CS 261 – Data Structures

Priority Queue ADT & Heaps

Priority Queue ADT

- Associates a "priority" with each object:
  - First element has the highest priority (typically, lowest value)
- Examples of priority queues:
  - To-do list with priorities
  - Active processes in an OS

Priority Queue Interface

```
void  add(newValue);

TYPE  getFirst();

void  removeFirst();
```

Priority Queues: Implementations

	`SortedVector`	`SortedList`	`LinkedList`
`add`	O(n) Binary search Slide data up	O(n) Linear search	O(1) **`addLast(obj)`**
`getFirst`	O(1) **`elementAt(0)`**	O(1) Returns **`head.obj`**	O(n) Linear search for smallest value
`removeFirst`	O(n) Slide data down O(1) → Reverse Order	O(1) **`head.remove()`**	O(n) Linear search then remove smallest

We can definitely do better than these!!!

What about a Skip List?

Heap Data Structure

Heap: has 2 completely different meanings:

1. Data structure for priority queues
2. Memory space for dynamic allocation

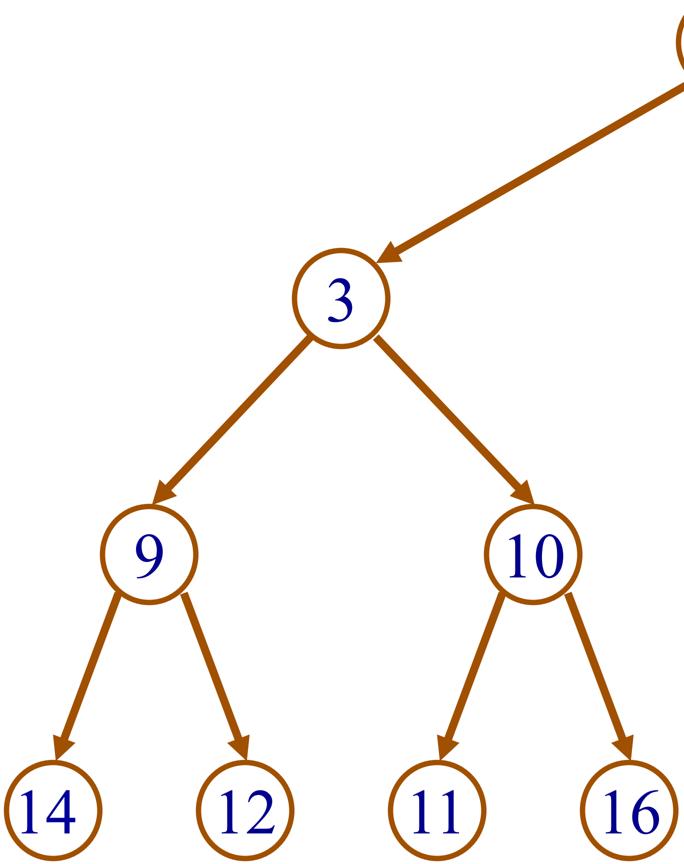
We will study the data structure

(not dynamic memory allocation)

Heap Data Structure

Heap = C*omplete* binary tree in which every node's value $\leq$ the children values

Heap: **Example**



Root = Smallest element

Next open spot

Last filled position

(not necessarily the last object added)

Complete Binary Tree

1. Every node has at most two children (binary)

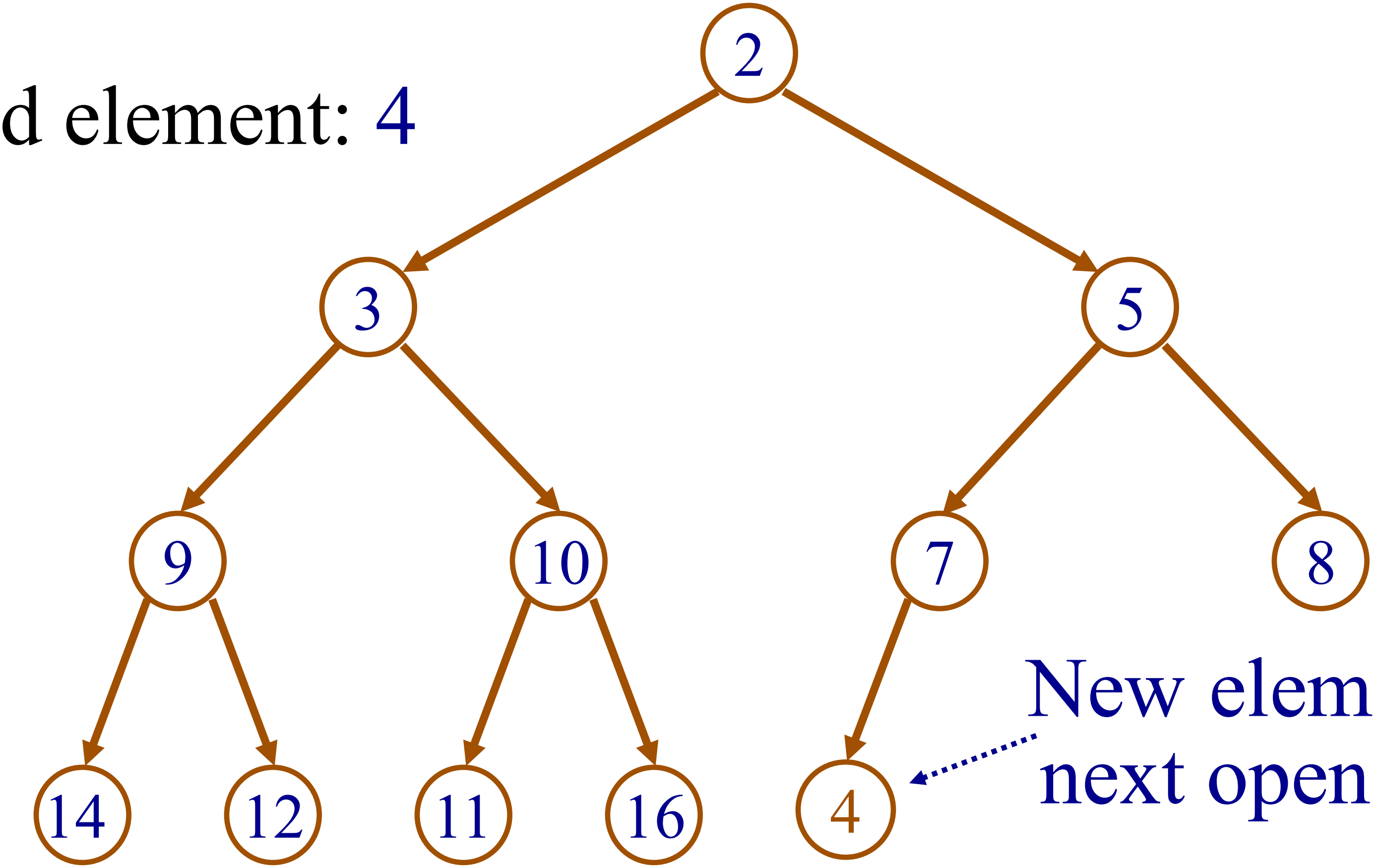
2. Children have an arbitrary order

3. Completely filled except for the bottom level, which is filled from left to right (complete)

4. Longest path is ceiling(log n) for n nodes

Maintaining the Heap: Addition

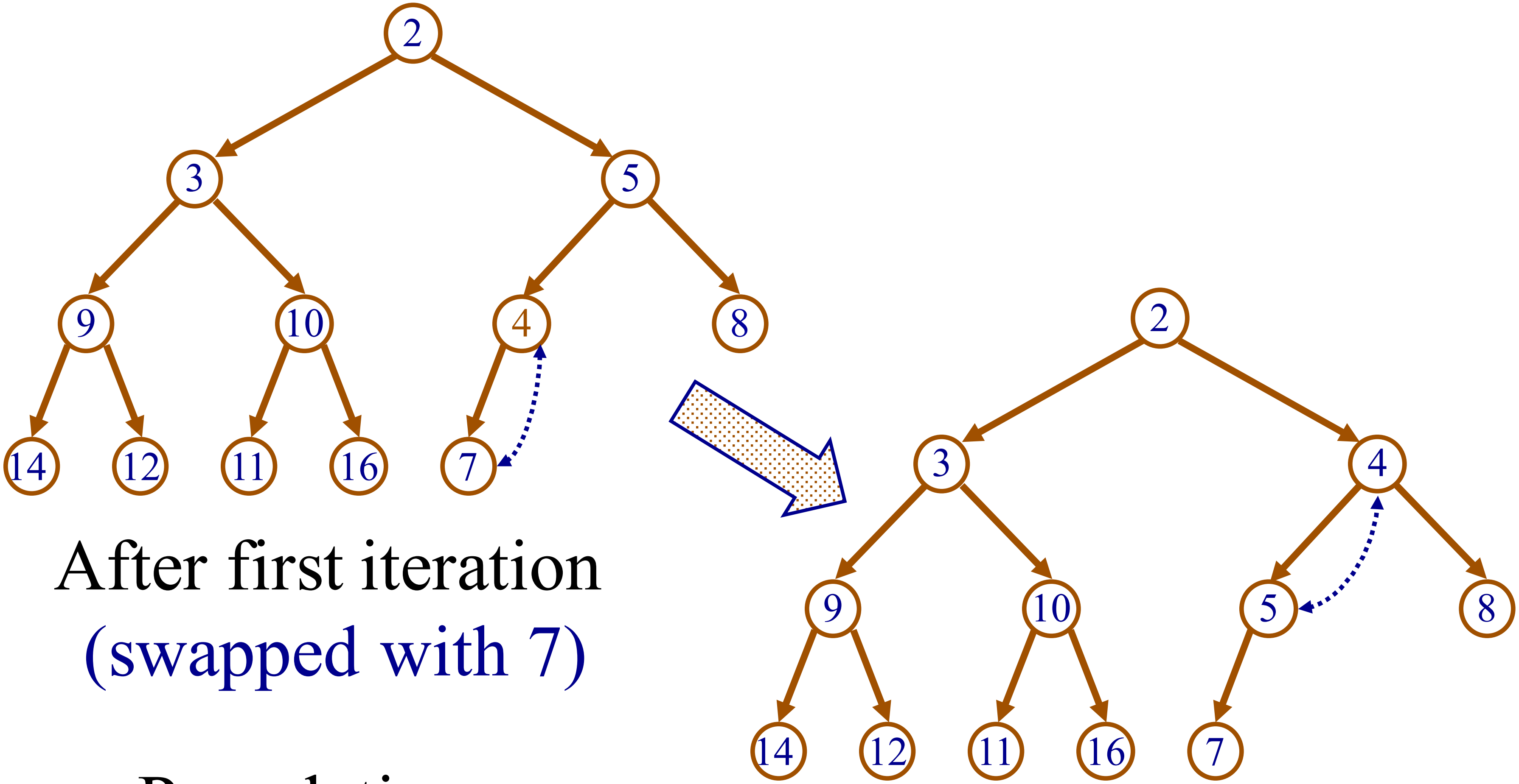
Add element: 4



New element in next open spot.

Place new element in next available position, then fix it by “percolating up”

Maintaining the Heap: **Addition (cont.)**



After first iteration
(swapped with 7)

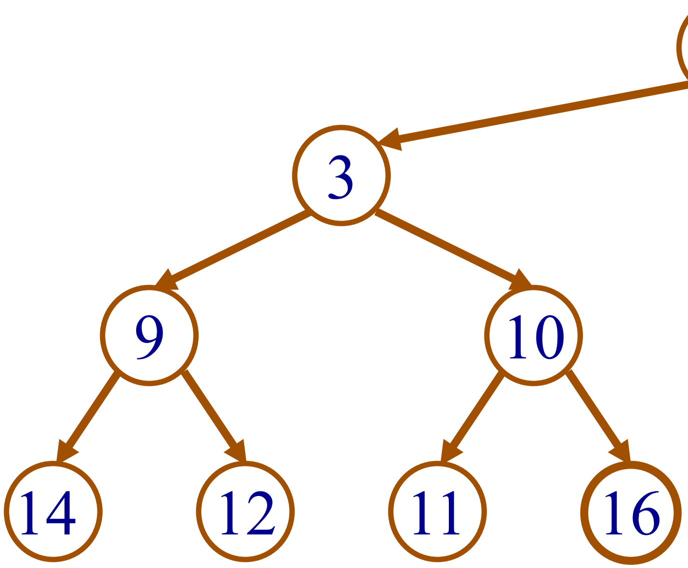
Percolating up:
while new value < parent,
swap value with parent

After second iteration
(swapped with 5)

Maintaining the Heap: Removal

The root is always the smallest element

➔ **`getFirst`** and **`removeFirst`** access and remove the root node



Which node becomes the root?

Maintaining the Heap: Removal

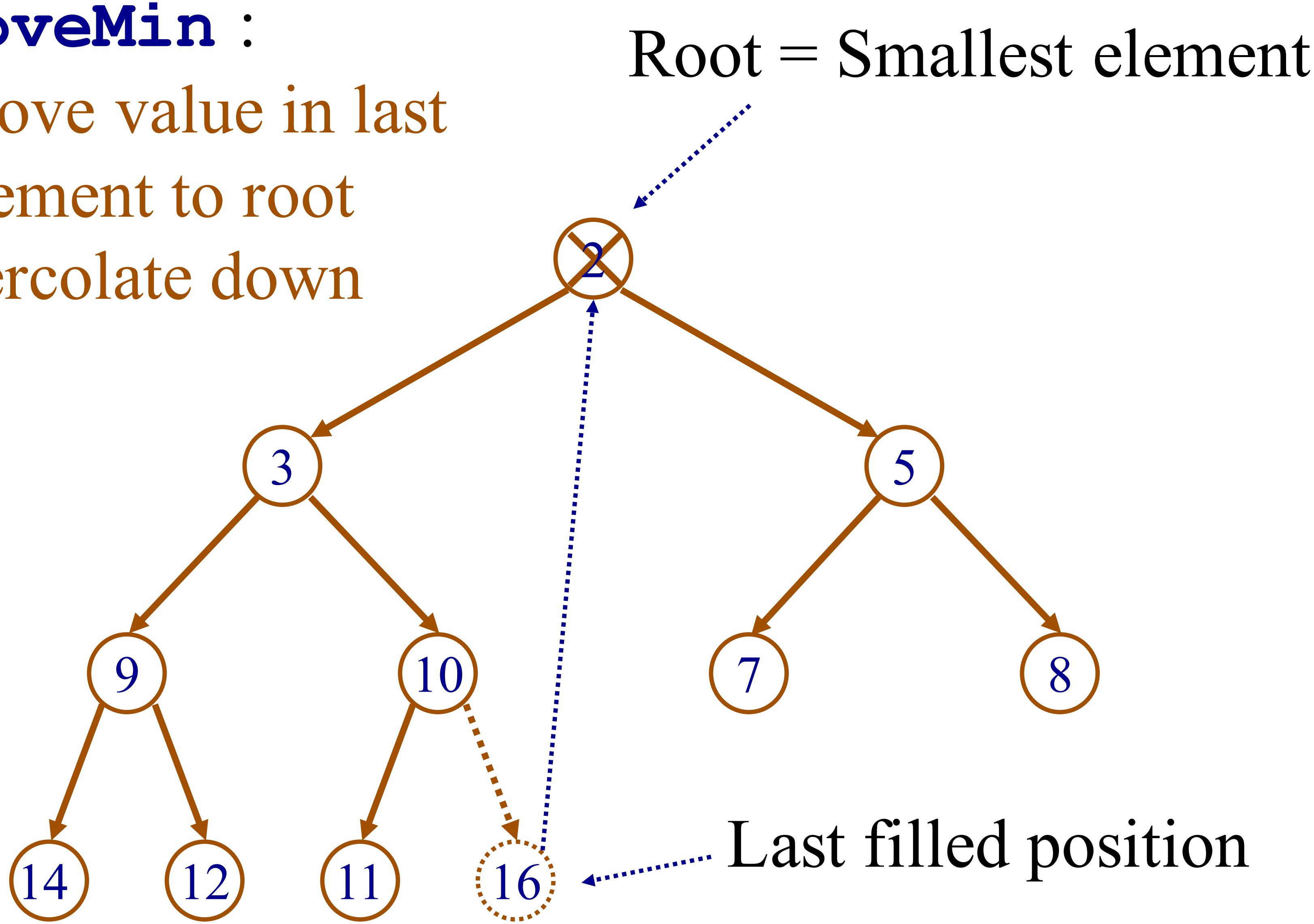
1. Replace the root with the element in the last filled position

2. Fix heap by “percolating down”

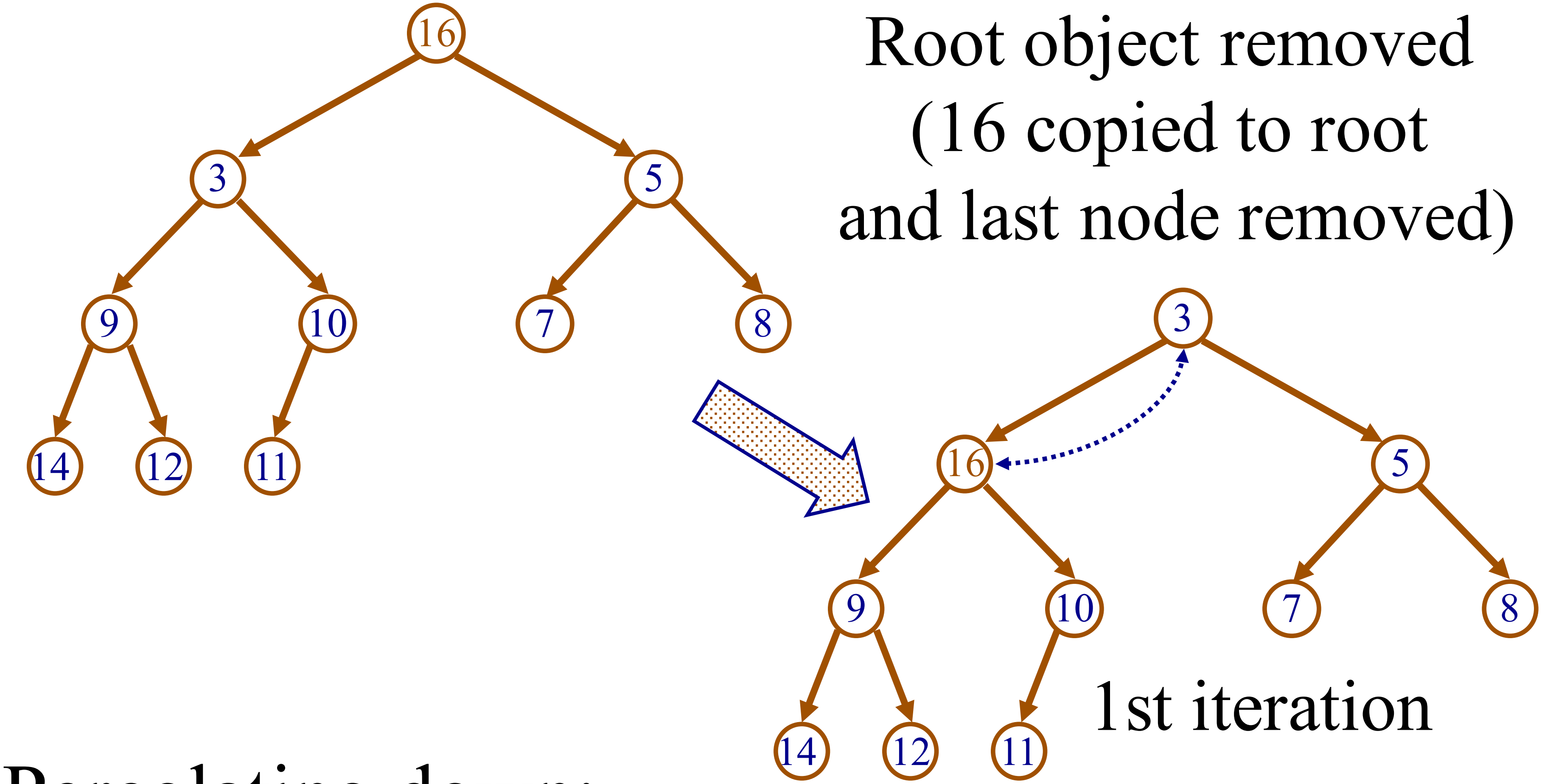
Maintaining the Heap: Removal

`removeMin` :

1. Move value in last element to root
2. Percolate down



Maintaining the Heap: Removal (cont.)

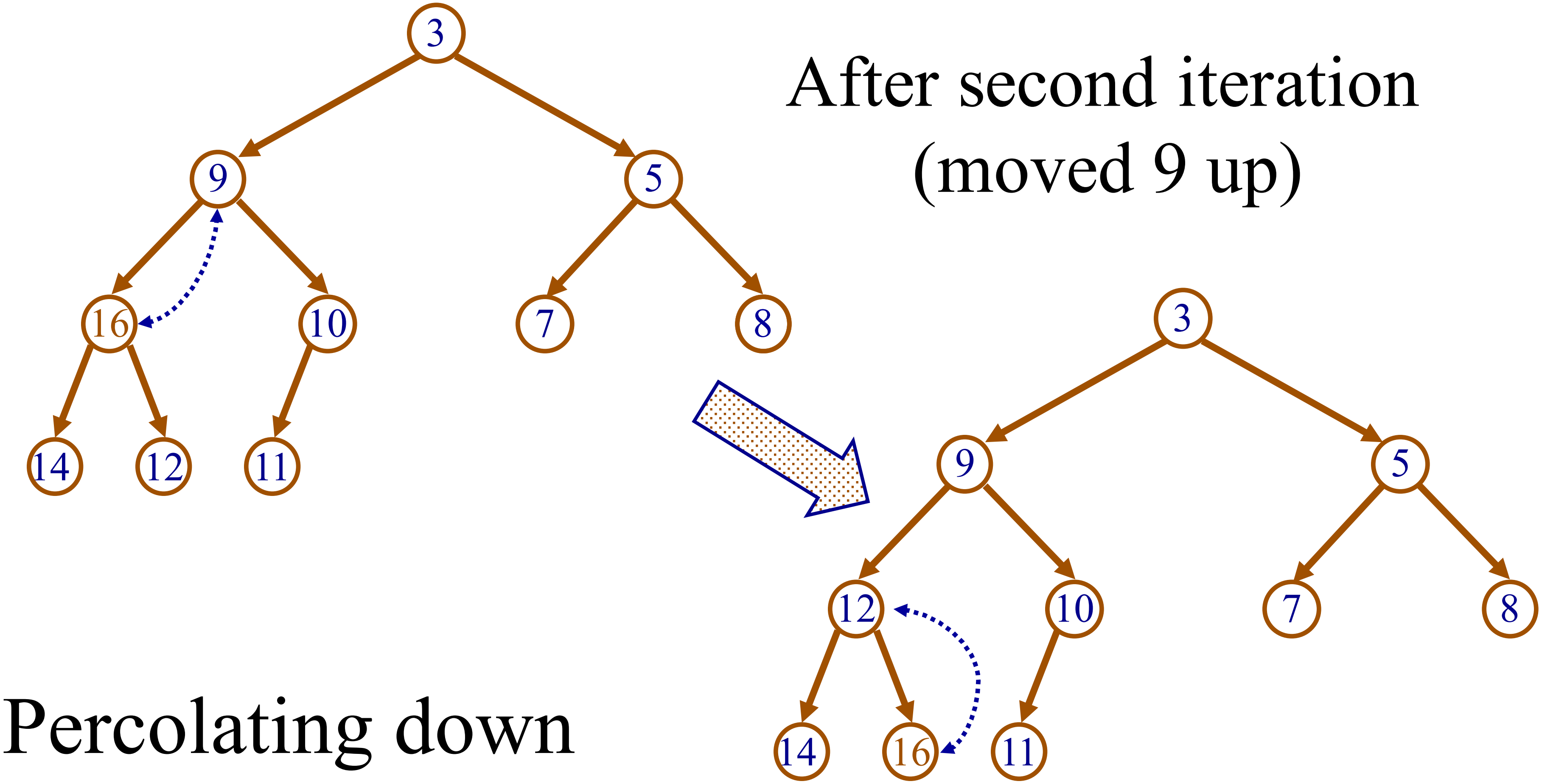


Root object removed
(16 copied to root
and last node removed)

1st iteration

Percolating down:
while new root > smallest child
swap with smallest child

Maintaining the Heap: Removal (cont.)

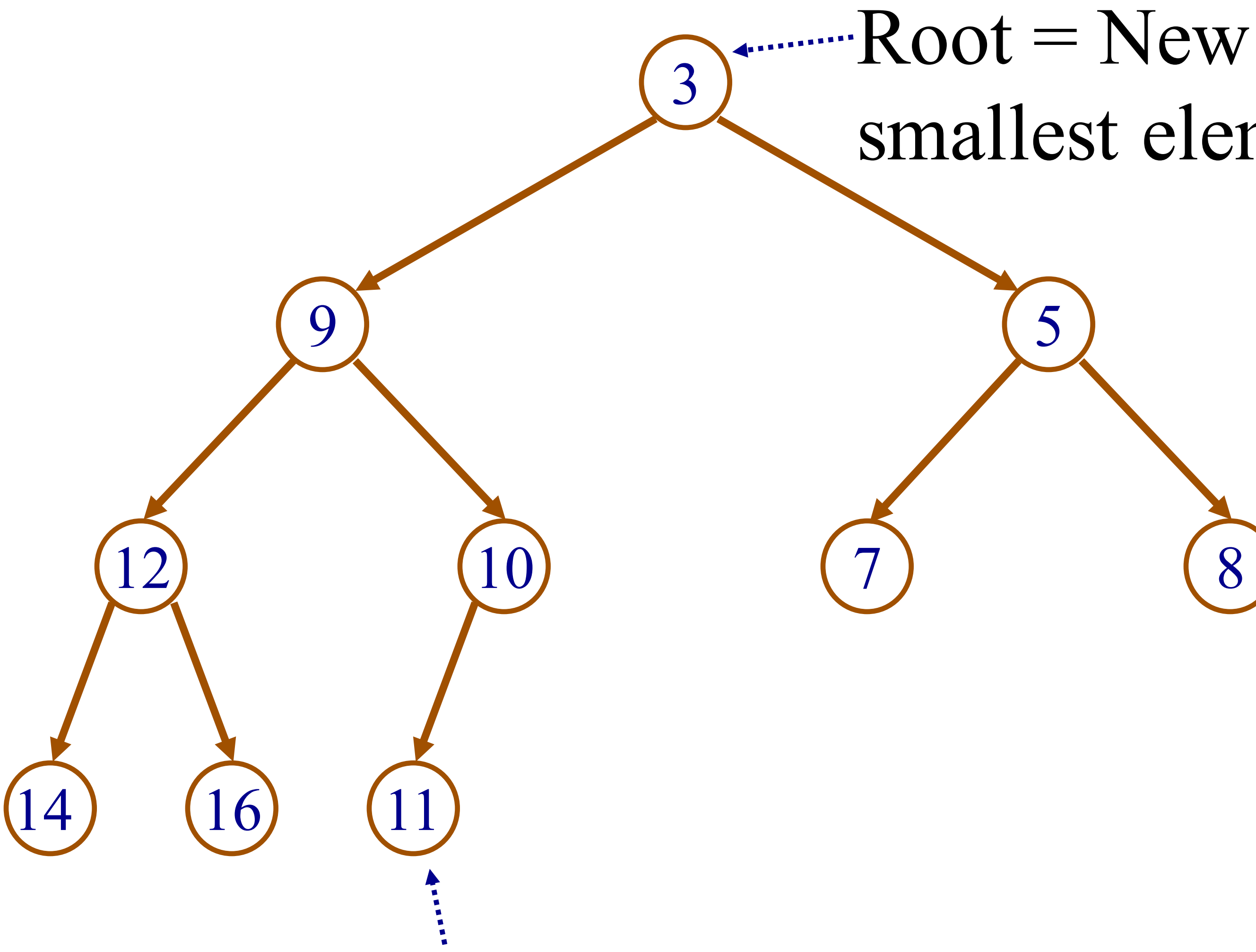


After second iteration
(moved 9 up)

Percolating down

After third iteration
(moved 12 up)

Maintaining the Heap: **Removal (cont.)**



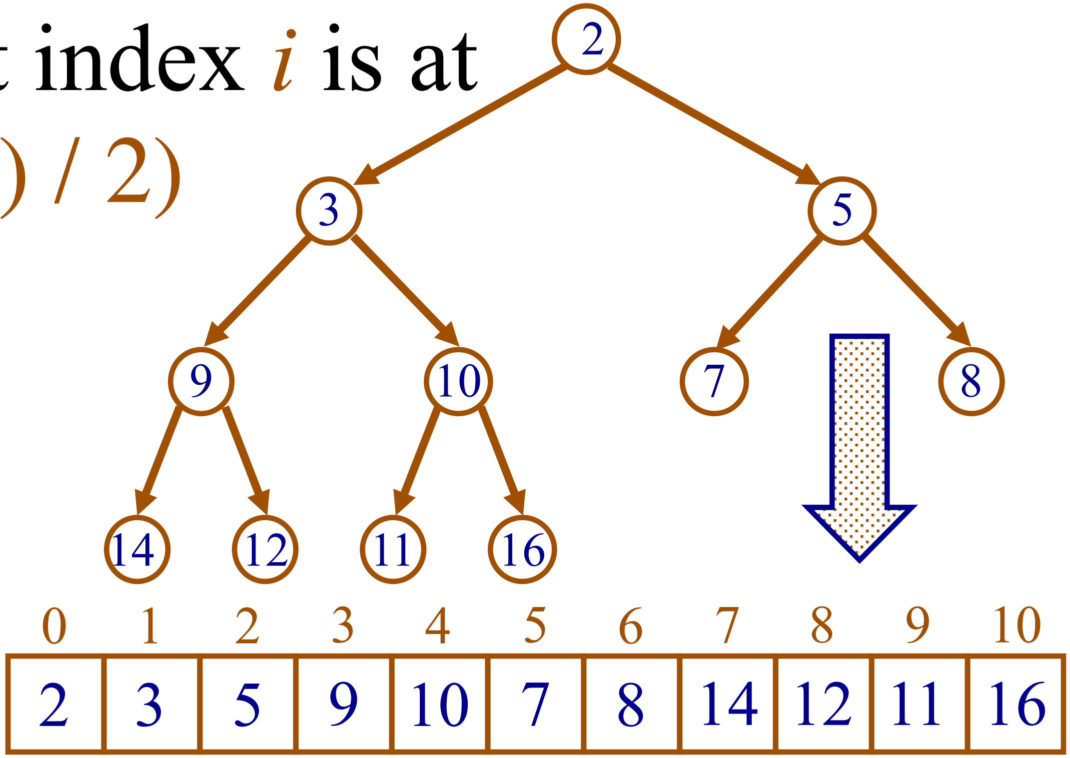
Root = New smallest element

New last filled position

Heap Representation as Dynamic Array

- Children of a node at index *i* are stored at indices $2i + 1$ and $2i + 2$

- Parent of node at index *i* is at index $\mathrm{floor}((i - 1) / 2)$

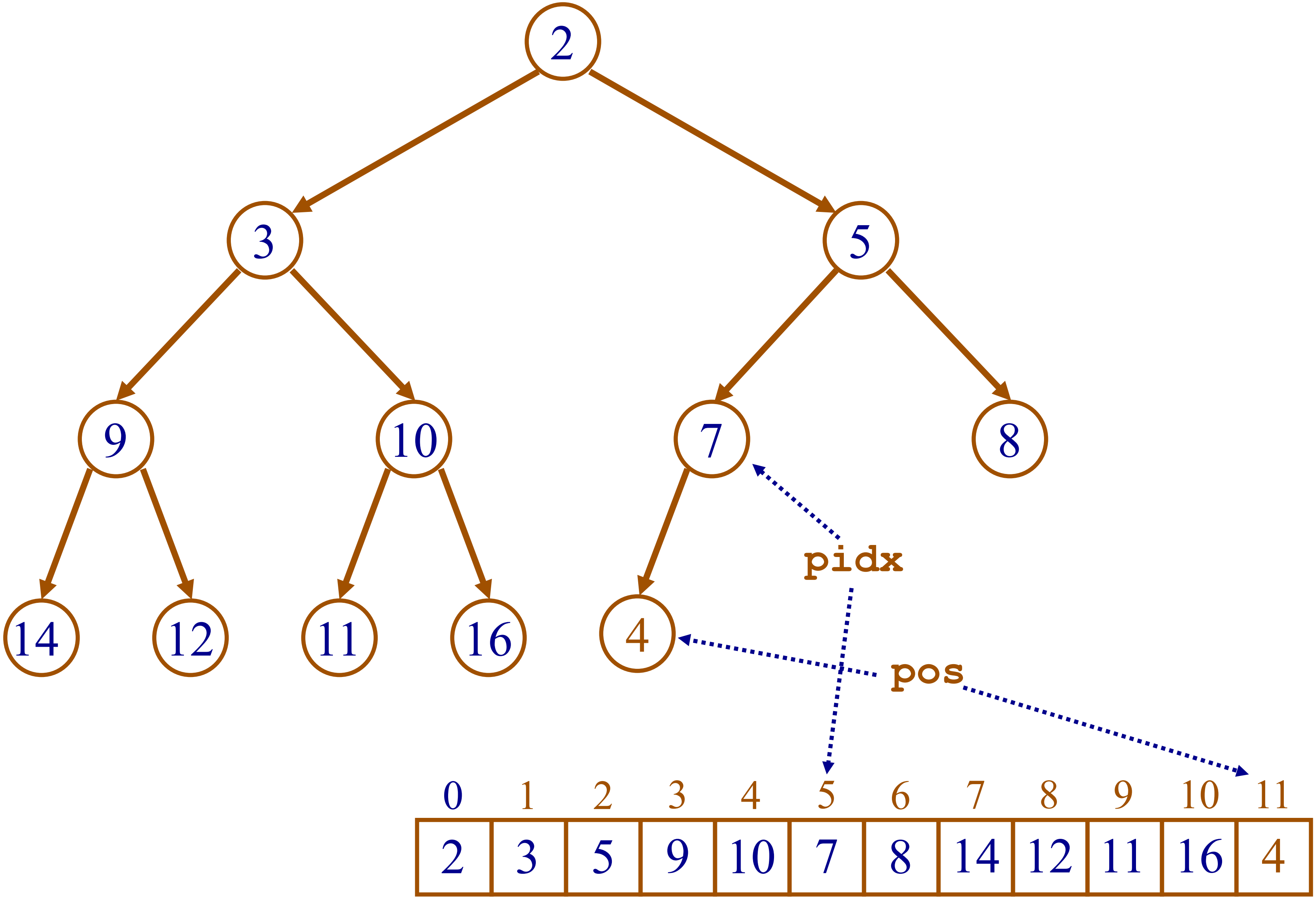


0	1	2	3	4	5	6	7	8	9	10
2	3	5	9	10	7	8	14	12	11	16

Creating New Abstraction

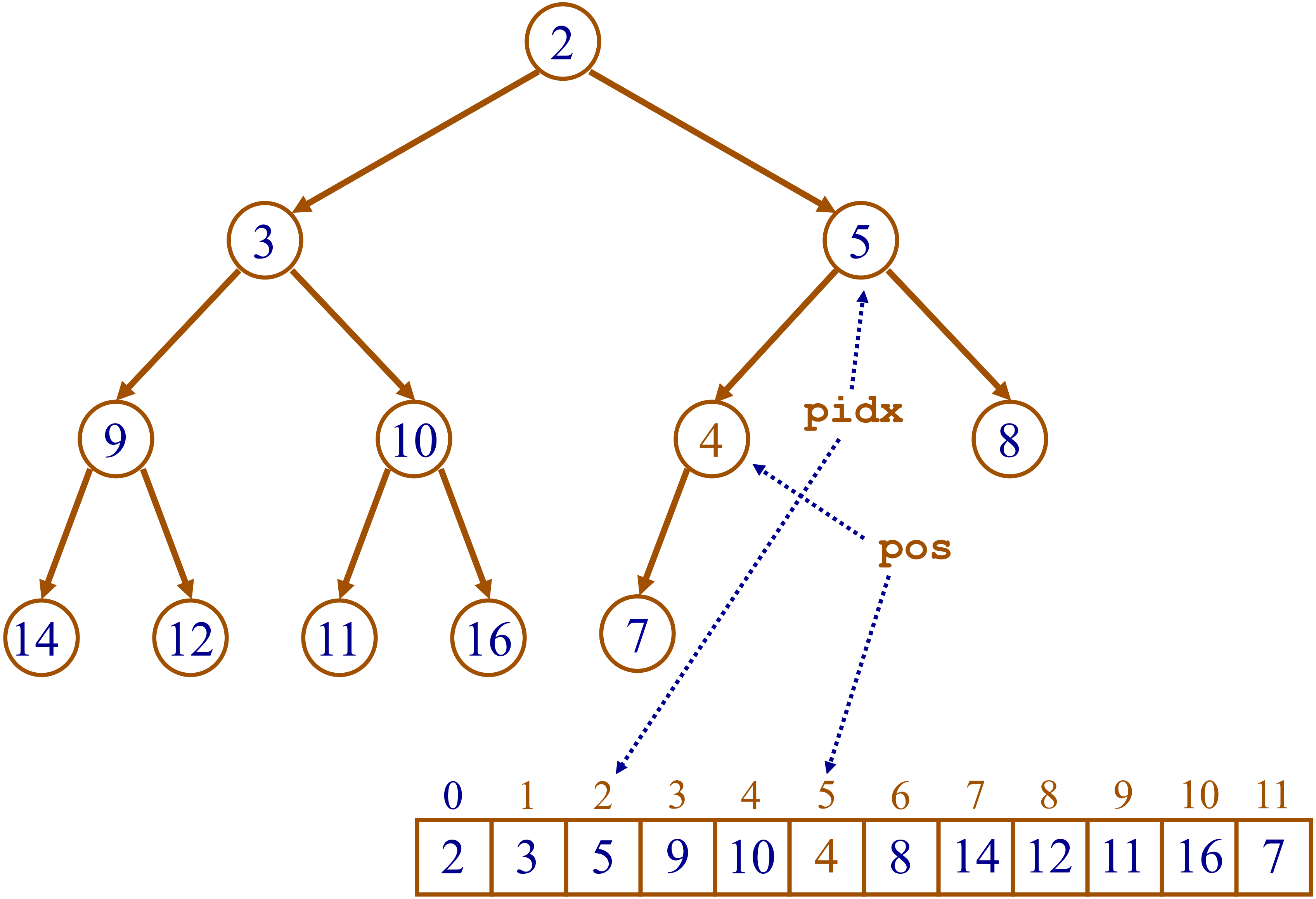
```
struct DynArr {
  TYPE *data;
  int size;
  int capacity;
};

struct DynArr *heap;
```

Implementation: Add 4 to the Heap



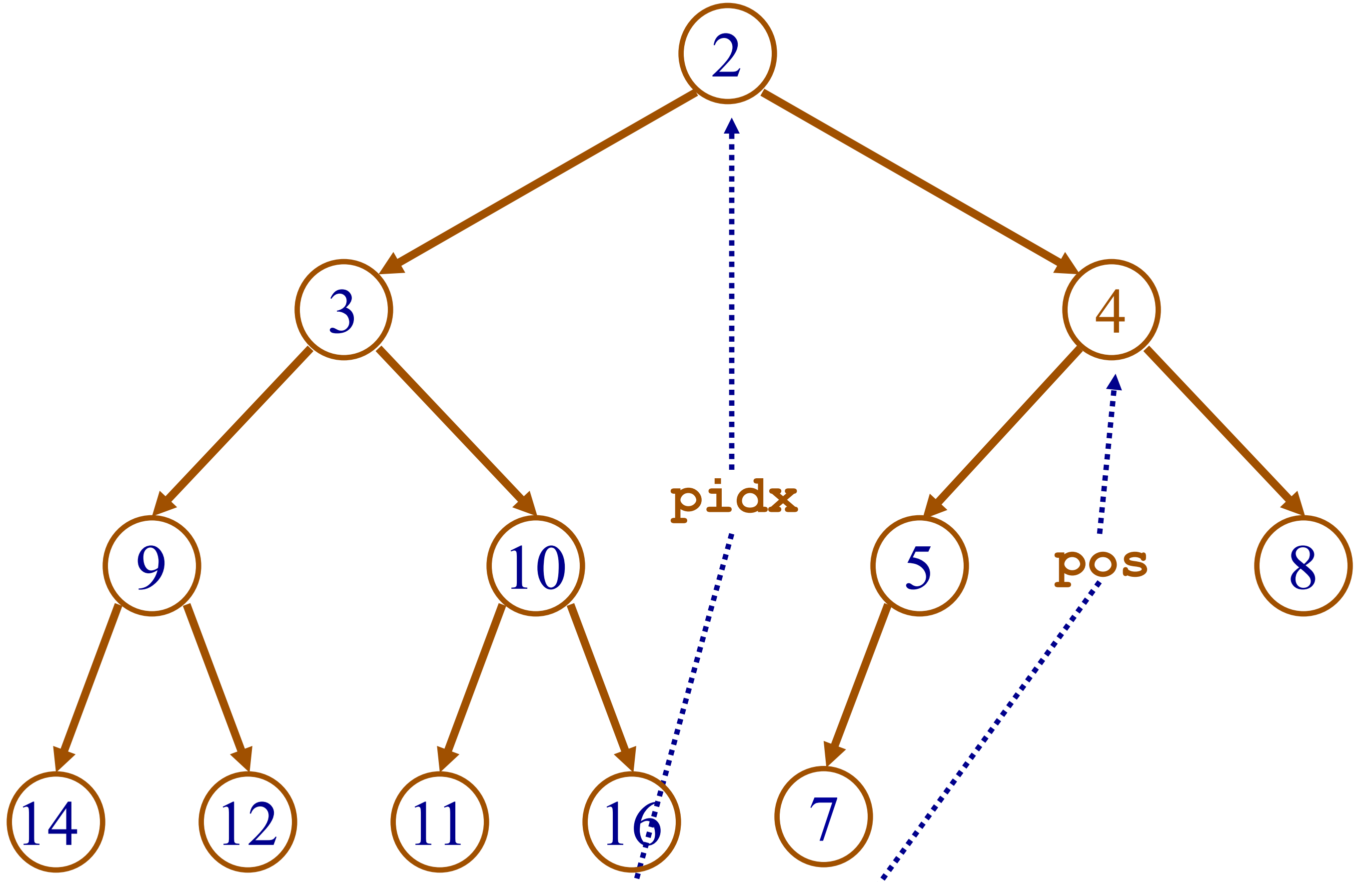
0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	9	10	7	8	14	12	11	16	4

Heap Implementation: **add (cont.)**



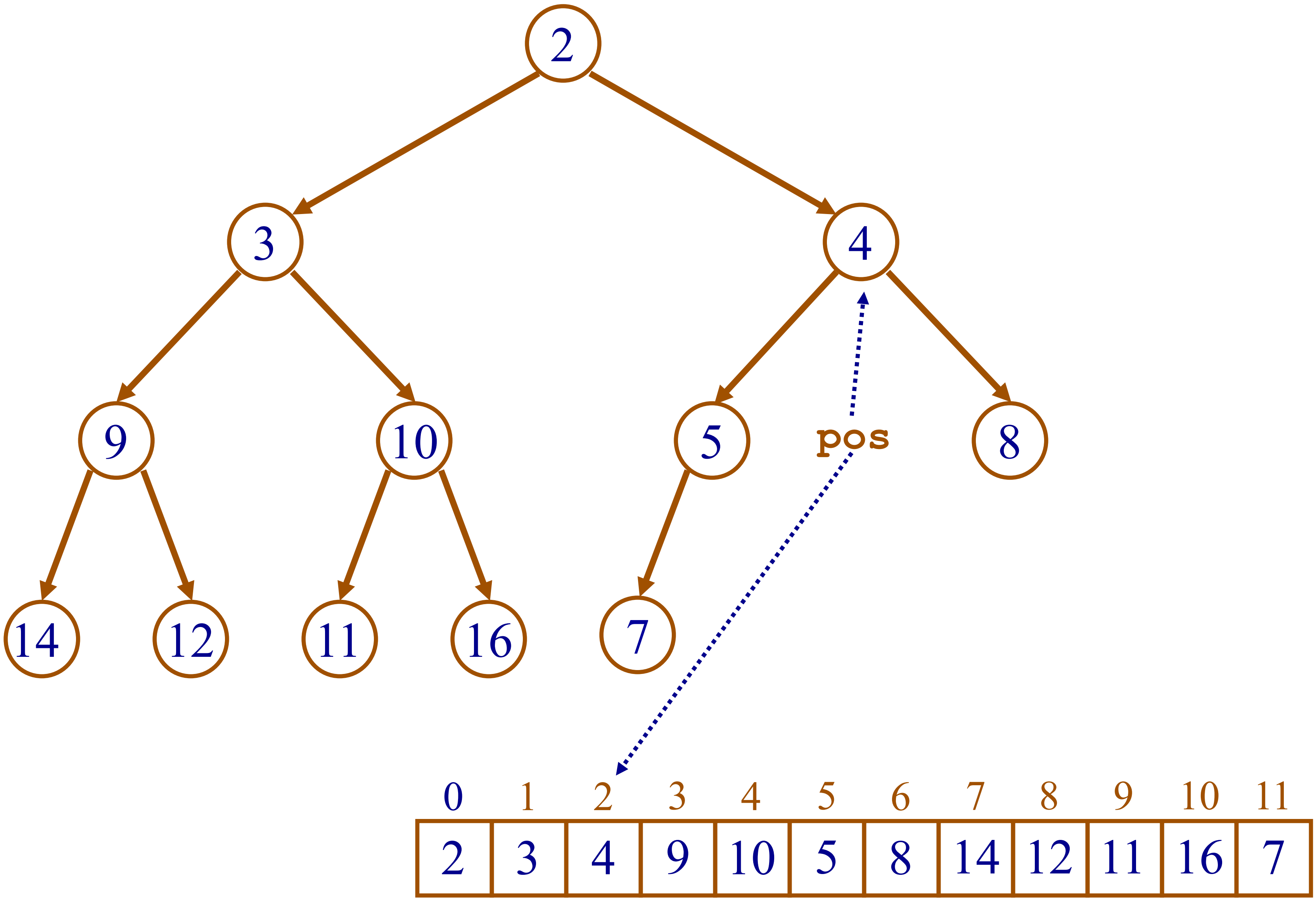
0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	9	10	4	8	14	12	11	16	7

Heap Implementation: add (cont.)



0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	9	10	5	8	14	12	11	16	7

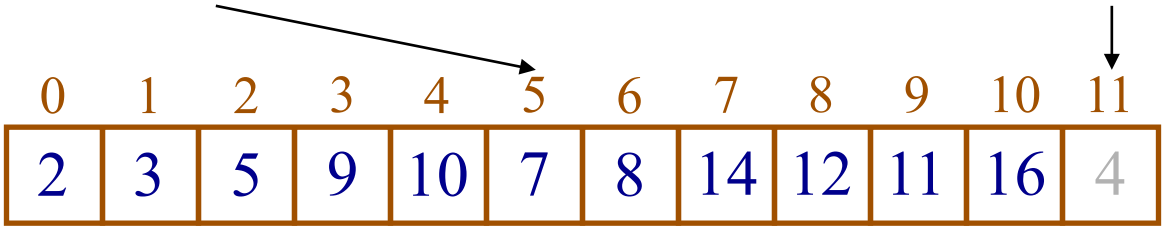
Heap Implementation: add (cont.)



0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	9	10	5	8	14	12	11	16	7

Heap Implementation: Add

```
void addHeap(struct DynArr *heap, TYPE val) {
    assert(heap);
    int pos = heap->size;  /* next open spot            */
    int pidx =(pos - 1)/2;          /* Get parent index */
    TYPE parentVal;
     ...
}
```



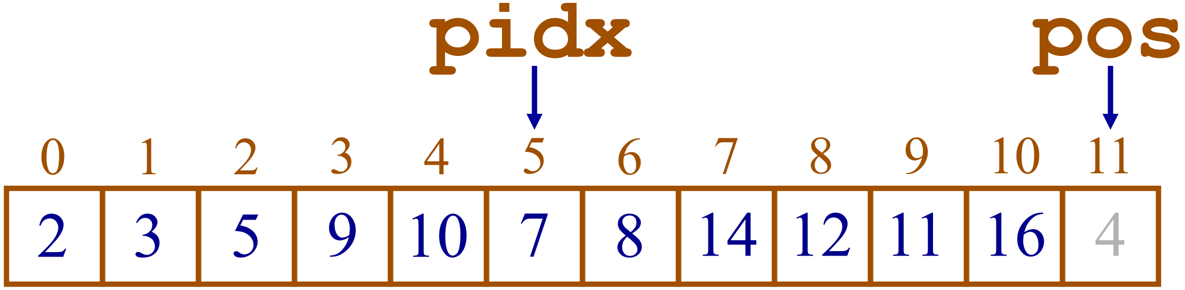
Parent position: **`pidx`**

Next open spot: **`pos`**

0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	9	10	7	8	14	12	11	16	4

Heap Implementation: add (cont.)

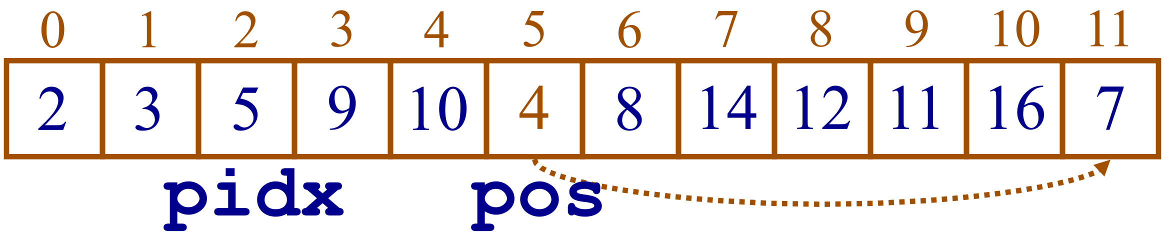
```
void addHeap(struct DynArr *heap, TYPE val) {
   ...
   /* Make room for the new element */
  if (pos >= heap->cap)
        _doubleCapacity(heap, 2 * heap->cap);
  ...
 }
```



0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	9	10	7	8	14	12	11	16	4

Heap Implementation: add (cont.)

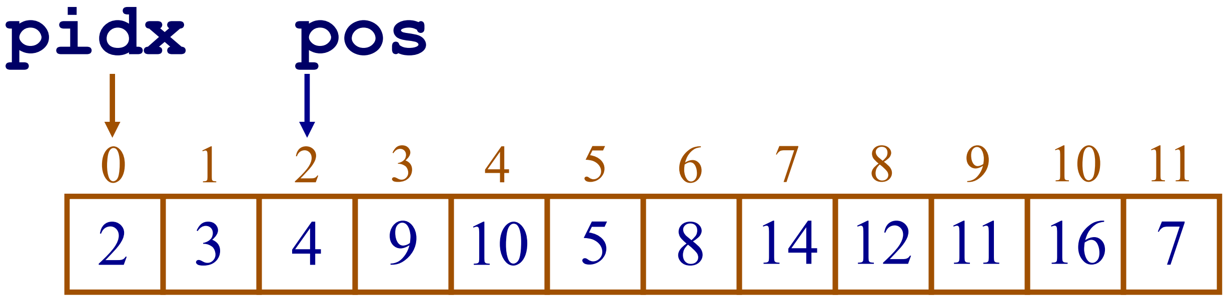
```
void addHeap(struct DynArr *heap, TYPE val) {
    ...
   /* While not at root and new value is less than parent */
   if (pos > 0) parentVal = heap->data[pidx];
   while (pos > 0 && LT(val, parentVal)) {
      /* Percolate upwards */
      heap->data[pos] = parentVal;
      pos = pidx;
      pidx  = (pos - 1) / 2; /* new parent index */
      if (pos > 0) parentVal = heap->data[pidx];

   }

  ...
}
```



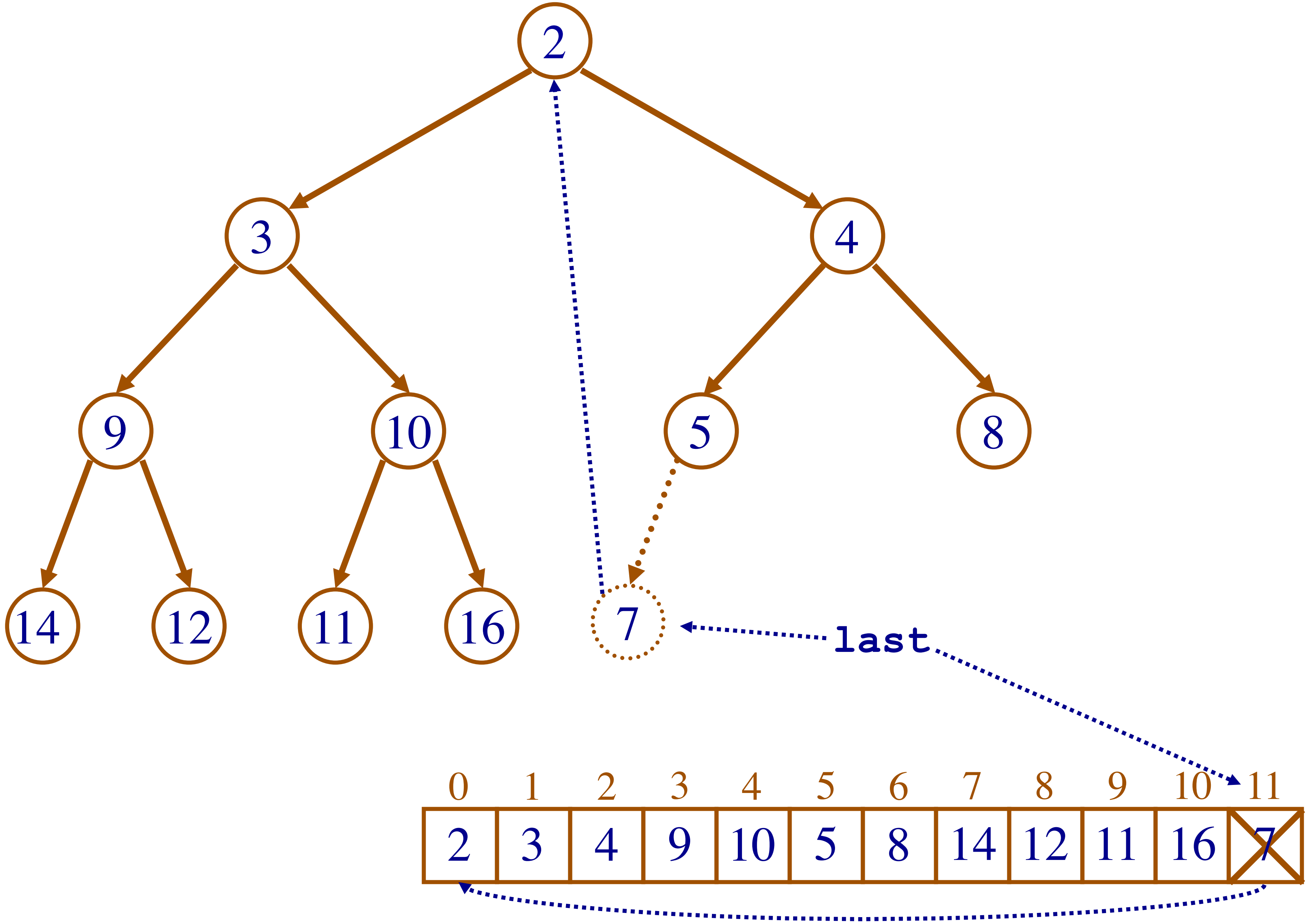
0	1	2	3	4	5	6	7	8	9	10	11
2	3	5	9	10	4	8	14	12	11	16	7

Heap Implementation: add (cont.)

```
void addHeap(struct DynArr *heap, TYPE val) {
    ...
  while (pos > 0 && LT(val, parentVal)) {
    /* percolate upwards */
  }
  heap->data[pos] = val;/*Now add new value*/

   heap->size++;
}
```

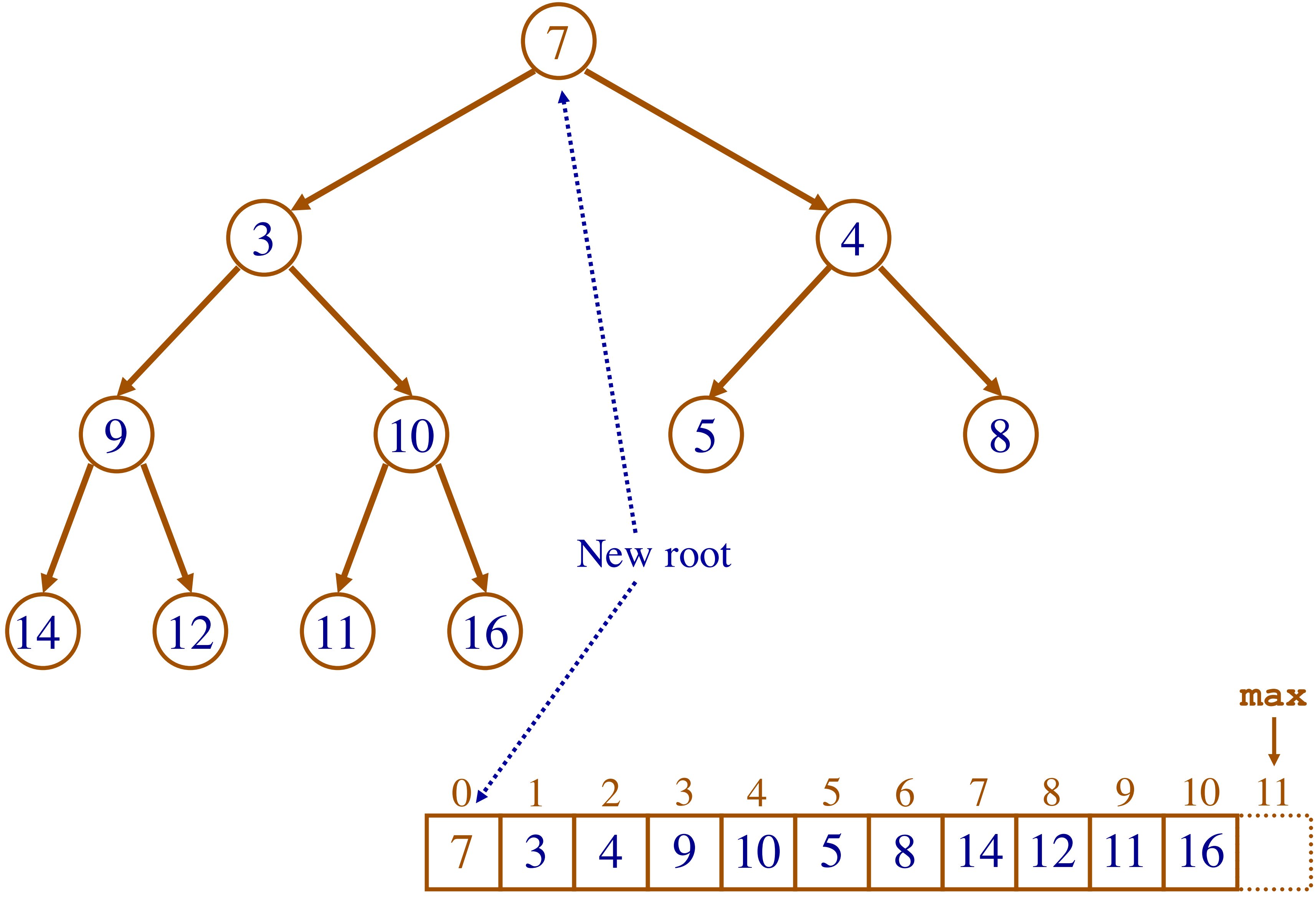


Heap Implementation: removeMin (cont.)

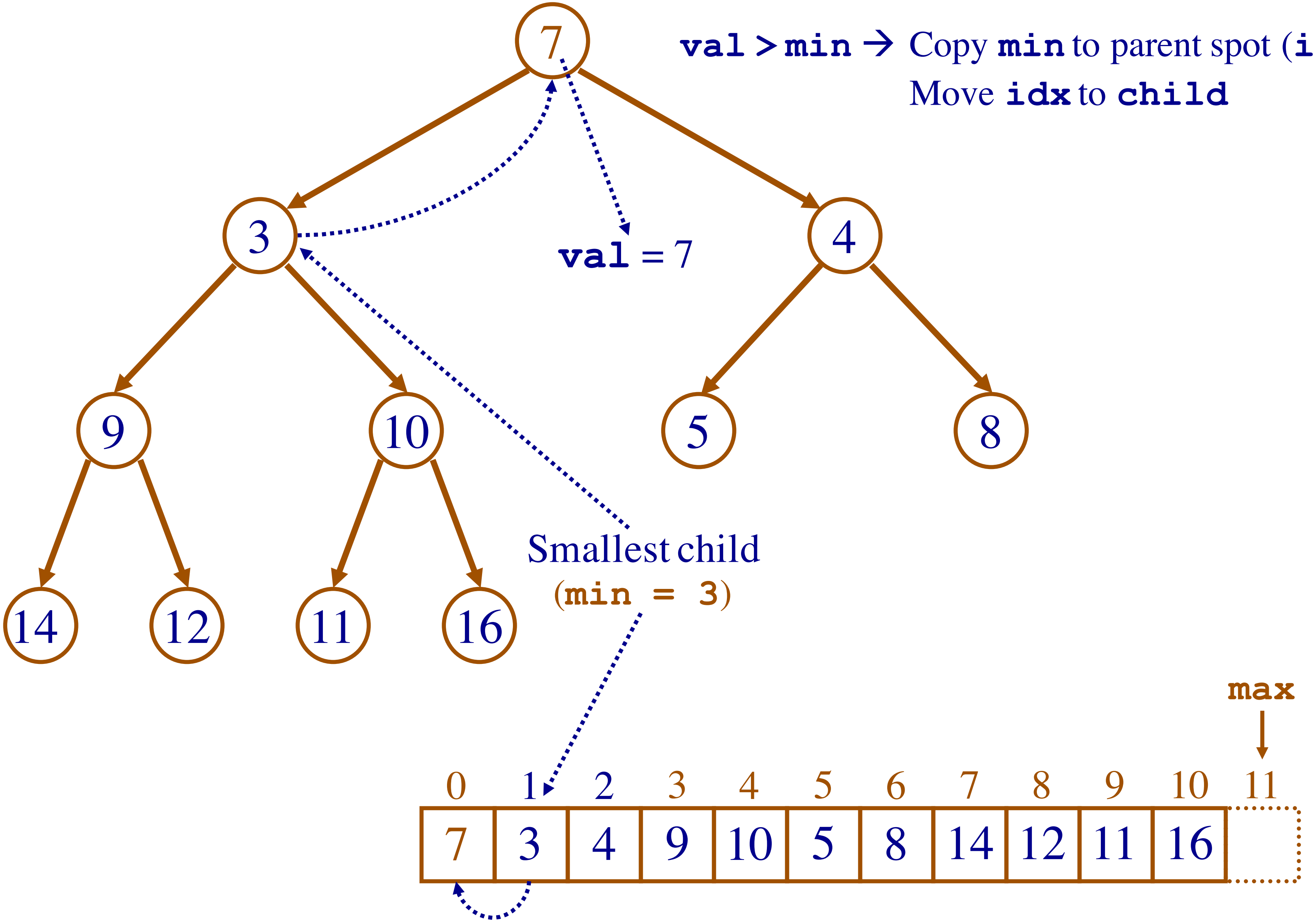


0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	9	10	5	8	14	12	11	16	~~7~~

Heap Implementation: removeMin (cont.)



Heap Implementation: removeMin (cont.)



`val` > **`min`** → Copy **`min`** to parent spot (**`idx`**)
Move **`idx`** to **`child`**

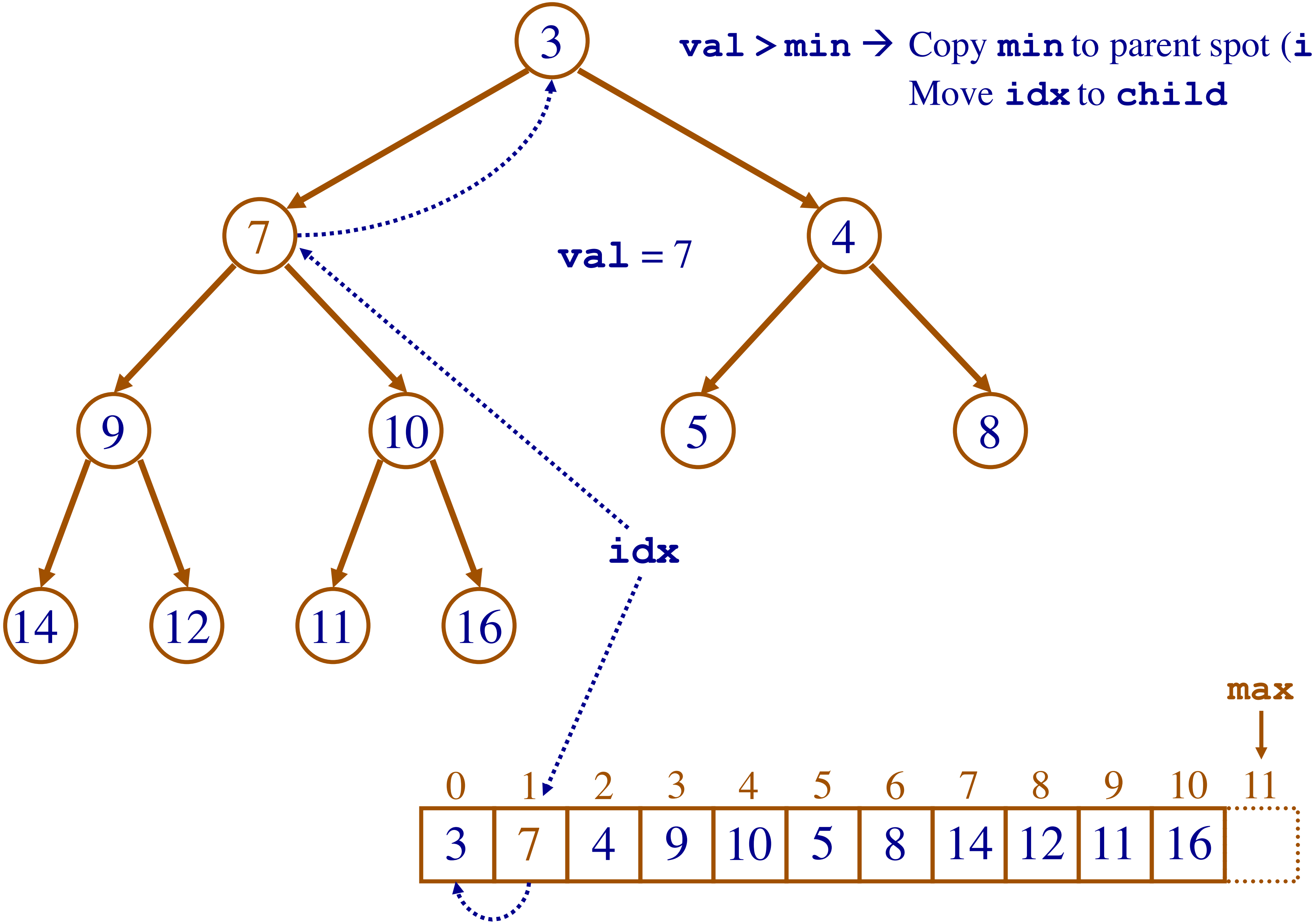
`val` = 7

Smallest child
(**`min = 3`**)

Heap Implementation: removeMin (cont.)

`val` > **`min`** → Copy **`min`** to parent spot (**`idx`**)

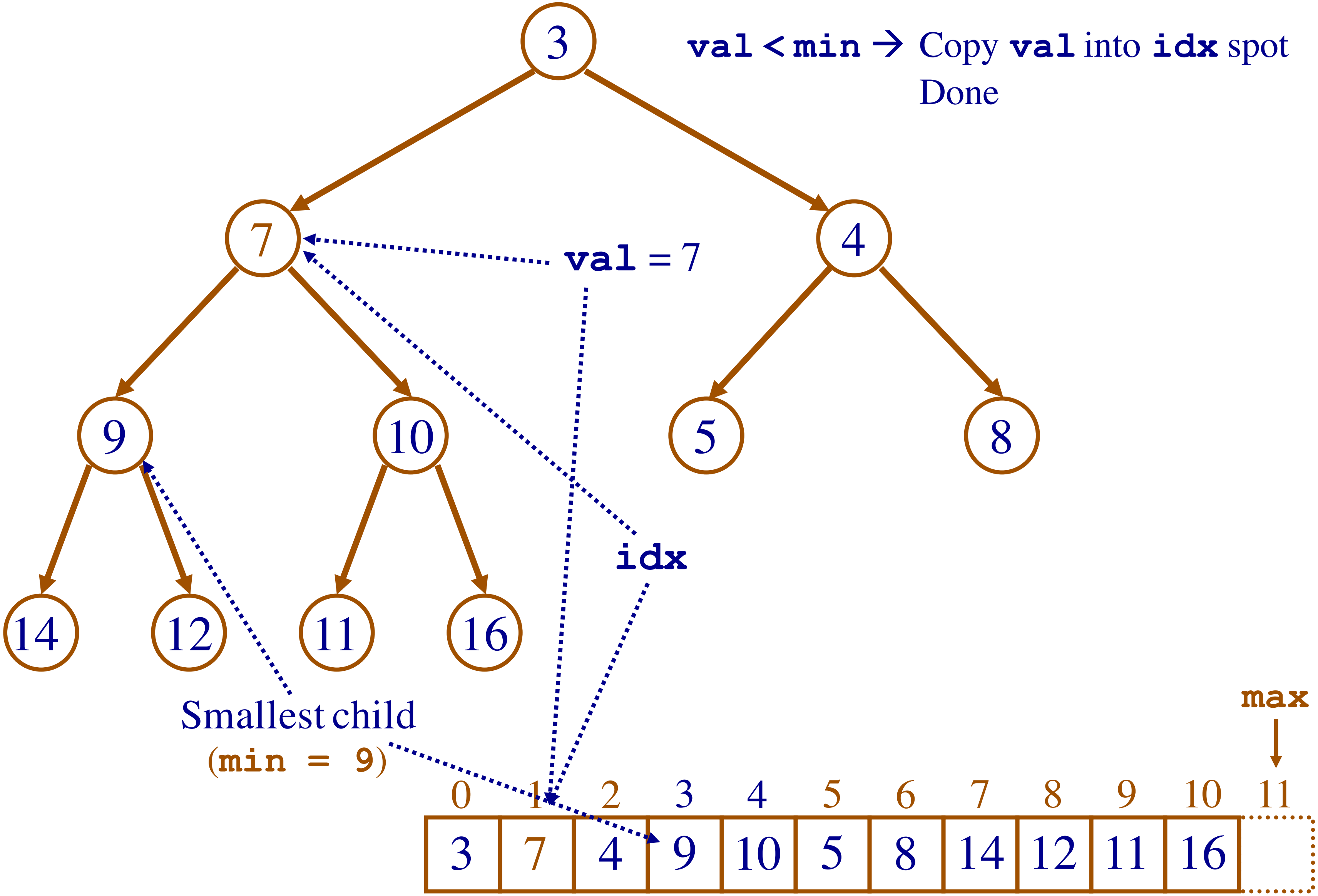
Move **`idx`** to **`child`**



`val` = 7

0	1	2	3	4	5	6	7	8	9	10	11
3	7	4	9	10	5	8	14	12	11	16	

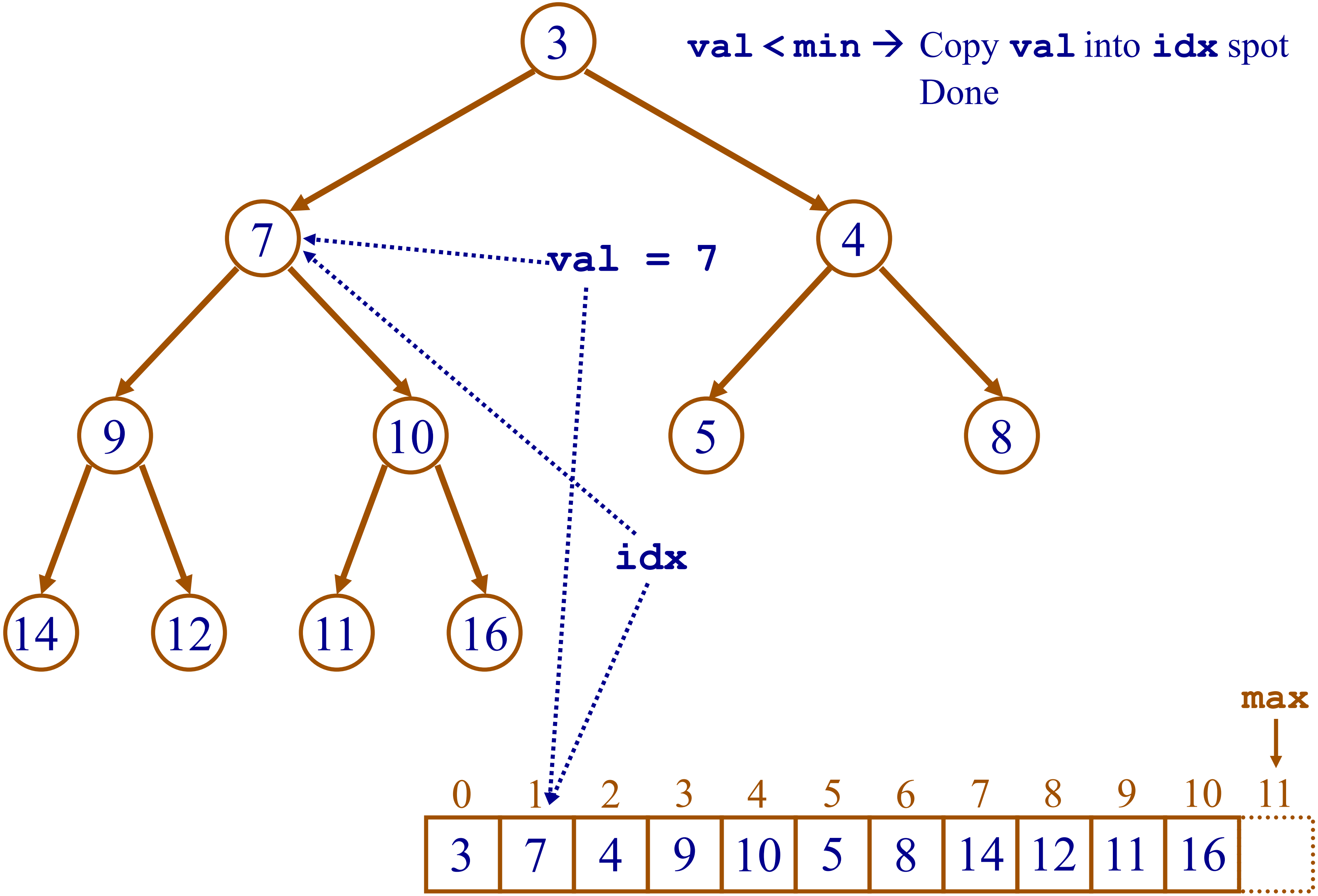
Heap Implementation: removeMin (cont.)



`val` < **`min`** → Copy **`val`** into **`idx`** spot
Done

Heap Implementation: **removeMin (cont.)**

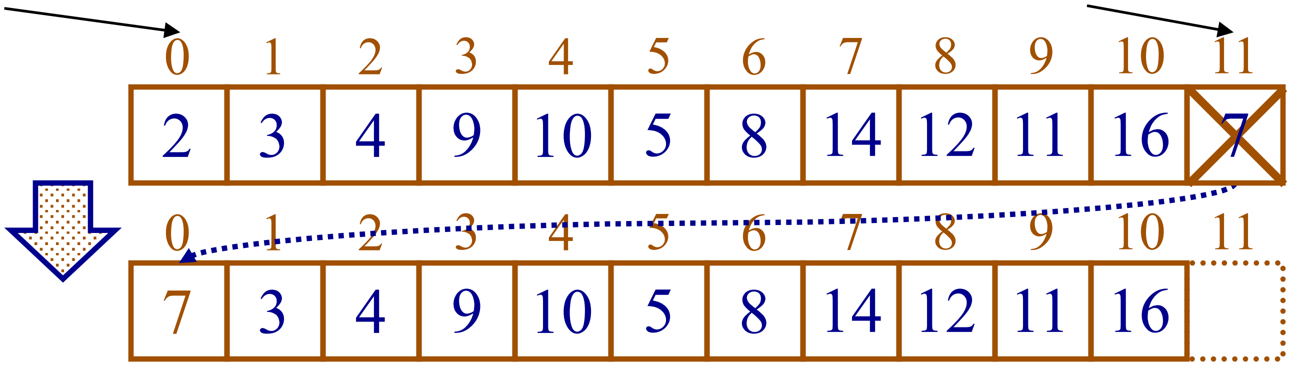
`val` < **`min`** → Copy **`val`** into **`idx`** spot
Done



0	1	2	3	4	5	6	7	8	9	10	11
3	7	4	9	10	5	8	14	12	11	16	

Heap Implementation: removeMin

```
void removeMinHeap(struct DynArr *heap) {
   assert(heap);
   int last = heap->size - 1;
   if (last !=0 ) /* Copy the last element to the first */
      heap->data[0] = heap->data[last];
   heap->size--;  /* Remove last */

   ...
```



First element

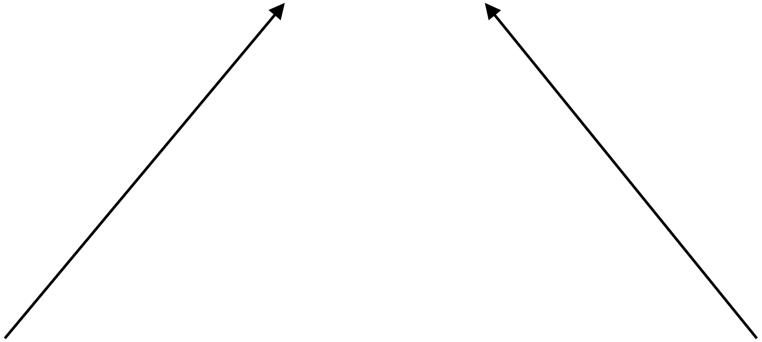
Last element

0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	9	10	5	8	14	12	11	16	~~7~~

0	1	2	3	4	5	6	7	8	9	10	11
7	3	4	9	10	5	8	14	12	11	16	

Heap Implementation: removeMin

```
  void removeMinHeap(struct DynArr *heap) {
    ...
    /* Rebuild heap property */
    _adjustHeap(heap, last-1, 0); /* recursion */
}
```



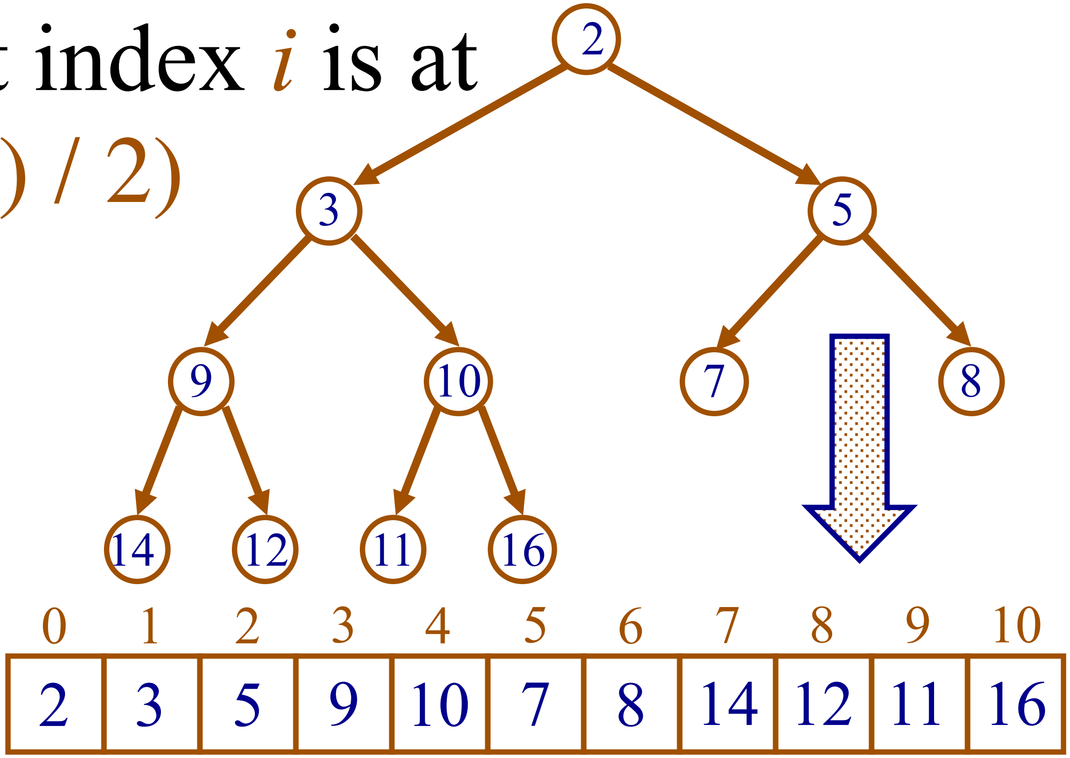
end index

start index

Heap Representation as Dynamic Array

- Children of a node at index *i* are stored at indices $2i + 1$ and $2i + 2$

- Parent of node at index *i* is at index $\text{floor}((i - 1) / 2)$



0	1	2	3	4	5	6	7	8	9	10
2	3	5	9	10	7	8	14	12	11	16

Recursive `_adjustHeap`

```
void _adjustHeap(struct DynArr *heap,
                                int maxIdx, int pos) {
    assert(heap);
    int leftIdx = pos * 2 + 1; /* left child index */
    int rghtIdx = pos * 2 + 2; /* right child index */
    if (rghtIdx < maxIdx){/* there are 2 children */
        /* swap with the smaller child if it is smaller than me */
    }
    else if (leftIdx < maxIdx){/* there is 1 child */
        /* swap if the left child is smaller than me */
    }
    /* else no children, done */
}
```

_adjustHeap

```
void _adjustHeap(struct DynArr *heap,
                              int maxIdx, int pos) {
   ...
   if(rightIdx <= maxIdx){ /* 2 children */
     smallIdx = _minIdx(leftIdx, rightIdx);
     if( LT(heap->data[smallIdx],heap->data[pos]) ){
          _swap(heap->data, pos, smallIdx);
          _adjustHeap(heap, maxIdx, smallIdx);
     }

   }else if(leftIdx <= maxIdx) /* One child */
        ...
```

end index

start index

_adjustHeap

```
void _adjustHeap(struct DynArr *heap,
                            int maxIdx, int pos) {
  ...
  if(rightIdx <= maxIdx){  /* 2 children */
     ...
  }else if(leftIdx <= maxIdx){  /* One child */
    if(LT(heap->data[leftIdx],heap->data[pos])){
      _swap(heap->data, pos, leftIdx);
      _adjustHeap(heap, maxIdx, leftIdx);
    }
  }
}
```

end index

start index

Useful Routines: Swap

```
void _swap(struct DynArr *da, int i, int j)
{
   /* Swap elements at indices i and j */
   TYPE tmp = da->data[i];
   da->data[i] = da->data[j];
   da->data[j] = tmp;
}
```

Useful Routines: minIdx

```
int _minIdx(struct DynArr *da, int i, int j)
{
    /* Return index of the smaller element  */
    if (LT(da->data[i], da->data[j]))
          return i;
    return j;
}
```

Priority Queues: Performance Evaluation

	`SortedVector`	`SortedList`	`Heap`	`SkipList`
`add`	O(n) Binary search Slide data up	O(n) Linear search	O($\log n$) Percolate up	O($\log n$) Add to all lists
`getMin`	O(1) **`get(0)`**	O(1) Returns **`head.val`**	O(1) Get root node	O(1) Get first link
`removeMin`	O(n) Slide data down O(1): Reverse Order	O(1) **`removeFront`**	O($\log n$) Percolate down	O($\log n$) Remove from all lists

Which implementation is the best for priority queue?

Priority Queues: Performance Evaluation

- Remember that a priority queue's main purpose is rapidly accessing and removing the smallest element!

- Consider a case where you will insert (and ultimately remove) n elements:
  - ReverseSortedVector and SortedList:

    Insertions: $n * n = n^2$

    Removals: $n * 1 = n$

    Total time: $n^2 + n = O(n^2)$
  - Heap:

    Insertions: $n * \log n$

    Removals: $n * \log n$

    Total time: $n * \log n + n * \log n = 2n \log n = O(n \log n)$

Priority Queue Application: Simulation

- Original, and one of most important, applications
- Discrete event driven simulation:
  - Actions represented by "events" – things that have (or will) happen at a given time
  - Priority queue maintains list of pending events → Highest priority is next event
  - Event pulled from list, executed → often spawns more events, which are inserted into priority queue
  - Loop until everything happens, or until fixed time is reached

Priority Queue Applications: Example

- Example: Ice cream store
  - People arrive
  - People order
  - People leave
- Simulation algorithm:
  1. Determine time of each event using random number generator with some distribution
  2. Put all events in priority queue based on when it happens
  3. Simulation framework pulls minimum (next to happen) and executes event