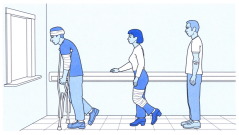



Estruturas de Dados / Programação 2
Heaps

Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

Priority Queues

- We implemented using a naive approach…
  - Enqueue: O(n)
  - Dequeue: O(1)



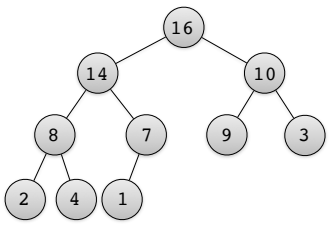
```
enqueue(pq, "a", 17);   [<a,17>]
enqueue(pq, "b", 12);   [<a,17>,<b,12>]
enqueue(pq, "c", 100);  [<c,100>,<a,17>,<b,12>]
enqueue(pq, "d", 22);   [<c,100>,<d,22>,<a,17>,<b,12>]
dequeue(pq);            [<d,22>,<a,17>,<b,12>]
dequeue(pq);            [<a,17>,<b,12>]
```

We can do much better!
O(log n)

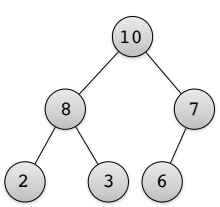
Heap

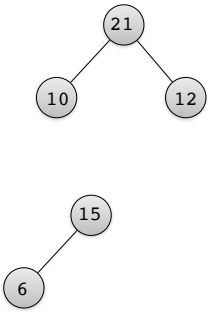
Definition

- For every node "i" other than the root, the value of a node is at most the value of its parent
- Completely filled on all levels except possibly the lowest, which is filled from left to right

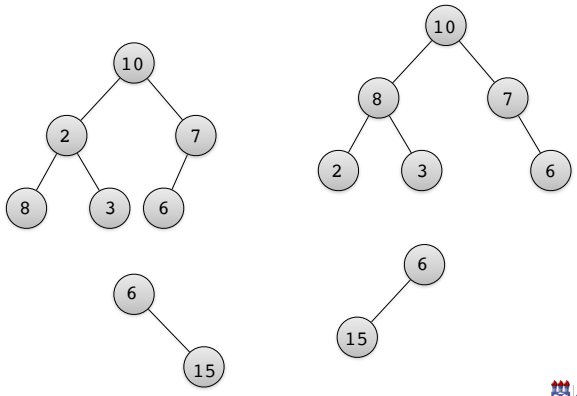


Heap examples

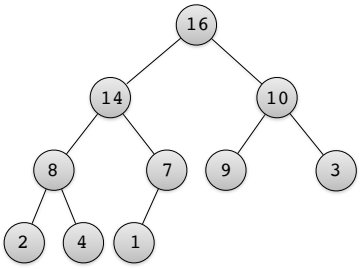




What about these ones?!



Array representation



Parent(i) = i/2 = i >> 1

Left(i) = 2i = i << 1

Right(i) = 2i + 1 = (i << 1) + 1

A[1 .. A.length]

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

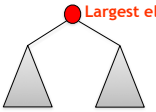
Valid heap elements: A[1 .. A.heap-size]

Max-Heaps and Min-Heaps

Max-heaps

A[Parent(i)] ≥ A[i]

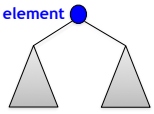
Largest element



Min-heaps

A[Parent(i)] ≤ A[i]

Smallest element



Abstract Data Type: Heap

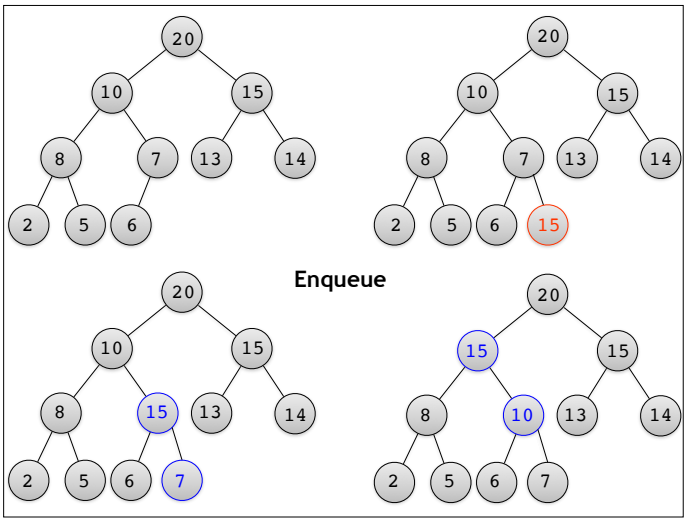
Heap ADT

```
heap* create_heap();

void enqueue(heap *heap, int item);

int dequeue(heap *heap);

int get_parent_index(heap *heap, int i);

int get_left_index(heap *heap, int i);

int get_right_index(heap *heap, int i);

void max_heapify(heap *heap, int i);

int item_of(heap *heap, int i);

void heapsort(heap *heap);
```

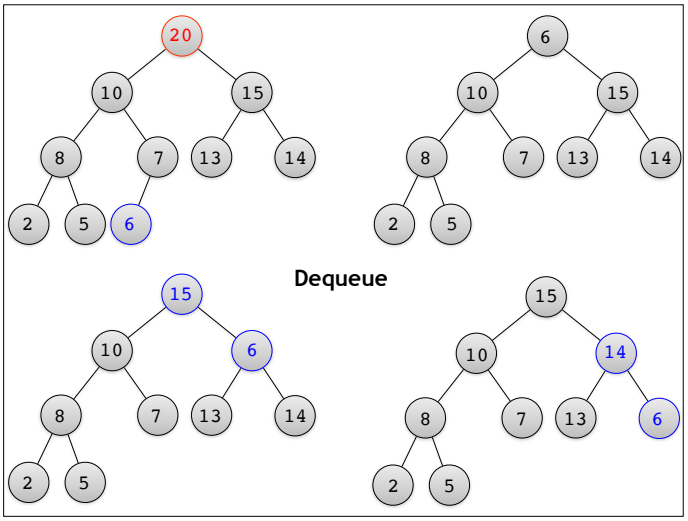
Struct

```
struct heap {
   int size;
   int data[MAX_HEAP_SIZE];
};
```

```
int get_parent_index(heap *heap, int i)
{
  return i/2;
}

int get_left_index(heap *heap, int i)
{
  return 2*i;
}

int get_right_index(heap *heap, int i)
{
  return 2*i + 1;
}

int item_of(heap *heap, int i)
{
  return heap->data[i];
}
```



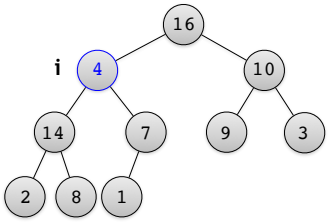

```
void enqueue(heap *heap, int item)
{
  if (heap->size >= MAX_HEAP_SIZE) {

    printf("Heap overflow");

  } else {
    heap->data[++heap->size] = item;

    int key_index = heap->size;
    int parent_index = get_parent_index(heap, heap->size);

    while (parent_index >= 1 &&
           heap->data[key_index] > heap->data[parent_index]) {

      swap(&heap->data[key_index], &heap->data[parent_index]);

      key_index = parent_index;

      parent_index = get_parent_index(heap, key_index);
    }
  }
}
```



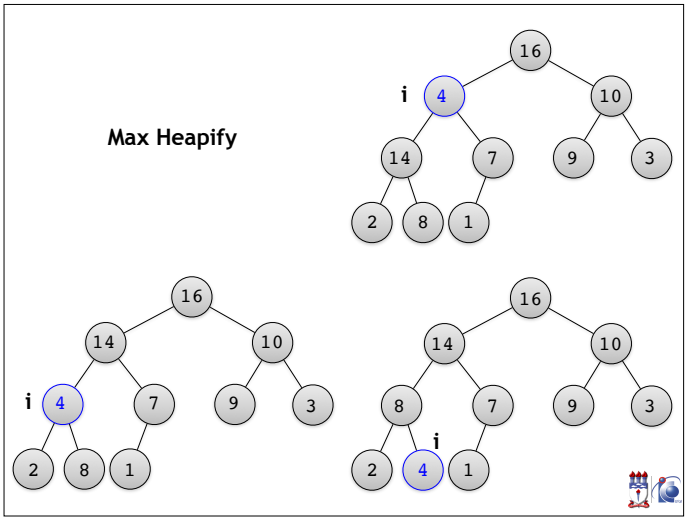

```
int dequeue(heap *heap)
{
  if (is_empty(heap)) {

    printf("Heap underflow");
    return -1;

  } else {

    int item = heap->data[1];

    heap->data[1] = heap->data[heap->size];

    heap->size--;

    max_heapify(heap, 1);

    return item;
  }
}
```

Maintaining the Heap property

- MAX-Heapify
- When called, it assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps

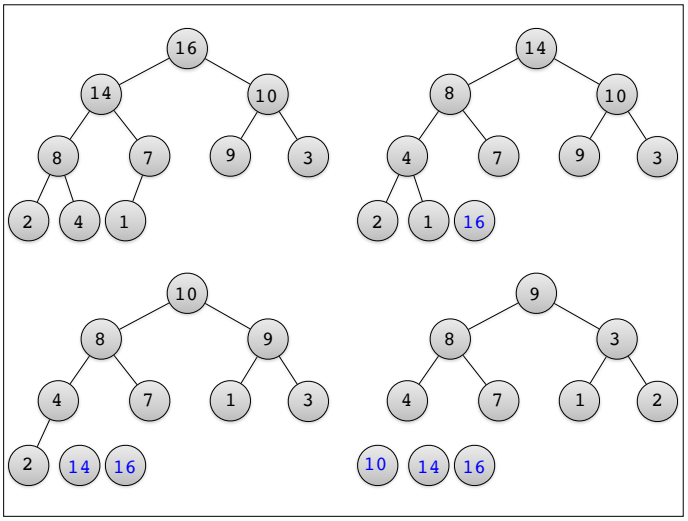


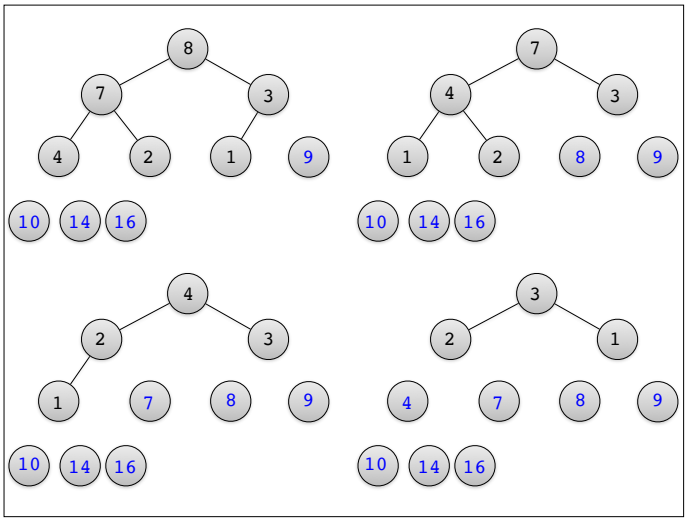
Max Heapify

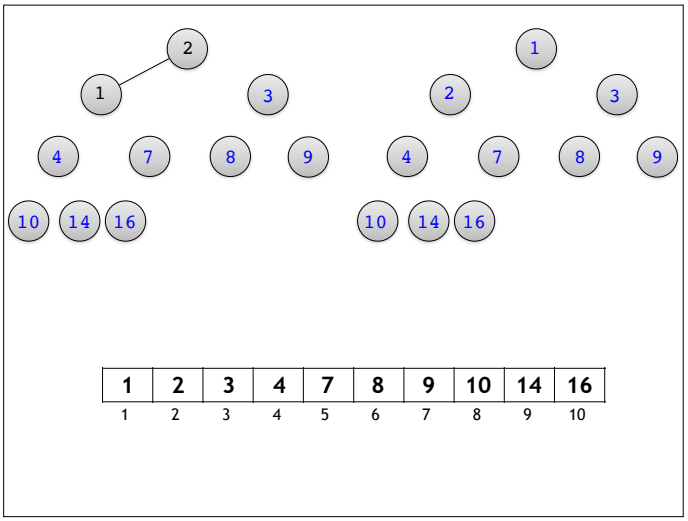



```
void max_heapify(heap *heap, int i)
{
  int largest;
  int left_index = get_left_index(heap, i);
  int right_index = get_right_index(heap, i);

  if (left_index <= heap->size &&
      heap->data[left_index] > heap->data[i]) {
    largest = left_index;
  } else {
    largest = i;
  }

  if (right_index <= heap->size &&
      heap->data[right_index] > heap->data[largest]) {
    largest = right_index;
  }

  if (heap->data[i] != heap->data[largest]) {
    swap(&heap->data[i], &heap->data[largest]);
    max_heapify(heap, largest);
  }
}
```

Heapsort







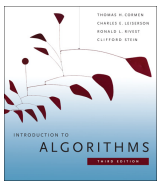
1	2	3	4	7	8	9	10	14	16
1	2	3	4	5	6	7	8	9	10

```
void heapsort(heap *heap)
{
  int i;
  for (i = heap->size; i >= 2; i--) {

    swap(&heap->data[1], &heap->data[i]);

    heap->size--;

    max_heapify(heap, 1);
  }
}
```

Execution: Heap Sort

6 5 3 1 8 7 2 4

References



Chapter 6

Chapters 6 and 9