CS 261 – Data Structures

BuildHeap and Heap Sort

HW6

```
#include <stdio.h>
FILE *filePtr;
char filename[100];

filePtr = fopen(filename, "w");

if (filePtr == NULL)
   printf("Cannot open %s\n", filename);


fprintf(filePtr, "%d\t%s\n", task.priority,
                             task.description);


fclose(filePtr);
```

HW6

```
#include <stdio.h>
FILE *filePtr;
char filename[100];
int priority;

filePtr = fopen(filename, "r");

if (filePointer == NULL)
   printf("Cannot open %s\n", filename);

while(fscanf(filePtr,"%d\t",&priority) != EOF)
{ ... }

fclose(filePtr);
```

HW6

```
#include <stdio.h>
FILE *filePtr;
char filename[100];
char desc[TASK_DESC_SIZE];

....
while(fscanf(filePtr,"%d\t",&priority)  != EOF)
{
   ...

   fgets(desc, sizeof(desc), filePtr);
}

fclose(filePtr);
```

Heap Implementation: Constructors

Given an array of data,

construct the heap

Heap Implementation: Constructors

```
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i =                    )




}
```

Heap Implementation: Constructors

```
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
       /* Make the heap from the subtree rooted at i */
       _adjustHeap(da, maxIdx, i);
  }
```

Heap Implementation: Constructors

```
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
       /* Make the heap from the subtree rooted at i */
       _adjustHeap(da, maxIdx, i);
  }
```

Why?

Heap Implementation: Constructors

```
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
        /* Make the heap from the subtree rooted at i */
        _adjustHeap(da, maxIdx, i);
}
```

Why?

At the beginning, only the leaves are proper heaps:

Leaves are all nodes with indices > maxIdx / 2

Heap Implementation: Constructors

```
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
       /* Make the heap from the subtree rooted at i */
       _adjustHeap(da, maxIdx, i);
  }
```

At each step, the subtree rooted at *i* becomes a heap

Heap Implementation: Build heap

```
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
      _adjustHeap(da, maxIdx, i); /* Make subtree rooted at i a heap */
  }
```

- For all subtrees that are are not already heaps:
  - Call `_adjustHeap` with the *largest* node index that is not already guaranteed to be a heap
  - Iterate until the root node becomes a heap

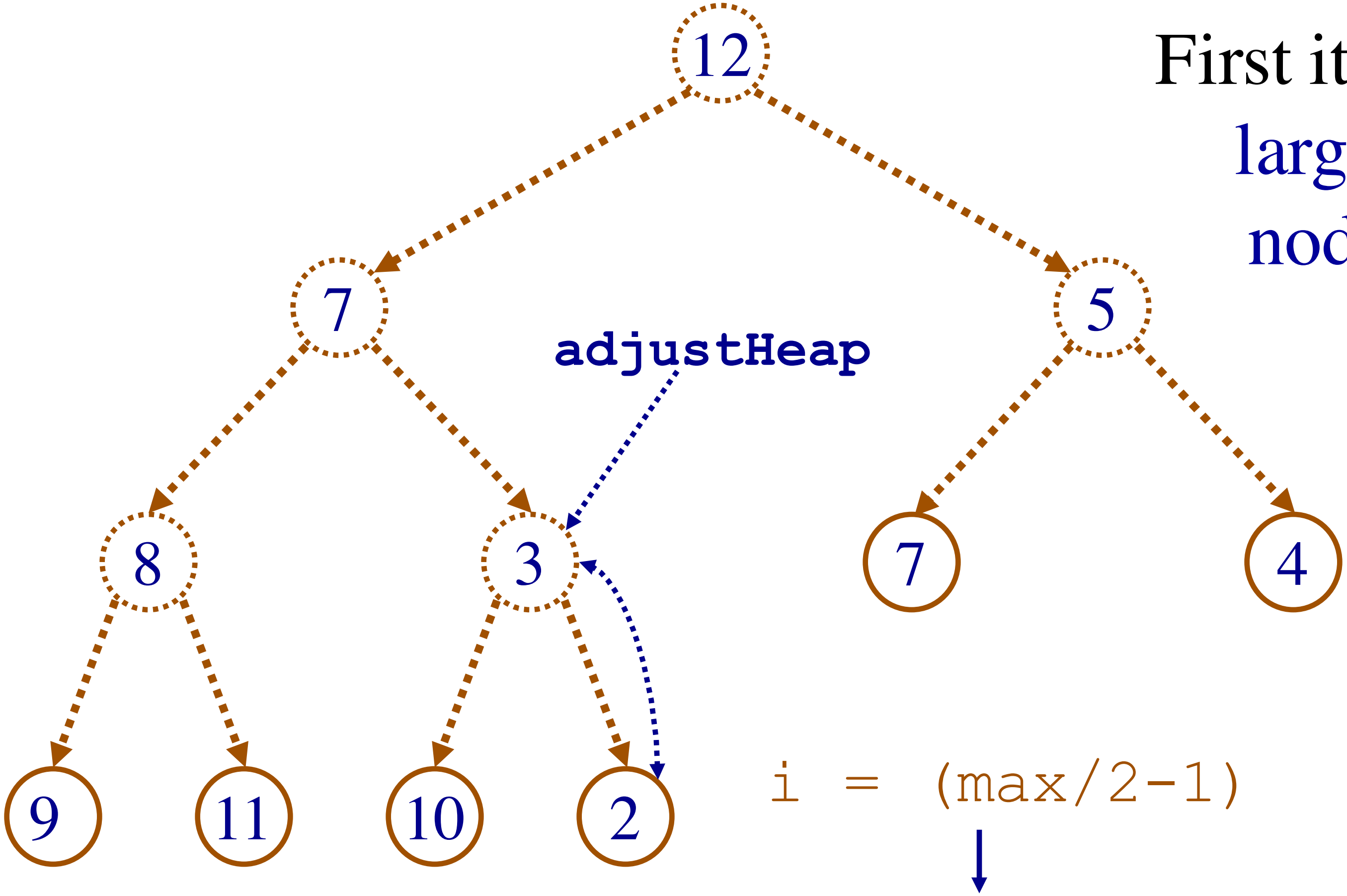
Heap Implementation: Build heap

```
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
      _adjustHeap(da, maxIdx, i); /* Make subtree rooted at i a heap */
  }
```

- Why call `_adjustHeap` with the *largest* node index ?
  - –Because its children, having larger indices, are already guaranteed to be heaps

Heap Implementation: _adjustHeap

First iteration: adjust largest non-leaf node (index 4)



i = (max/2-1)

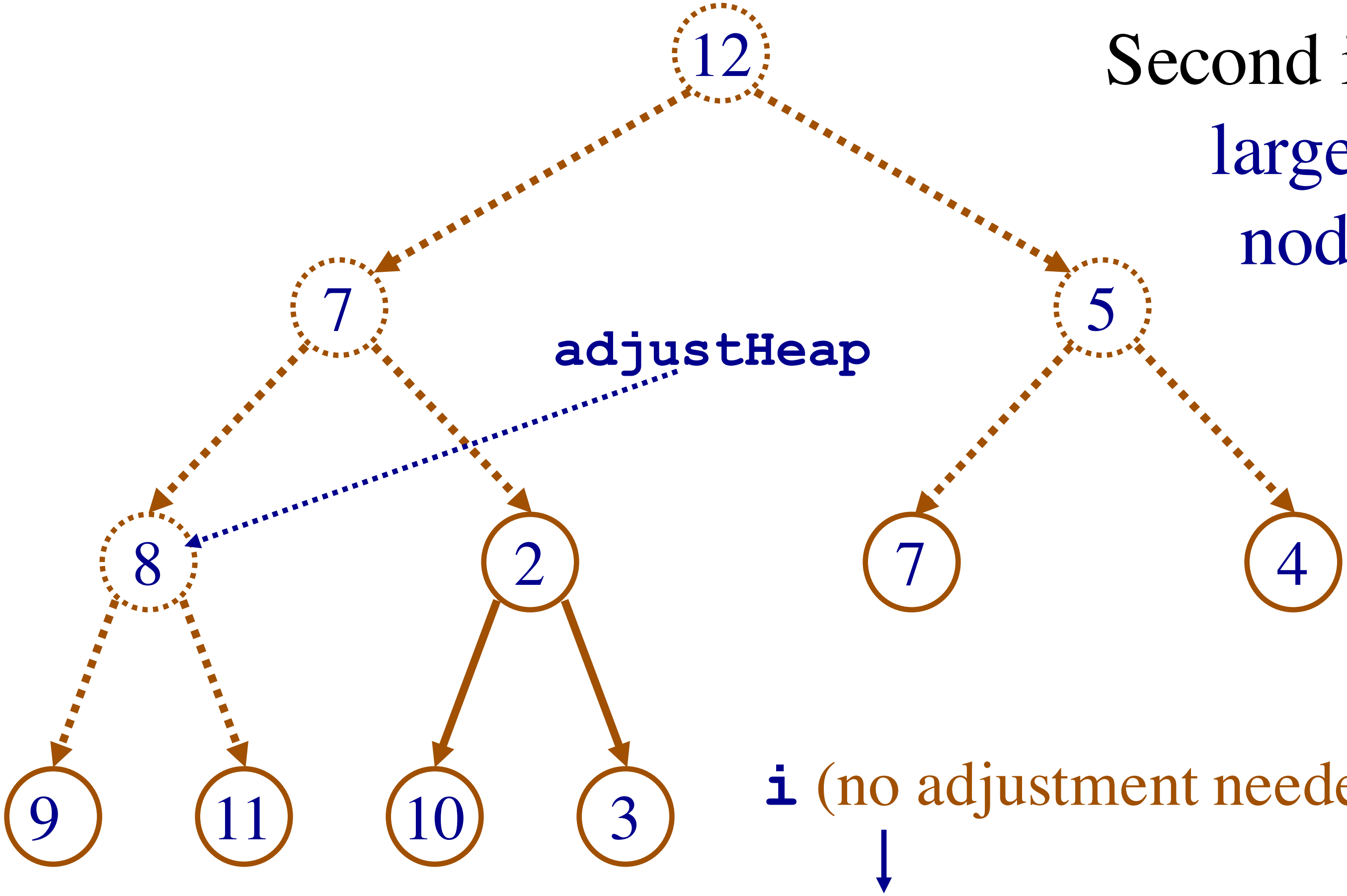
max

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

Already heaps (leaf nodes)

Heap Implementation: adjustHeap (cont.)

Second iteration: adjust largest non-heap node (index 3)



i (no adjustment needed)

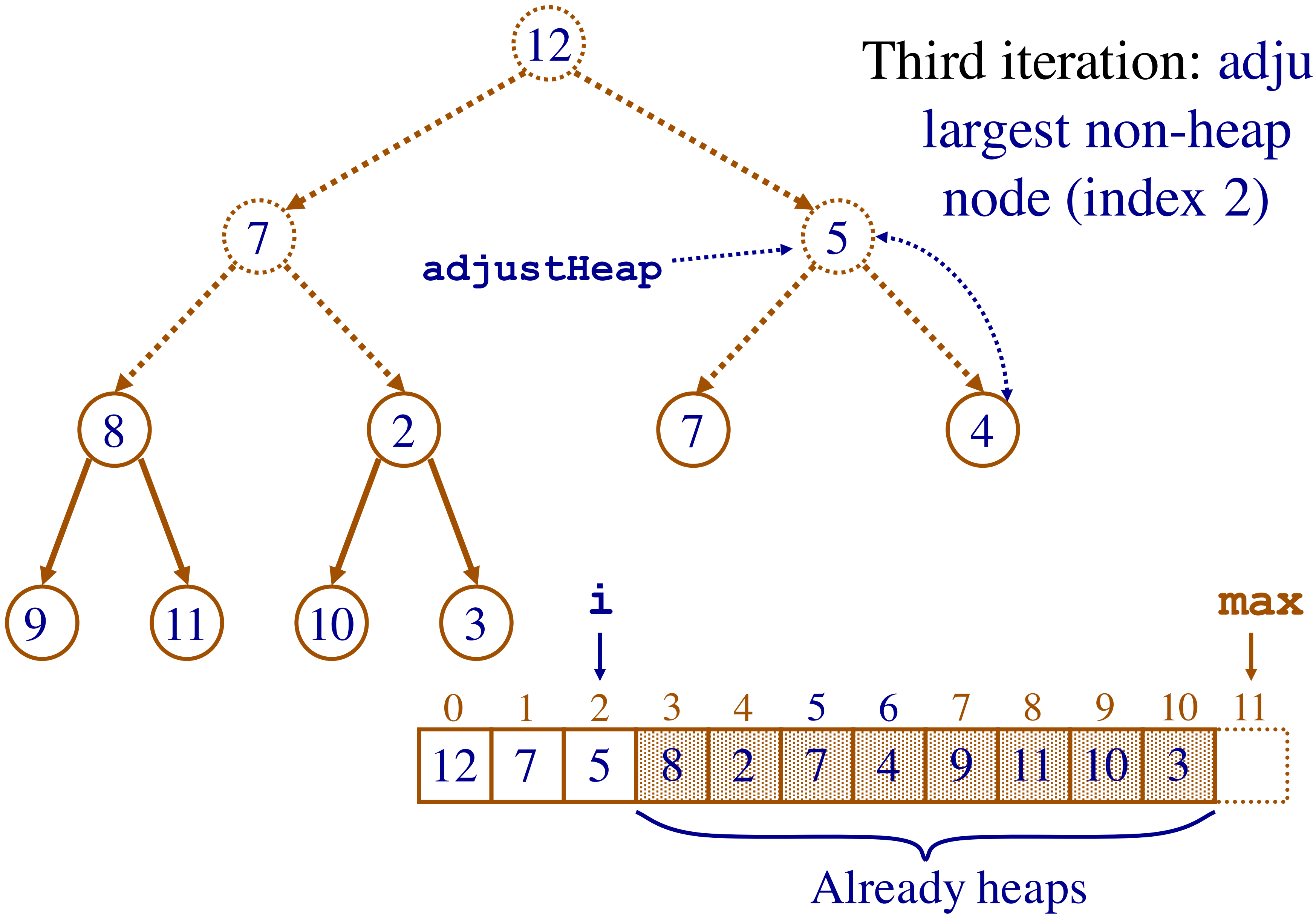
max

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

Already heaps

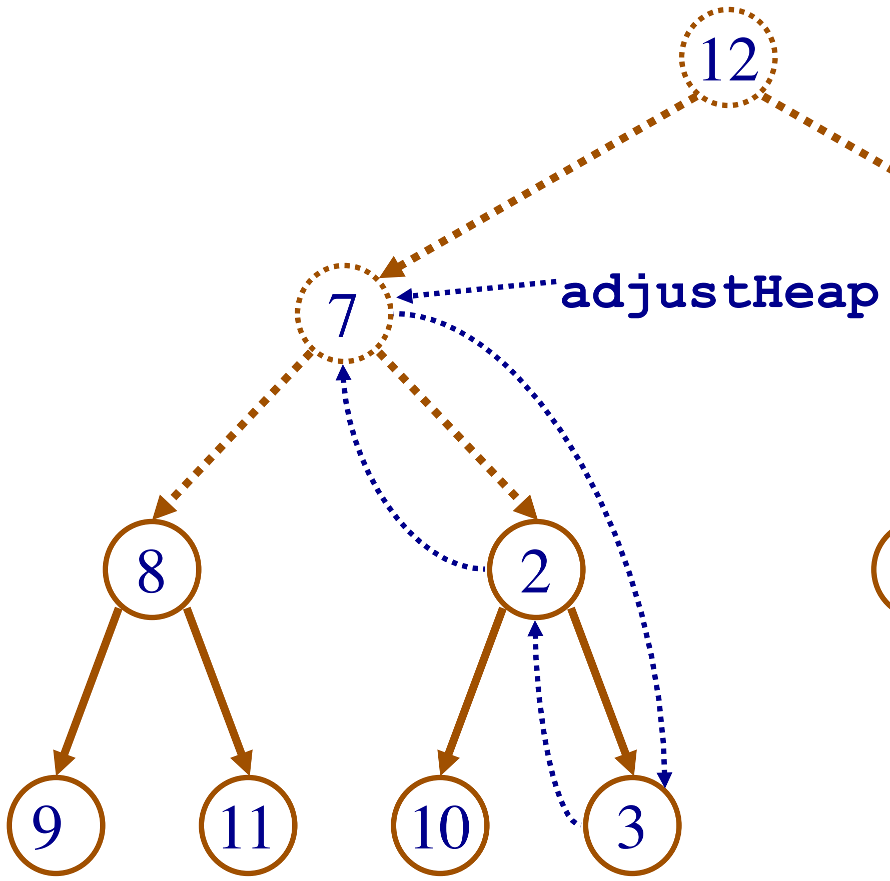
Heap Implementation: _adjustHeap (cont.)

Third iteration: adjust largest non-heap node (index 2)



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

Heap Implementation: adjustHeap (cont.)



Fourth iteration: adjust largest non-heap node (index 1)

max

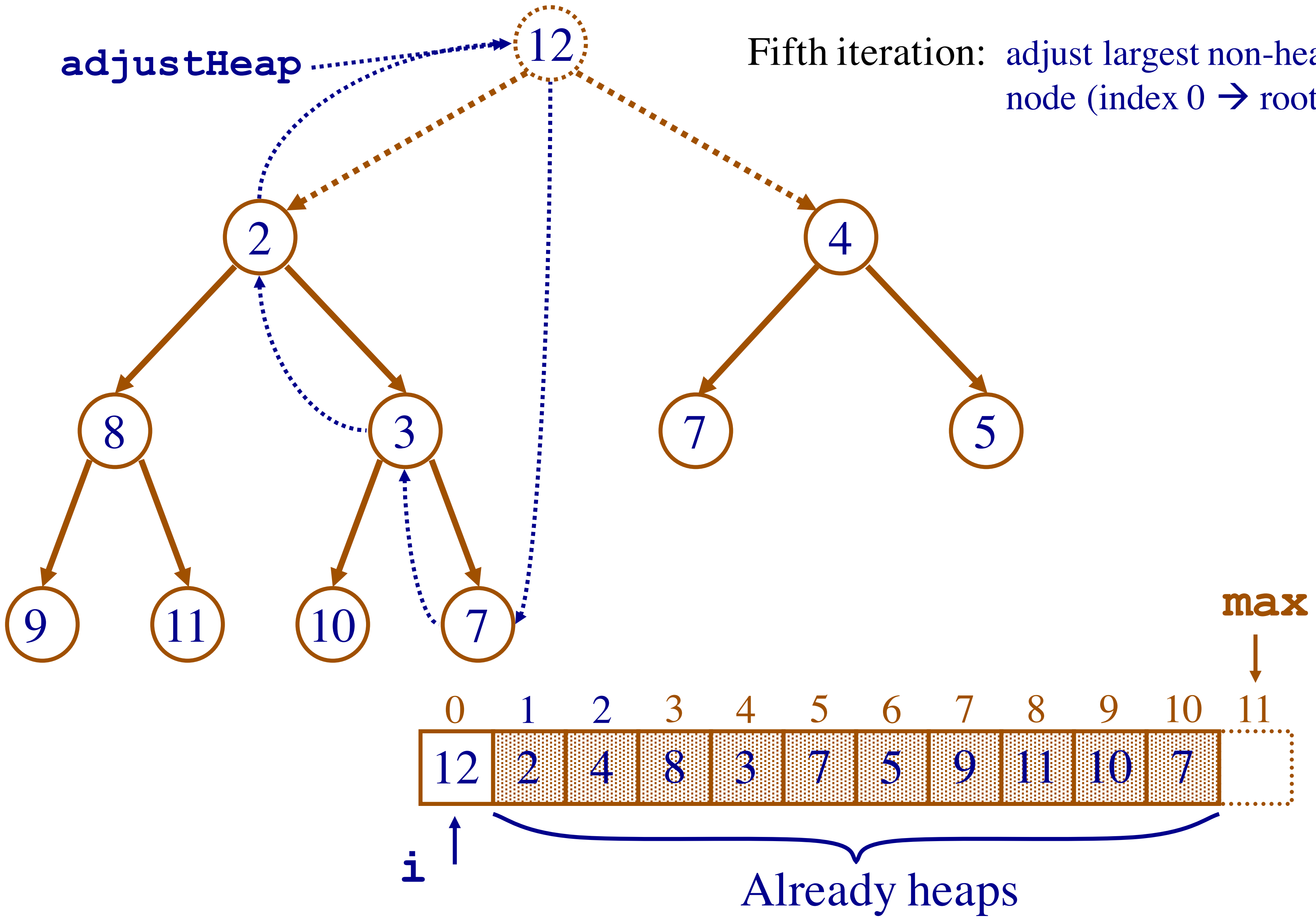
0	1	2	3	4	5	6	7	8	9	10	11
12	7	4	8	2	7	5	9	11	10	3	

i

Already heaps

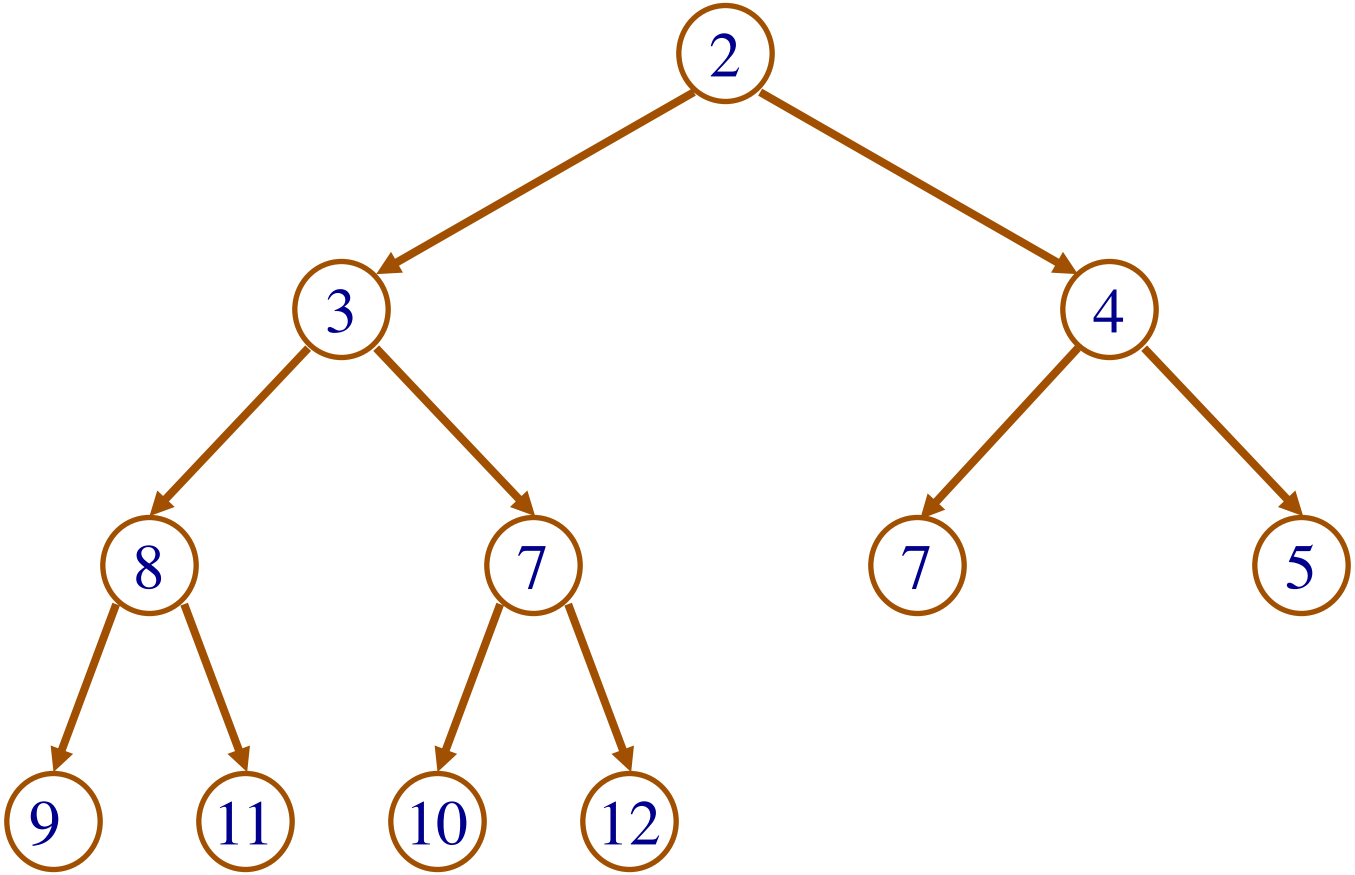
Heap Implementation: adjustHeap (cont.)

Fifth iteration: adjust largest non-heap node (index 0 → root)



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

Heap Implementation: adjustHeap (cont.)



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

Already heaps (entire tree)

Heap Implementation: Sort Descending

Sorts the data in descending order:

1. Builds heap from initial (unsorted) data
2. Iteratively swaps the smallest element (at index 0) with last *unsorted* element
3. Adjust the heap after each swap, but only considers the *unsorted* data

Heap Implementation: Sort Descending

```
void heapSort(struct dyArray * data) {
   int i;
   buildHeap(data);
   for (i = sizeDynArr(data)-1; i > 0; i--){
      swapDynArr(data,i,0);/*Swap  last el. with the first*/
      _adjustHeap(data,i,0);  /* build heap property*/
   }
}
```

Heap Analysis: Sort

- Execution time:
  - Build heap:
    - n calls to **`adjustHeap`** = $n \log n$
  - Loop:
    - n calls to **`adjustHeap`** = $n \log n$
  - Total:
    - $2n \log n = O(n \log n)$

Heap Analysis: Sort

- Advantages/disadvantages:
  - Same average as merge sort and quick sort
  - Doesn’t require extra space as the merge sort does
  - Doesn’t suffer if data is already sorted or mostly sorted