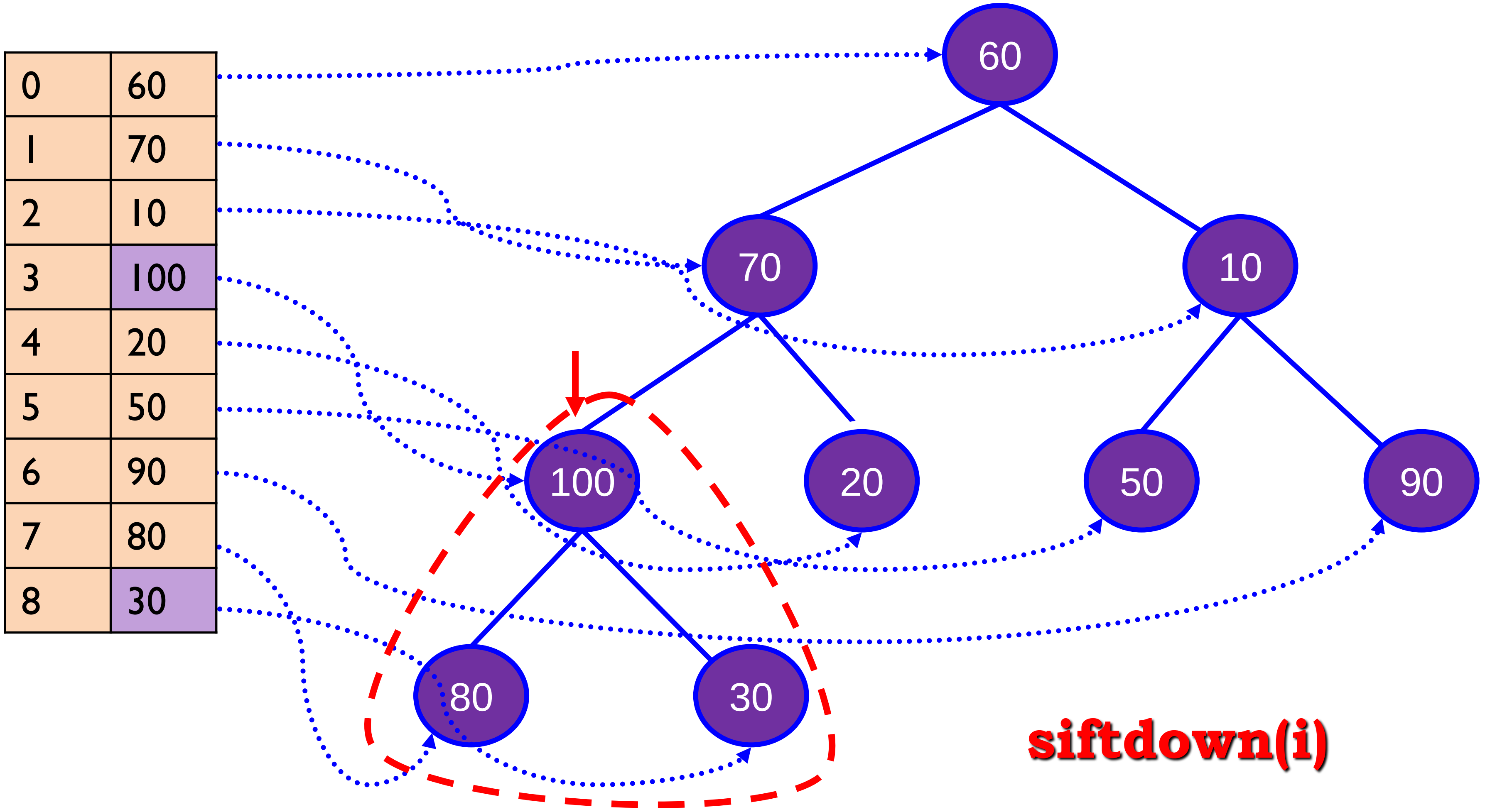


0	60
1	70
2	10
3	30
4	20
5	50
6	90
7	80
8	100



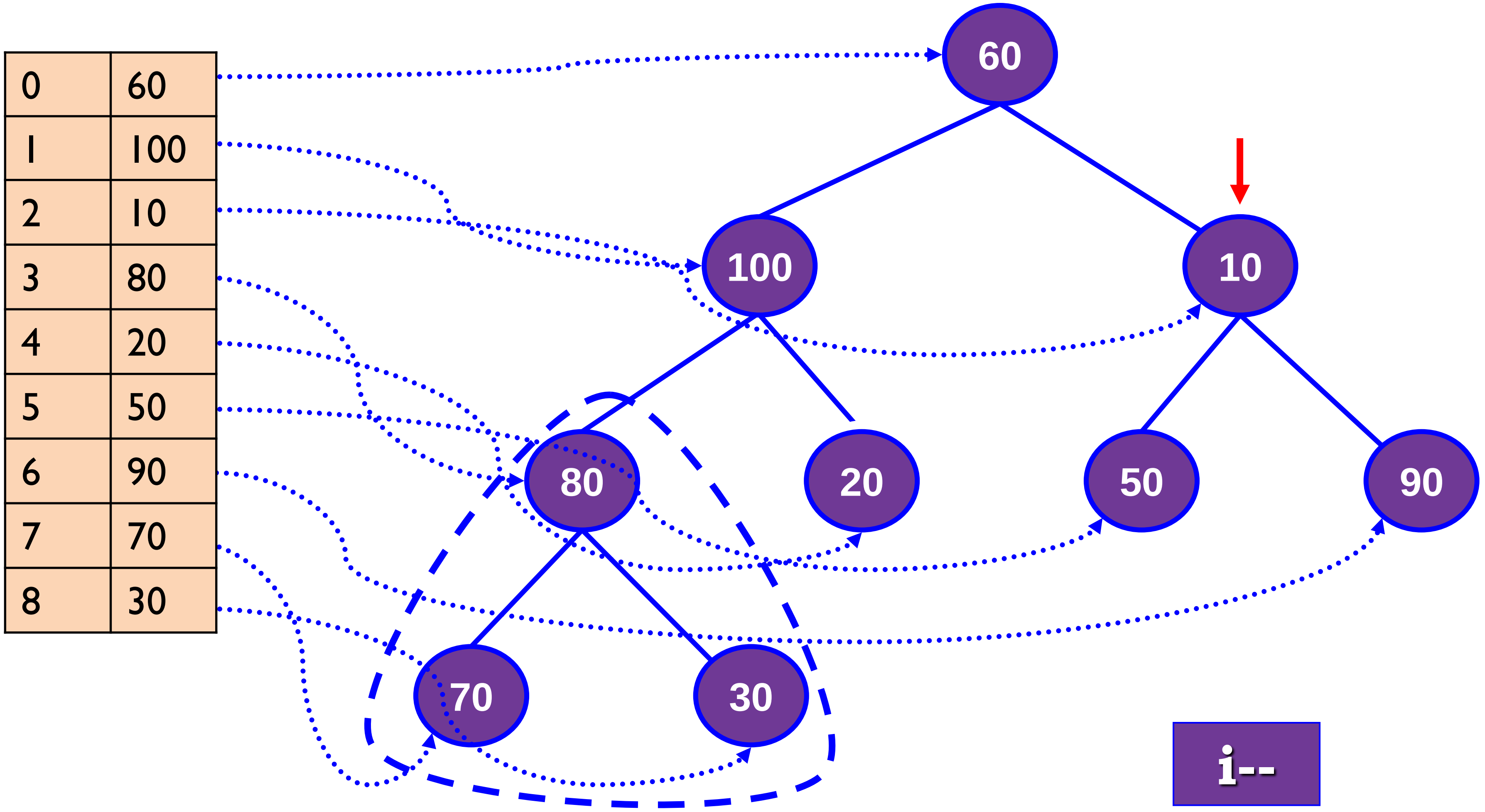
0	60
1	70
2	10
3	30
4	20
5	50
6	90
7	80
8	100

siftdown(i)

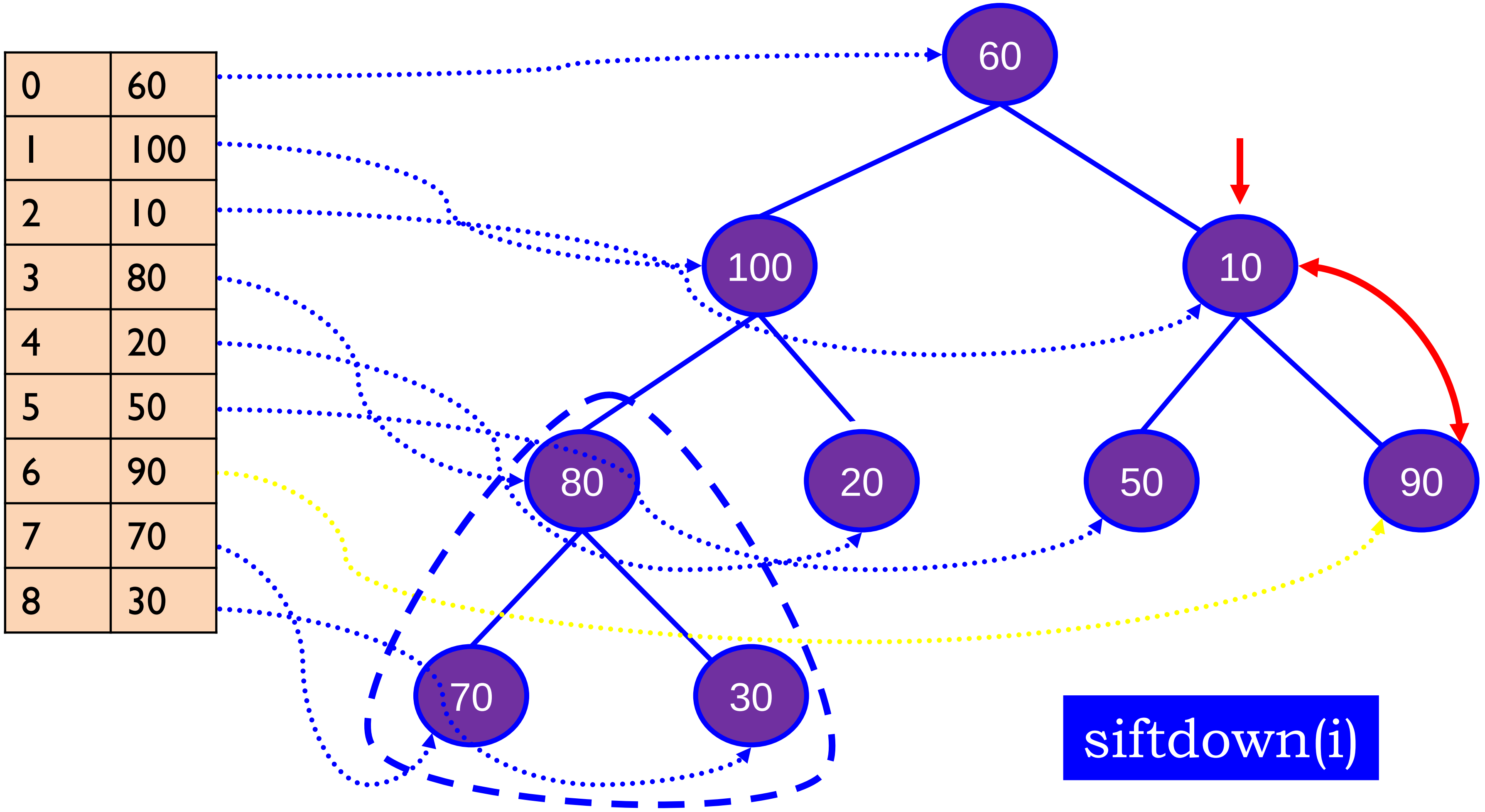


0	60
1	70
2	10
3	100
4	20
5	50
6	90
7	80
8	30

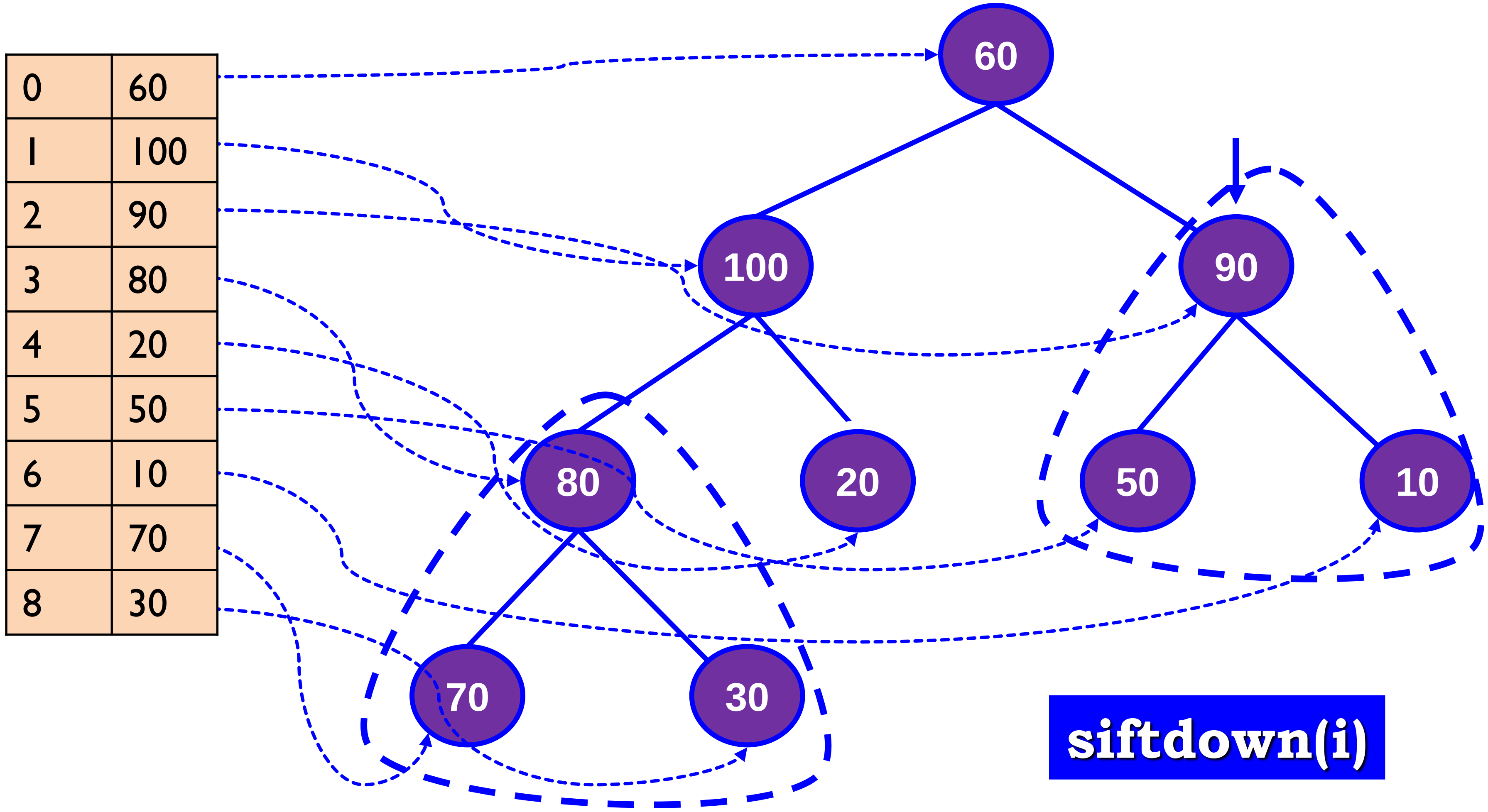
siftdown(i)



0	60
1	100
2	10
3	80
4	20
5	50
6	90
7	70
8	30



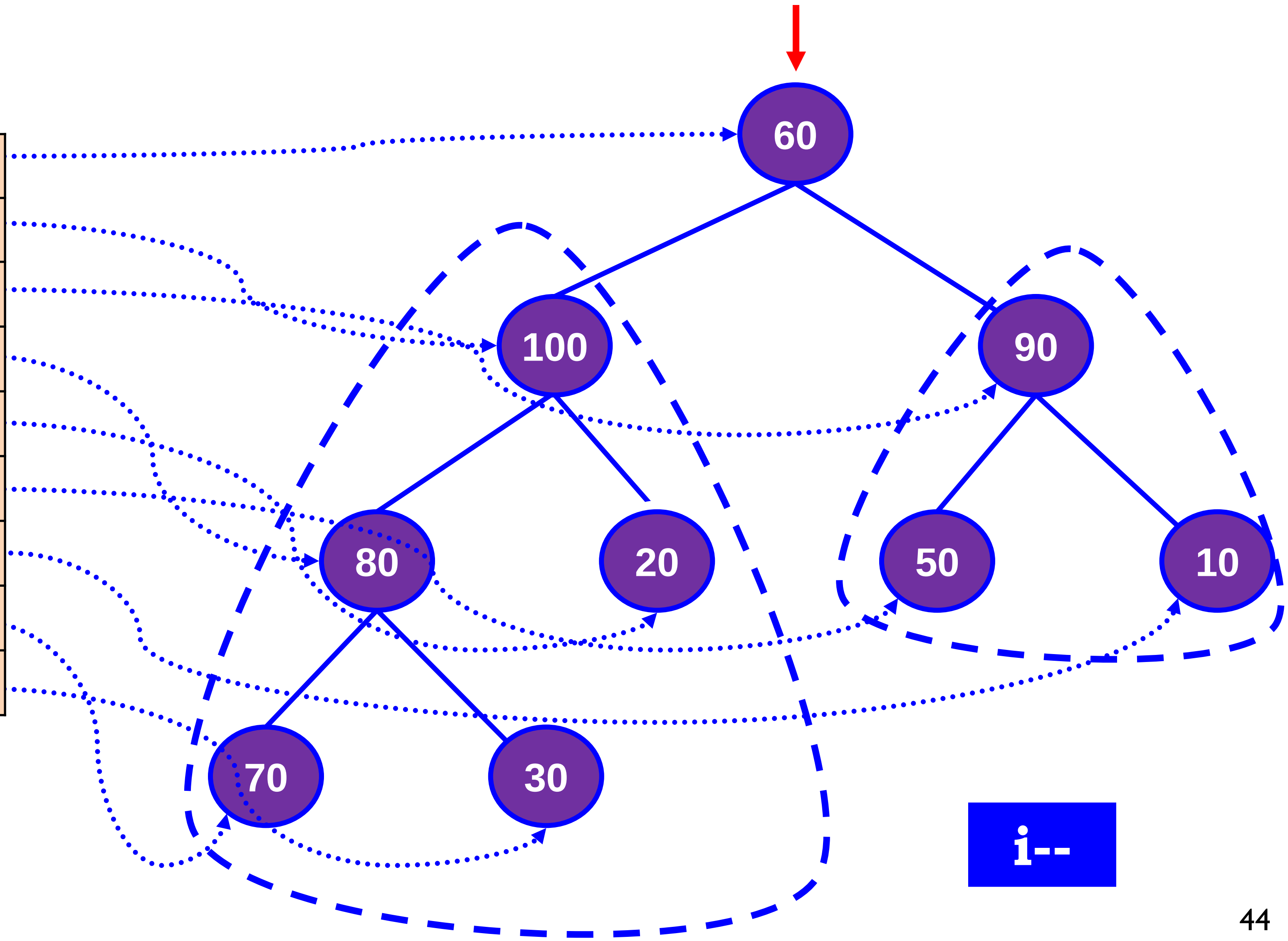
0	60
1	100
2	10
3	80
4	20
5	50
6	90
7	70
8	30



0	60
1	100
2	90
3	80
4	20
5	50
6	10
7	70
8	30

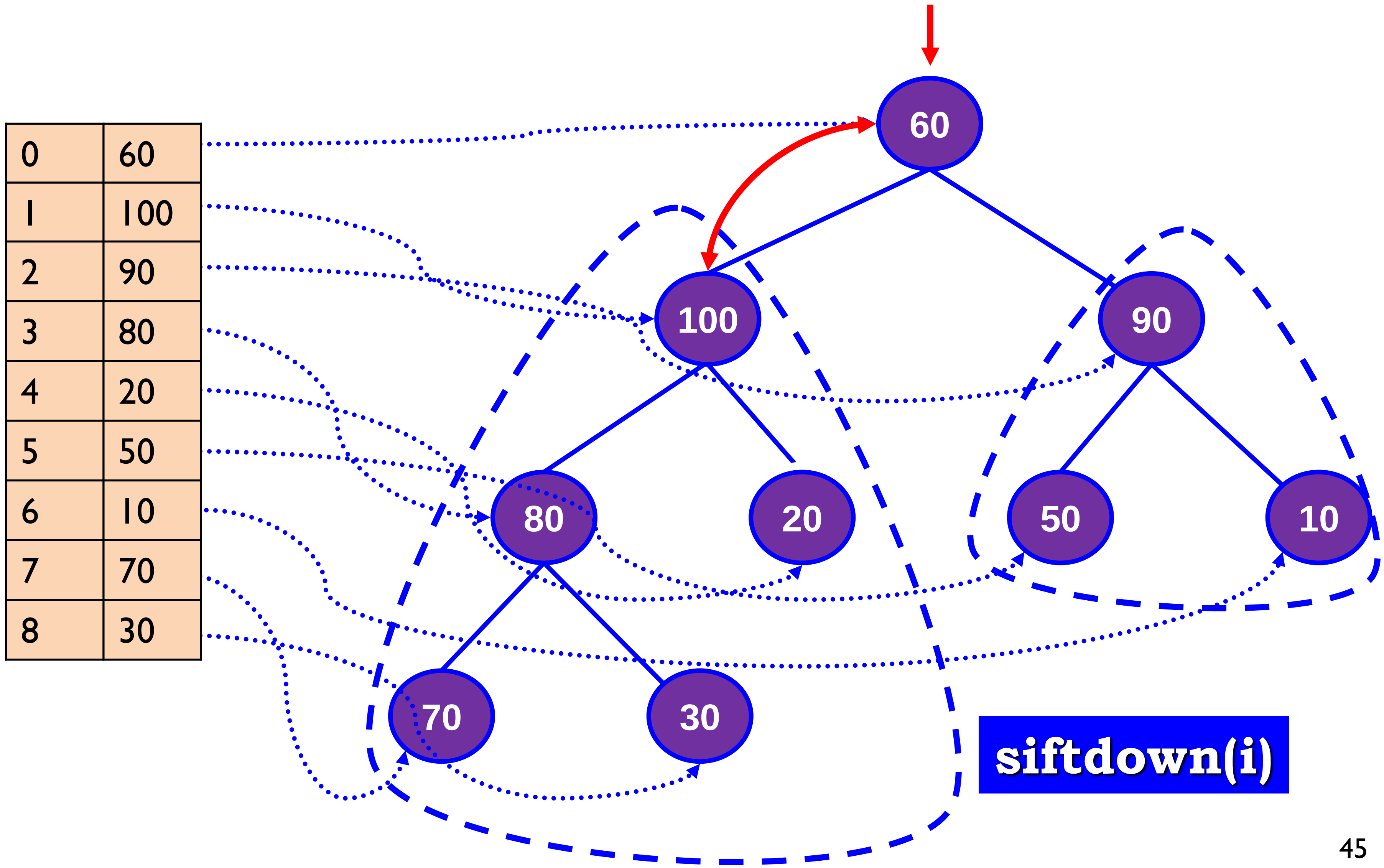
siftdown(i)

0	60
1	100
2	90
3	80
4	20
5	50
6	10
7	70
8	30

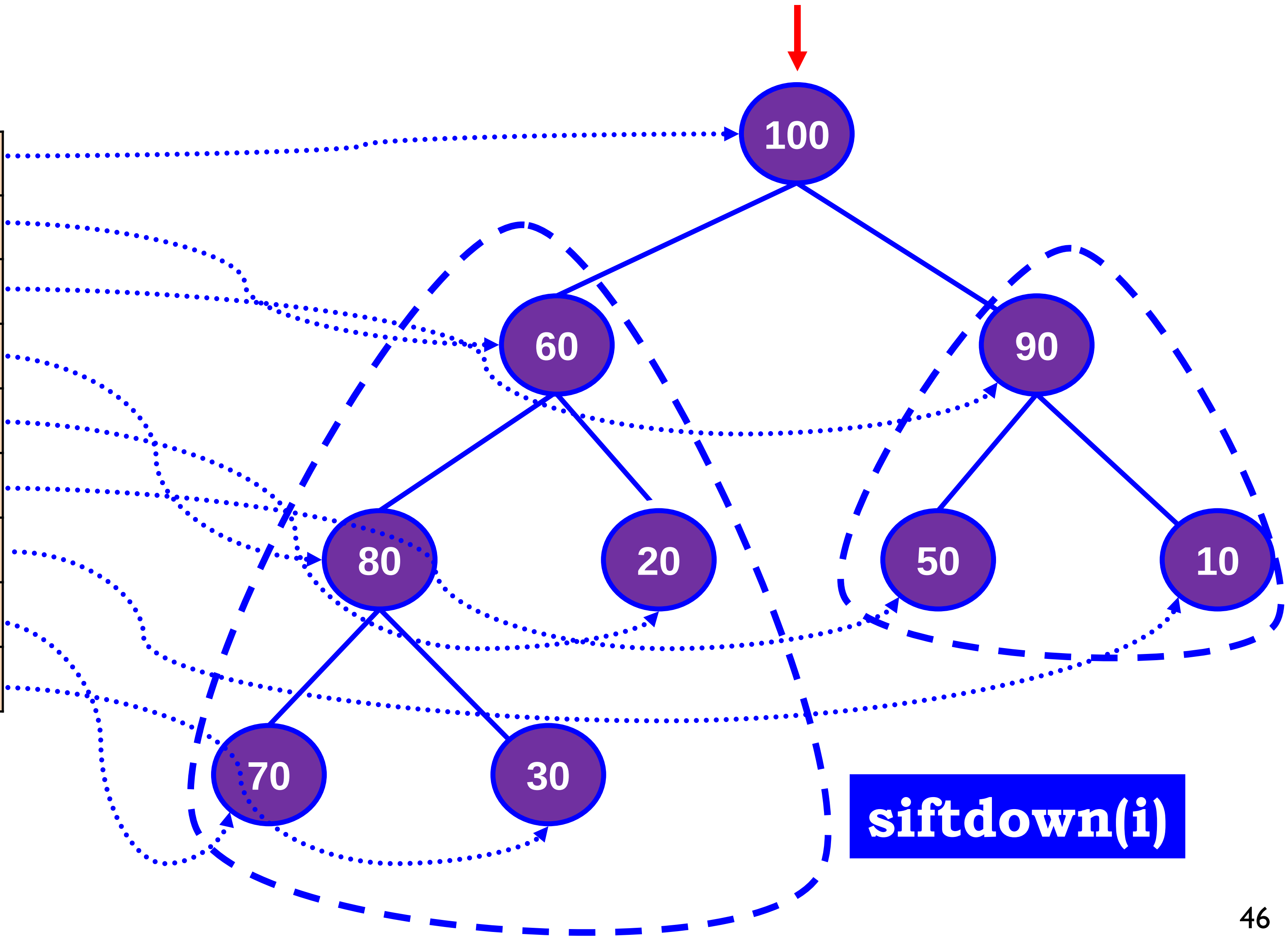


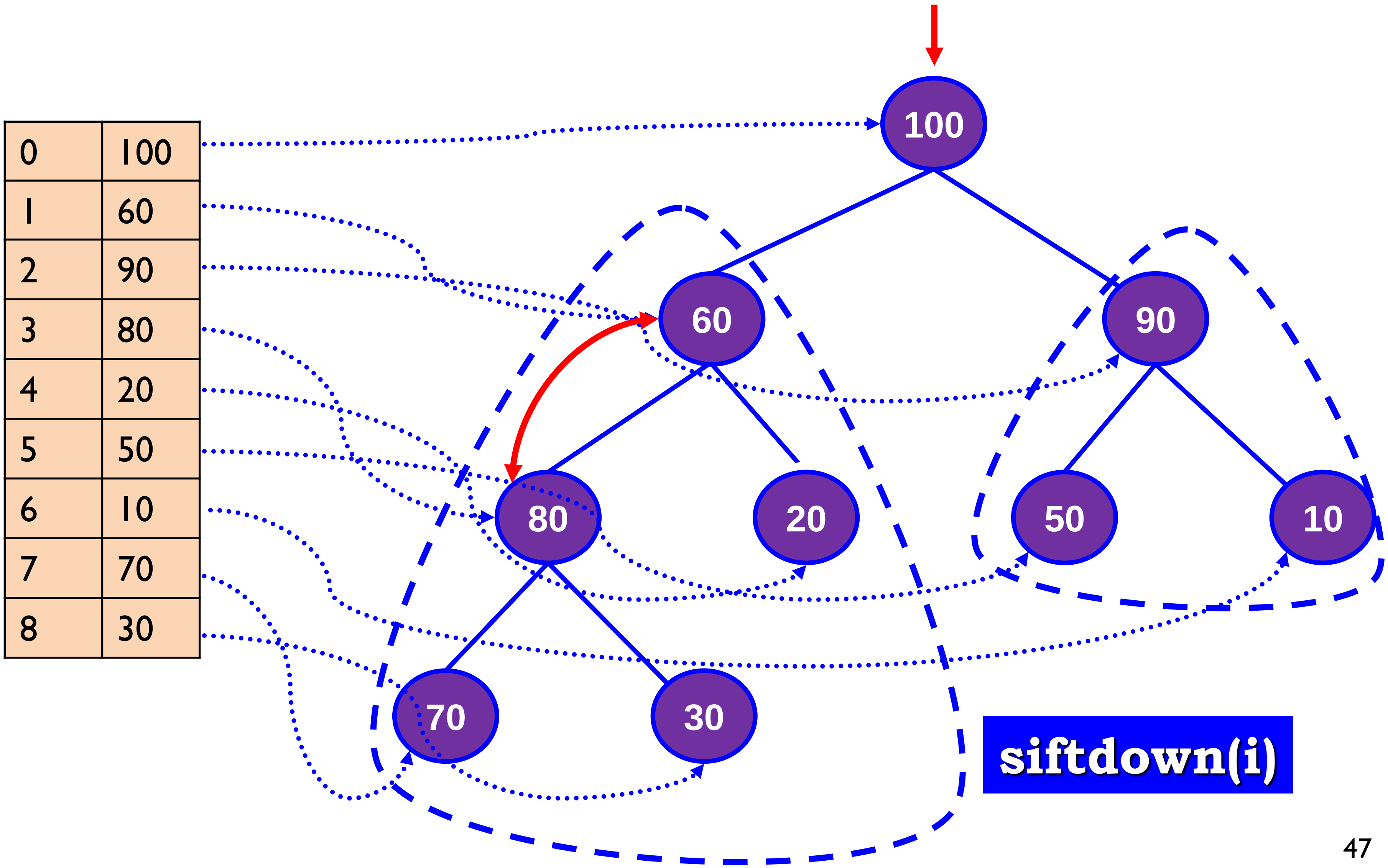
i--

0	60
1	100
2	90
3	80
4	20
5	50
6	10
7	70
8	30



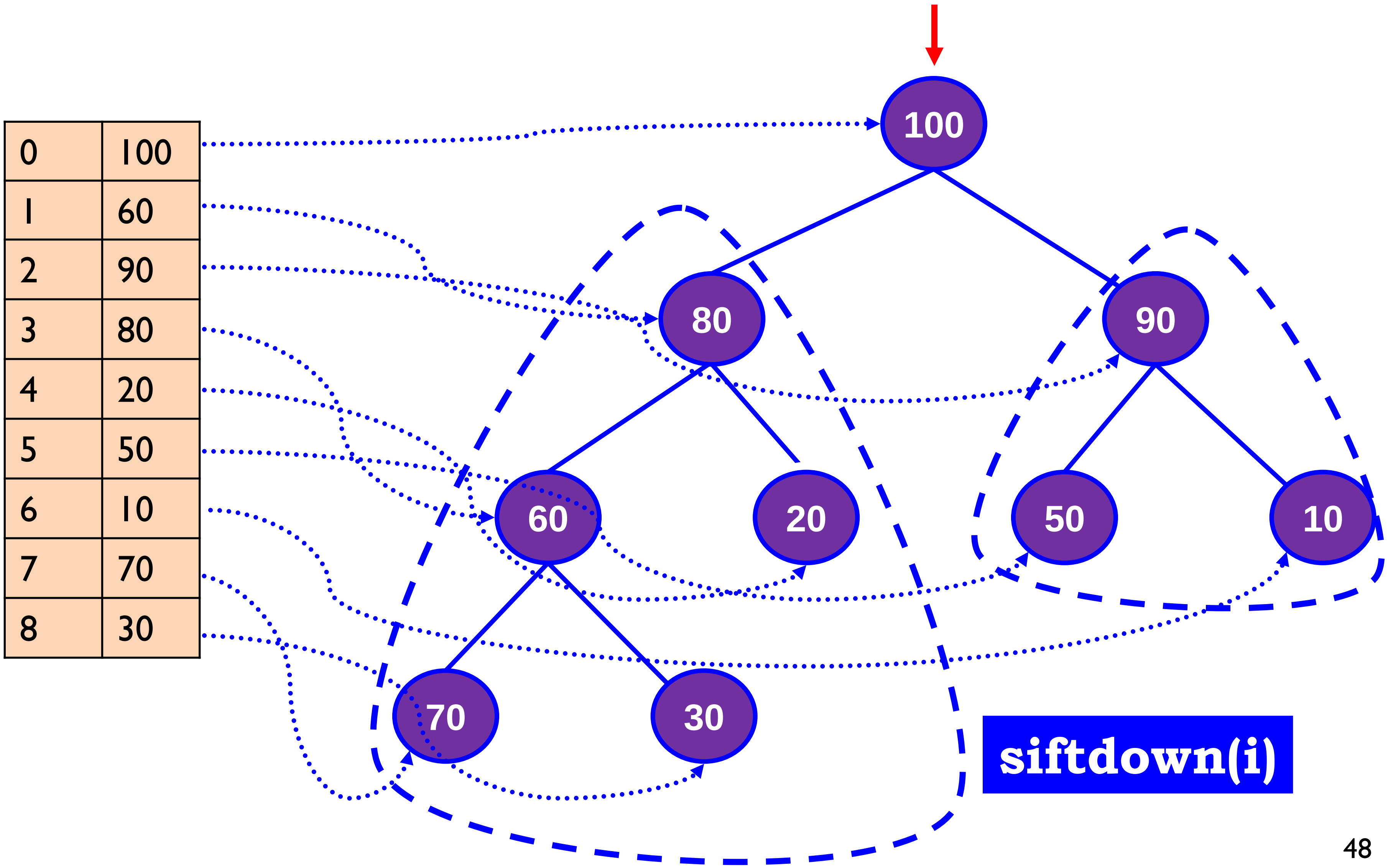
0	100
1	60
2	90
3	80
4	20
5	50
6	10
7	70
8	30



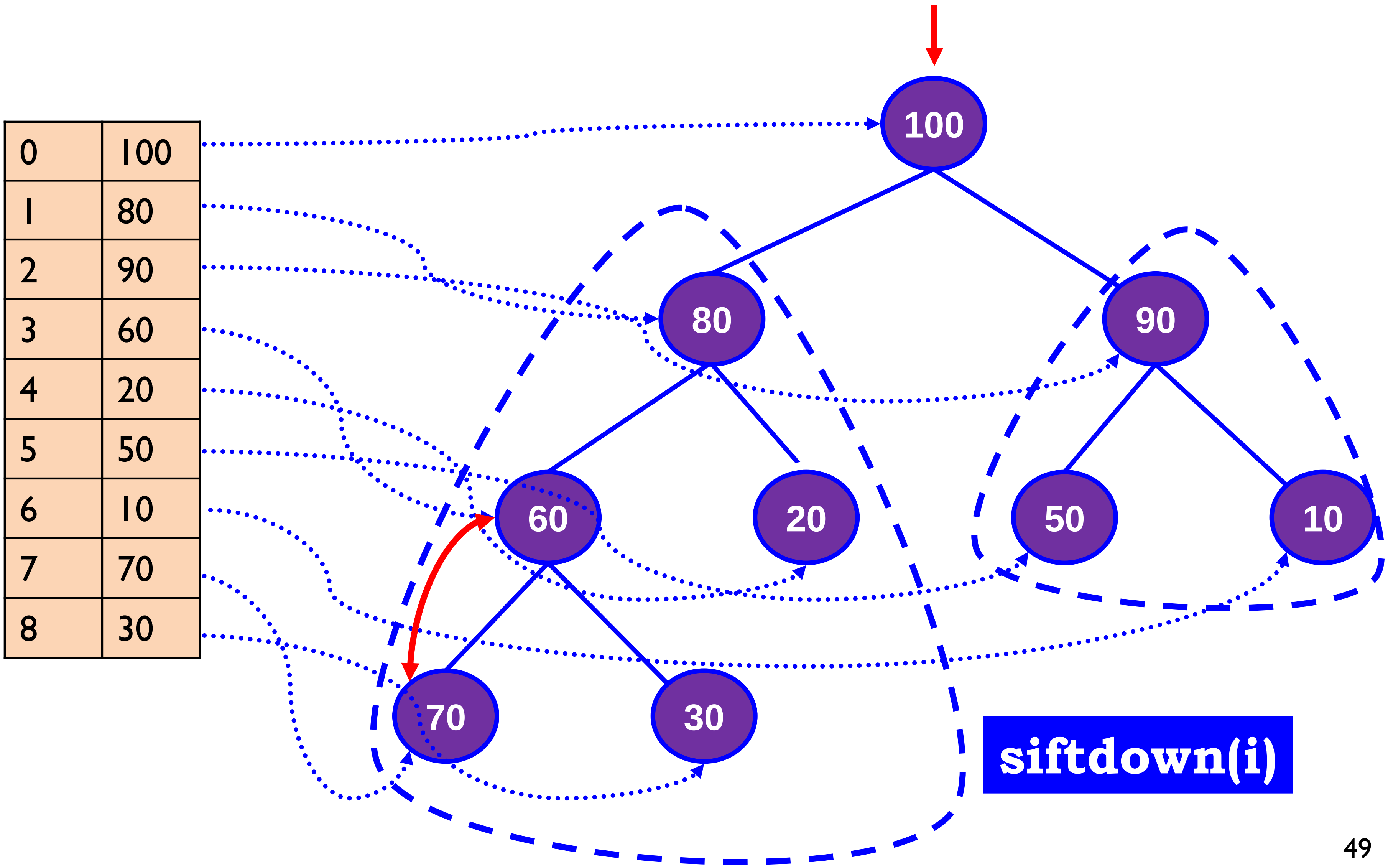


0	100
1	60
2	90
3	80
4	20
5	50
6	10
7	70
8	30

siftdown(i)

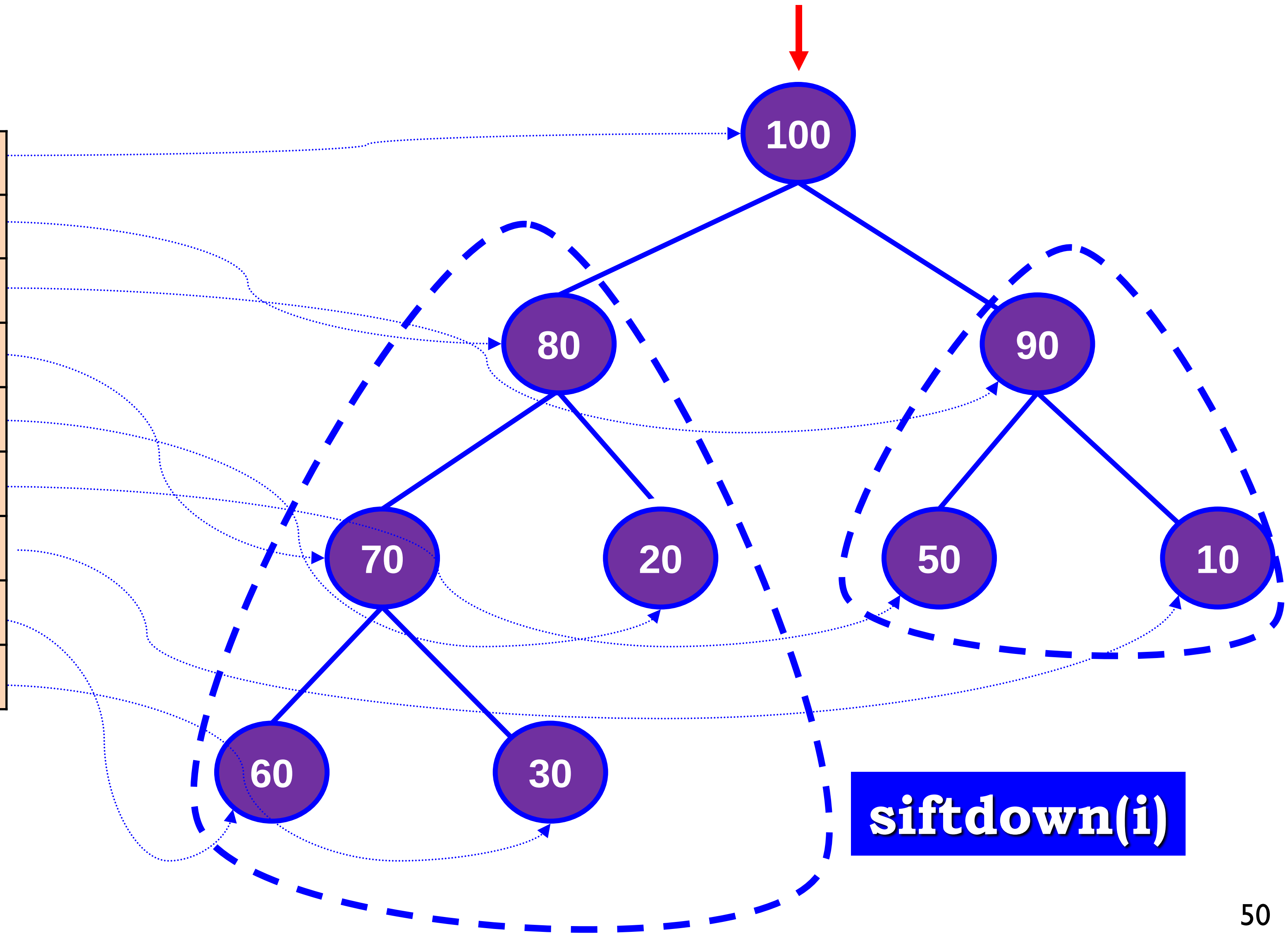


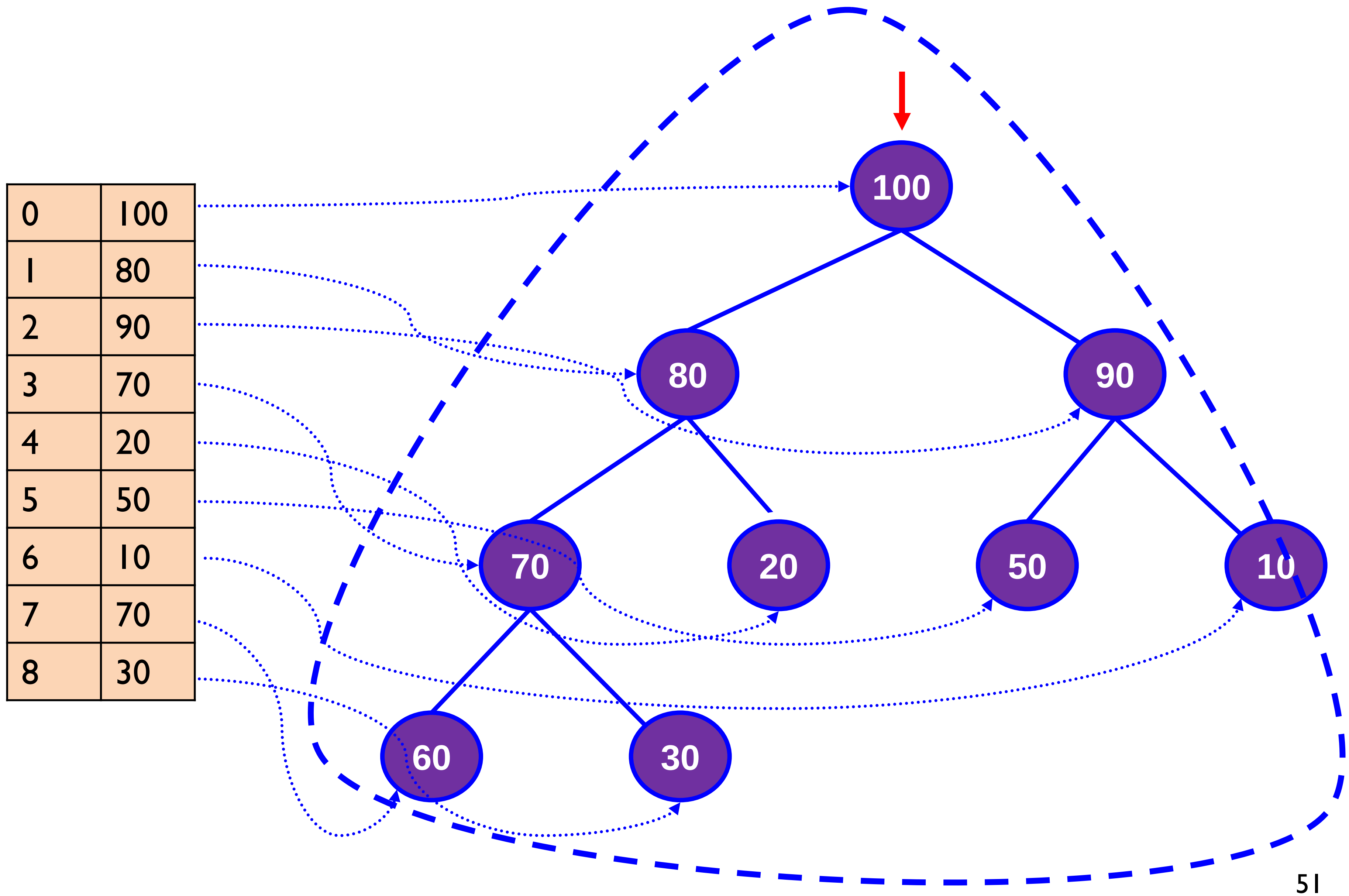
0	100
1	60
2	90
3	80
4	20
5	50
6	10
7	70
8	30



0	100
1	80
2	90
3	60
4	20
5	50
6	10
7	70
8	30

0	100
1	80
2	90
3	70
4	20
5	50
6	10
7	70
8	30





0	100
1	80
2	90
3	70
4	20
5	50
6	10
7	70
8	30

```
procedure ADJUST (var A: arraytype; i,n : integer);
//Complete binary tree with roots A[2*i] and A[2*i+1] are
combined with A[i] to form a heap  $1 \le i \le n$//
var j: integer; item: itemtype;
begin {ADJUST}
        j $\leftarrow$ 2*i; item $\leftarrow$ A[i];
        while ($j \le n$) do
        begin
        if ( $j < n$ and A[j]<A[j+1]) then  // Compare left and
                                                  right  child//
        j ← j+1; //j points to the larger child//
        if ( $item \geq A[j]$ ) then exit //a position for item is found//
        else
        begin
           A[ $\lfloor j/2 \rfloor$ ] ← A[j]; //move larger child up a level//
           j ← 2 * j;
        end;
A[ $\lfloor j/2 \rfloor$ ] ← item;
end {ADJUST}.
```

Given n elements, we can create heap by applying ADJUST procedure.

Since leaf nodes are already heap so we can start by calling ADJUST procedure for the parents of leaf nodes and work our way up, level by level, until we reach to the root.

```
procedure HEAPIFY(Var A: arraytype; n : integer);
// re-adjust the elements in A[1..n] to form a heap//
var  i : integer;
begin
        for i ← $\lfloor n/2 \rfloor$ downto 1 do
        ADJUST(A,i,n);
end.
```

Worse case analysis of HEAPIFY

Let $2^k - 1 \leq n < 2^k$, $k = \lceil \log_2(n+1) \rceil$

Number of iterations of ADJUST for the worse case for a node at level ***i*** is **(*k*-*i*)**, so the total time for HEAPIFY is proportional to -

$$\sum_{i=1}^{k} 2^{i-1}.(k-i)$$

$$= \left[k \sum_{i=1}^{k} 2^{i-1} - \sum_{i=1}^{k} i.2^{i-1} \right]$$

$$= \left[k \left[\frac{2^k - 1}{2 - 1} \right] - [(k+1).2^k - 2^{k+1} + 1] \right]$$

$$= \left[k2^k - k - [k.2^k + 2^k - 2^{k+1} + 1] \right]$$

$$= \left(-2^k - k + 2^{k+1} - 1 \right)$$

$$= \left(2^{k+1} - 2^k - k - 1 \right)$$

$$= \left(2^k - k - 1 \right)$$

$$= \left(2^{\lceil \log(n+1) \rceil} - \lceil \log(n+1) \rceil - 1 \right)$$

$$\leq 2(n+1) - (\log(n+1) + 1) - 1$$

$$< 2(n+1) - (\log(n) + 1) - 1, n \geq 2$$

$$= 2n - \log n$$

$$= O(n)$$

Comparison:	INSERT	HEAPIFY
best case	O(n)	O(n)
worse case	O(n log n)	O(n)
elements available before heap creation	not necessary	necessary

Heap sort: The most important example of the use of heap arises in the application of the heap sort.

What is Simple strategy?

It is to continuously remove the maximum value from the remaining unsorted elements **- bubble sort O(n^2)**

- **Bubble sort** starts at the end of the array, compares to nearest neighbours and if the second one is smaller it swaps them. Thus, when it finds the smallest element it keeps on swapping it until it gets to the position 1. Algorithm continues in the loop, “bubbling up” the smallest elements in each pass.

Continued in the next presentation