CS261 Data Structures

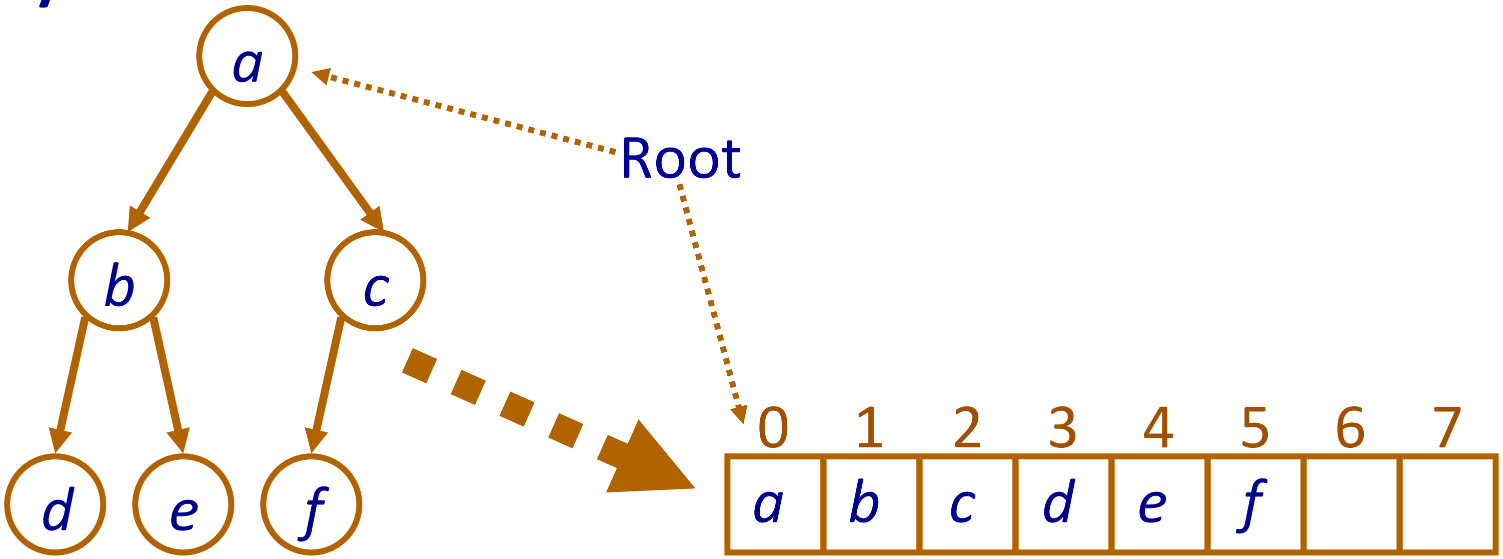
Heap Implementation

Goals

- Heap Representation
- Heap Priority Queue ADT Implementation

Dynamic Array Representation

Complete binary tree has structure that is efficiently implemented with a **DynArr**:



0	1	2	3	4	5	6	7
a	*b*	*c*	*d*	*e*	*f*		

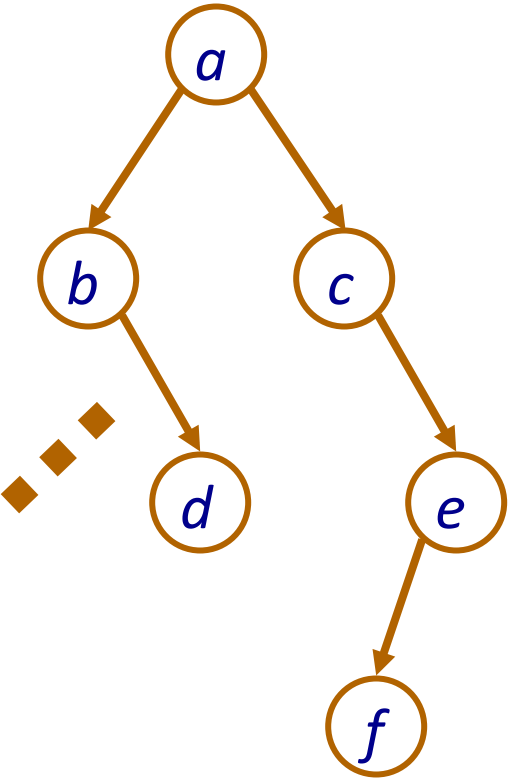
– Children of node *i* are stored at 2*i* + 1 and 2*i* + 2

– Parent of node *i* is at floor((*i* - 1) / 2)

Why is this a bad idea if tree is not complete?

Dynamic Array Implementation (cont.)

If the tree is not complete (it is thin, unbalanced, etc.), the **DynArr** implementation will be full of holes



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	*b*	*c*		*d*		*e*							*f*		

Big gaps where the level is not filled!

Heap Implementation: add

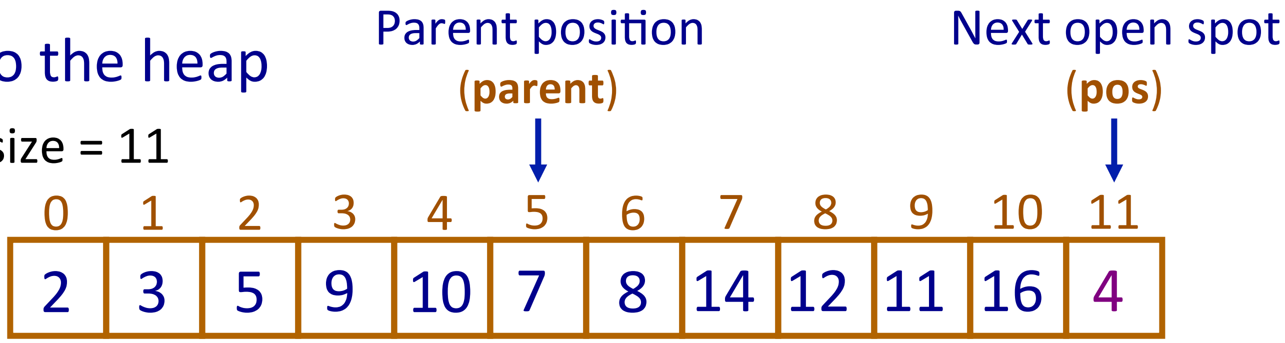
```
void addHeap(struct DynArr *heap, TYPE val) {
  int parent;

  int pos = sizeDynArr(heap);
  addDynArr(heap, val);   /*sets capacity if necessary */

  while(pos != 0){
    parent = (pos-1)/2;
     if(compare(getDynArr(heap, pos), getDynArr(heap, parent)) == -1){
          swapDynArr(heap, parent, pos);
          pos = parent;
     } else return;
  }
}
```

Example: add 4 to the heap

Prior to addition, size = 11



Parent position (parent)

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.)

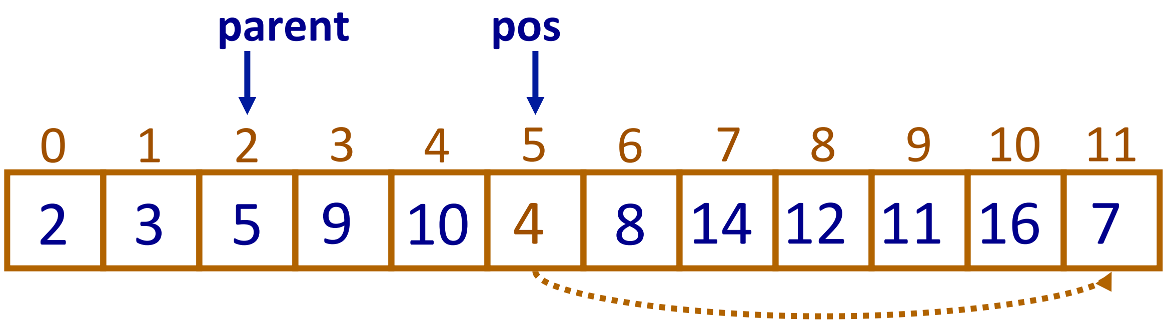
```
void addHeap(struct DynArr *heap, TYPE val) {
    int parent;

    int pos = sizeDynArr(heap);
    addDynArr(heap, val);   /*sets capacity if necessary */

    while(pos != 0){
      parent = (pos-1)/2;
       if(compare(getDynArr(heap, pos), getDynArr(heap, parent)) == -1){
            swapDynArr(heap, parent, pos);
            pos = parent;
       } else return;
    }
}
```

After first iteration: "swapped" new value (4) with parent (7)

New parent value: 5



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.)

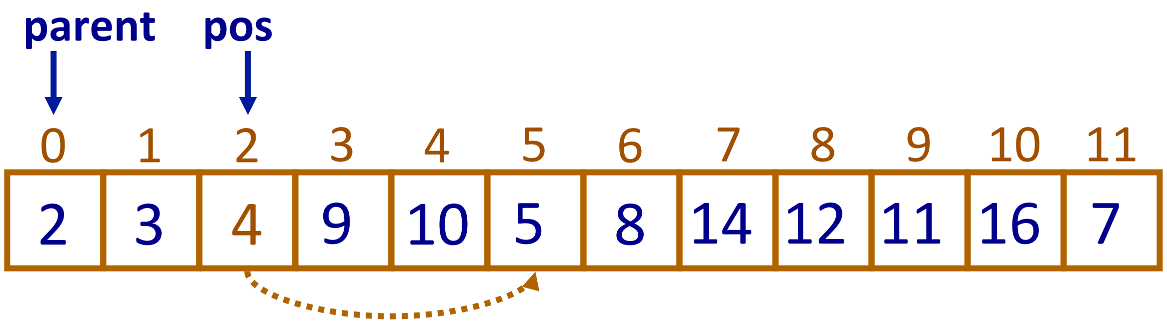
```
void addHeap(struct DynArr *heap, TYPE val) {
    int parent;

    int pos = sizeDynArr(heap);
    addDynArr(heap, val);   /*sets capacity if necessary */

    while(pos != 0){
      parent = (pos-1)/2;
       if(compare(getDynArr(heap, pos), getDynArr(heap, parent)) == -1){
            swapDynArr(heap, parent, pos);
            pos = parent;
       } else return;
    }
}
```

After second iteration: “swapped” new value (4) with parent (5)

New parent value: 2



parent pos

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

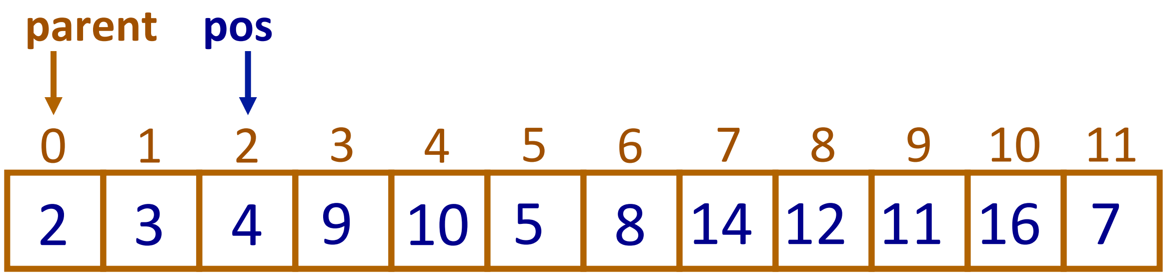
Heap Implementation: add (cont.)

```
void addHeap(struct DynArr *heap, TYPE val) {
    int parent;

    int pos = sizeDynArr(heap);
    addDynArr(heap, val);   /*sets capacity if necessary */

    while(pos != 0){
      parent = (pos-1)/2;
       if(compare(getDynArr(heap, pos), getDynArr(heap, parent)) == -1){
            swapDynArr(heap, parent, pos);
            pos = parent;
       } else return;
    }
}
```

If test fails: returns from iteration

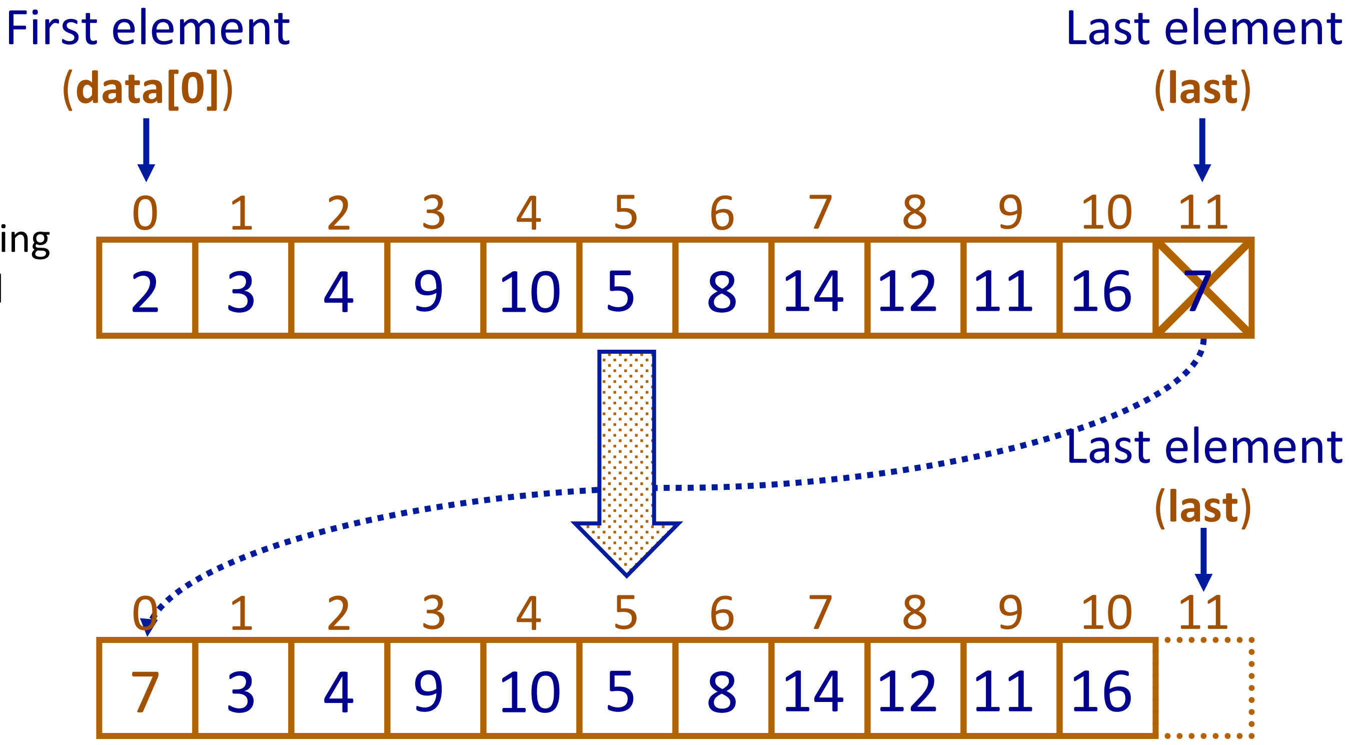


0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	9	10	5	8	14	12	11	16	7

Heap Implementation: removeMin

```
void removeMinHeap(DynArr *heap){
     int last;
     assert(sizeDynArr(heap) > 0);
     last = sizeDynArr(heap) – 1;
     putDynArr(heap, 0, getDynArr(heap, last));   /* Copy the last element to the first */
     removeAtDynArr(heap, last);                  /* Remove last element. */
     _adjustHeap(heap, last , 0);                 /* Rebuild heap */
}
```

Percolates down from Index 0 to last (not including last…which is one beyond the end now!)



First element (data[0])

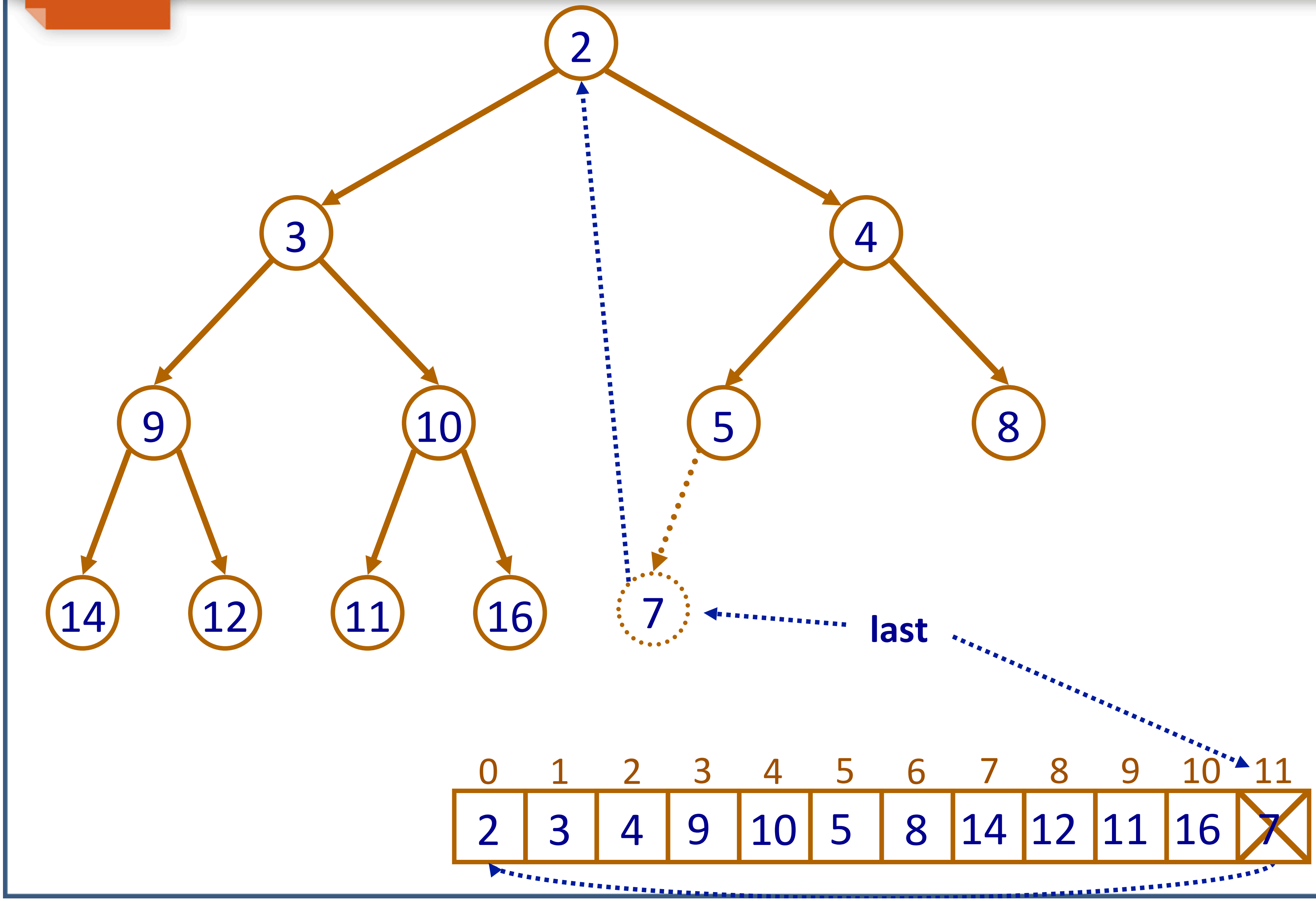
Last element (last)

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

Last element (last)

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

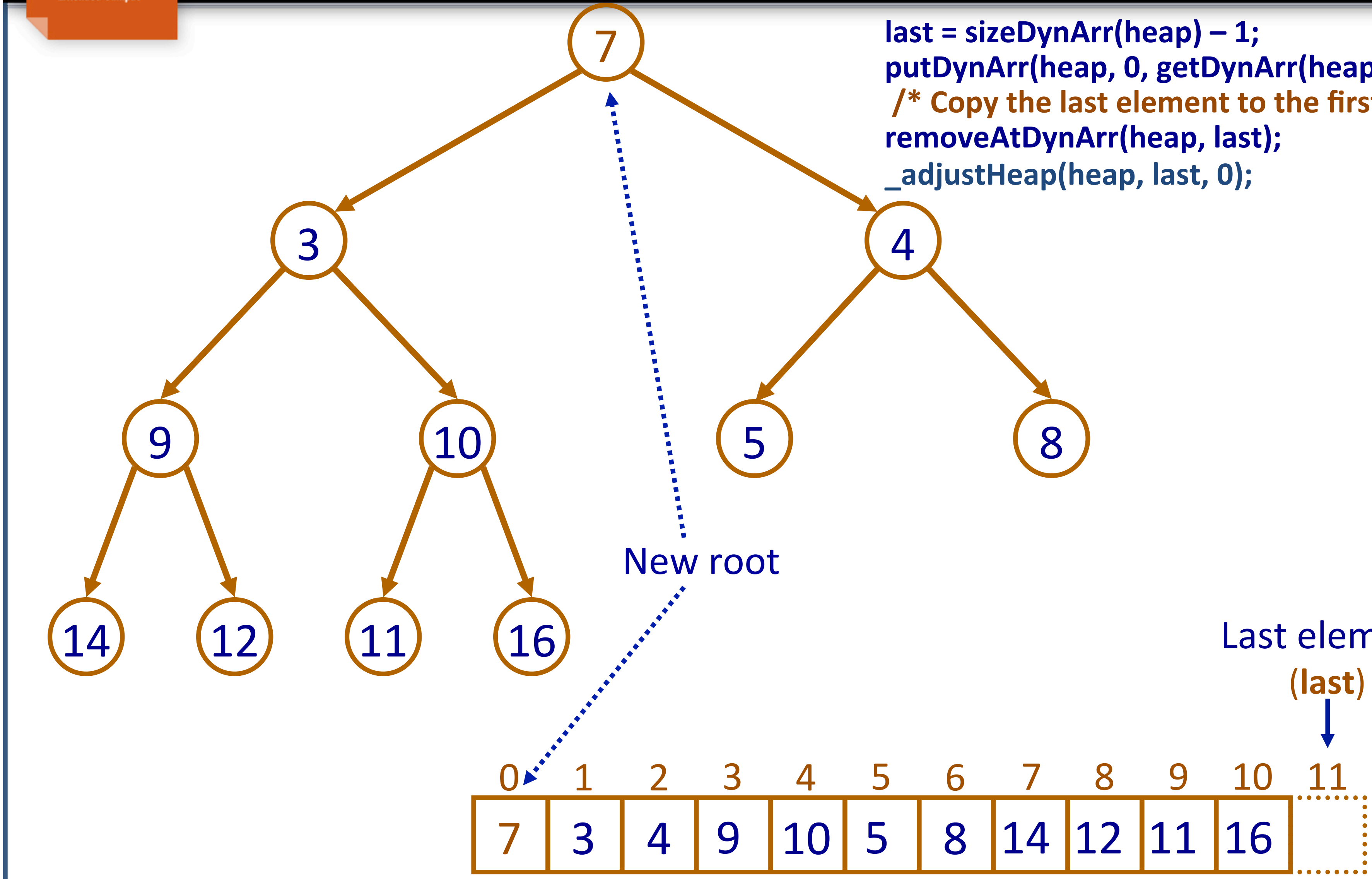
Heap Implementation: removeMin



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.)

```
last = sizeDynArr(heap) – 1;
putDynArr(heap, 0, getDynArr(heap, last));
 /* Copy the last element to the first */
removeAtDynArr(heap, last);
_adjustHeap(heap, last, 0);
```

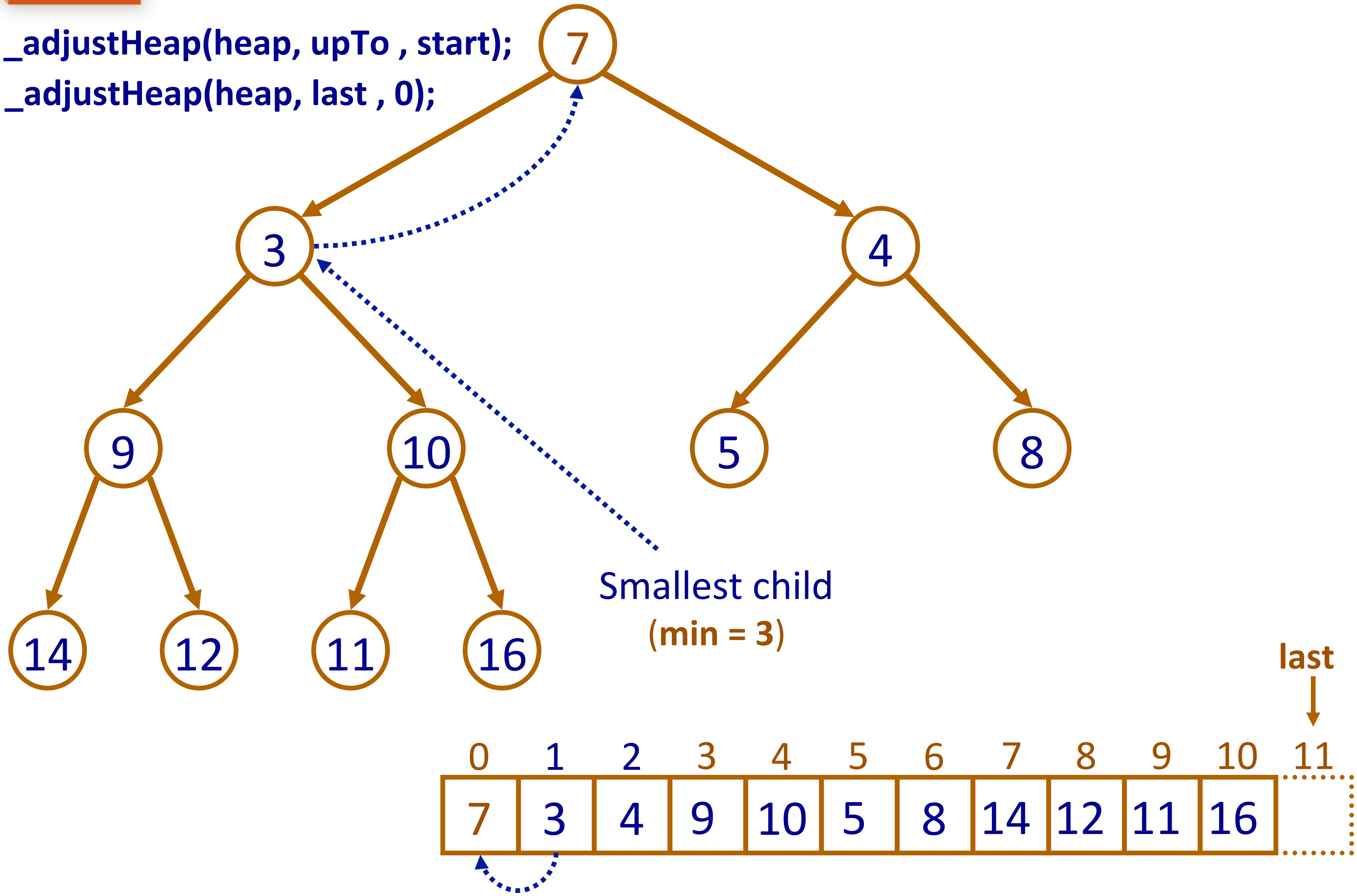


0	1	2	3	4	5	6	7	8	9	10	11
7	3	4	9	10	5	8	14	12	11	16	

Heap Implementation: _adjustHeap

_adjustHeap(heap, upTo , start);

_adjustHeap(heap, last , 0);



Smallest child

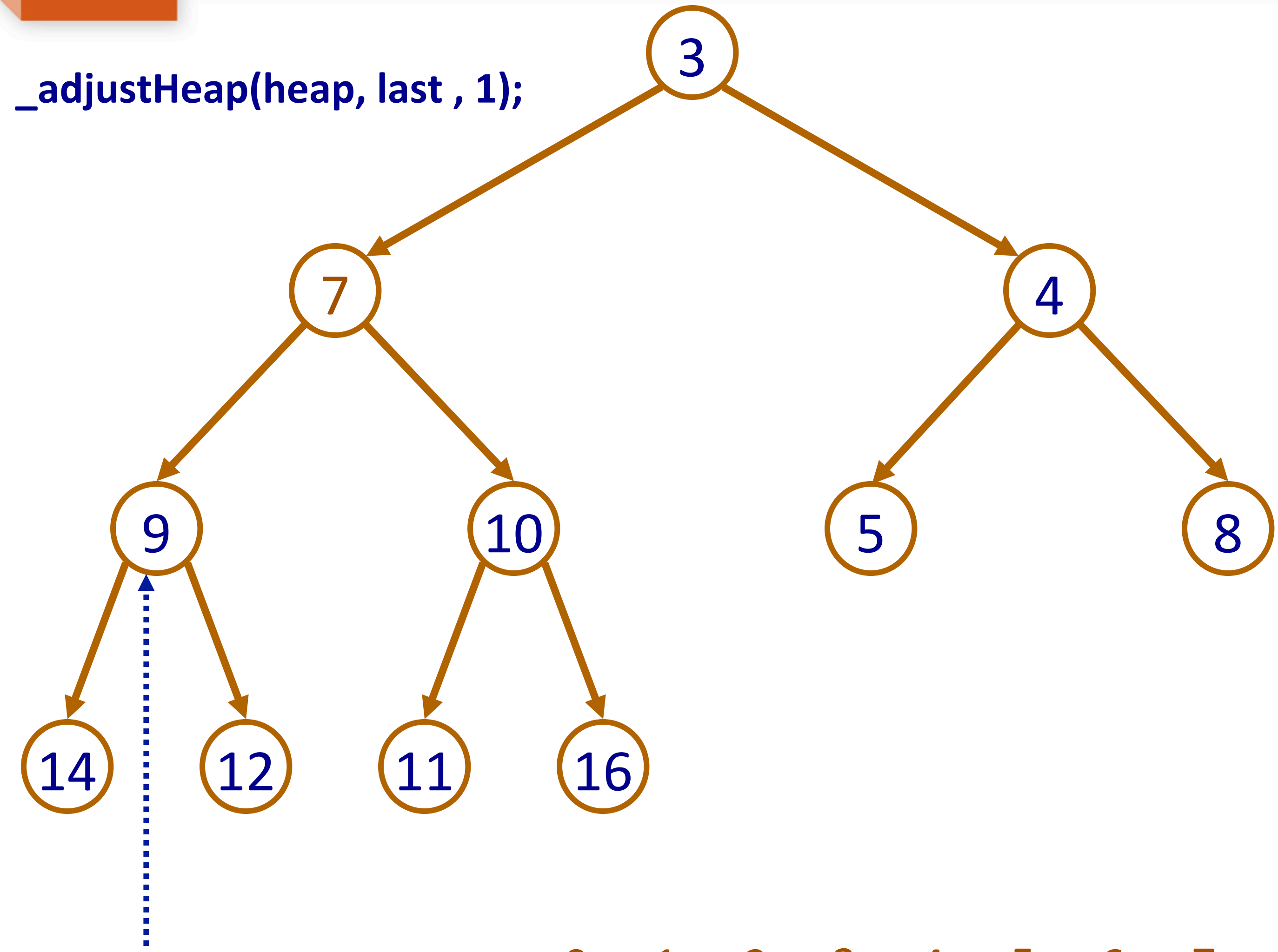
(min = 3)

last

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: _adjustHeap

```
_adjustHeap(heap, last , 1);
```

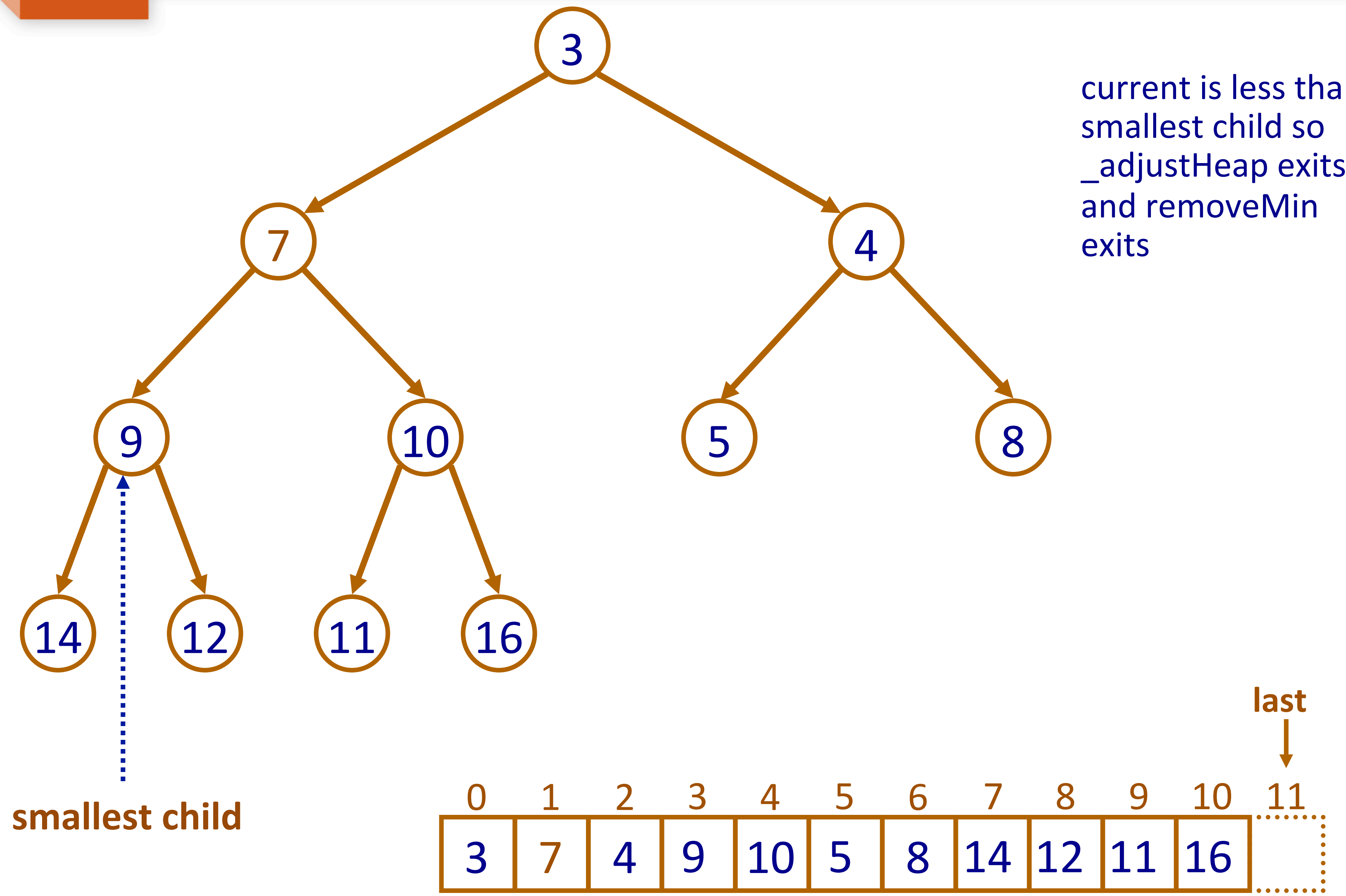


smallest child

last

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: _adjustHeap



current is less than smallest child so _adjustHeap exits and removeMin exits

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

Recursive _adjustHeap

```
void _adjustHeap(struct DynArr *heap, int max, int pos) {
 int leftIdx = pos * 2 + 1;
 int rghtIdx = pos * 2 + 2;

 if (rghtIdx < max) {
    /* Have two children? */
    /* Get index of smallest child (_minIdx). */
    /* Compare smallest child to pos. */
    /* If necessary, swap and call _adjustHeap(max, minIdx). */
 }
 else if (leftIdx < max) {
    /* Have only one child. */
    /* Compare child to parent. */
     /* If necessary, swap and call _adjustHeap(max, leftIdx). */
 }
    /* Else no children, we are at bottom → done. */
}
```

Useful Routines

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

int minIdx(struct DynArr *arr, int i, int j) {
  /* Return index of smallest element value. */
  if (compare(arr->data[i], arr->data[j]) == -1)
    return i;
  return j;
}
```

Priority Queues: Performance Evaluation

	`SortedVector`	`SortedList`	`Heap`
`add`	O(*n*) Binary search Slide data up	O(*n*) Linear search	O(log *n*) Percolate up

`getMin`	O(1)
	get(0)
		O(1)
		Returns **firstLink val**
			O(1)
			Get root node

`removeMin`	O(*n*)
	Slide data down
	O(1): Reverse Order
		O(1)
		removeFront()
			O(log *n*)
			Percolate down

So, which is the best implementation of a priority queue?

Priority Queues: Performance Evaluation

- Recall 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: $n * \log n + n * \log n = 2n \log n = O(n \log n)$

Your Turn

- Complete Worksheet #33