Heaps (Priority Queues)

Note: Heaps' text representation will be its array representation in the format of [10 2 3] (root 10, left 2, right 3).

Introduction

- Many applications involve priorities:
  - short jobs should go first;
  - most urgent cases should go first
- Examples:
  - ordering CPU jobs
  - ordering printing jobs
  - emergency room admission processing

Priority Queue ADT

- Priority Queue operations
  - create
  - destroy
  - insert (or enqueue)
  - deleteMin (or deleteMax or dequeue)
    - Depending on how the priority is defined
  - is_empty
- Priority Queue property:
  - For two elements in the queue, *x* and *y*, if *x* has a lower (or higher) priority value than *y*, *x* will be deleted before *y*

Possible implementations

To support the operations of insert() and deleteMin(), we could use

- Unsorted list:
  - Unsorted lists give us fast inserts, but need to search the whole list to delete.
- Sorted list:
  - Sorted lists give us fast delete (it's just the first element), but we have to either search the whole list (for linked list) or move the whole list (for array) on an insert.
- Binary Search Tree:
  - Depending on how balanced they are, can be close to list.
  - Could be an overkill since it supports many operations that are not required.
- Or Heap !

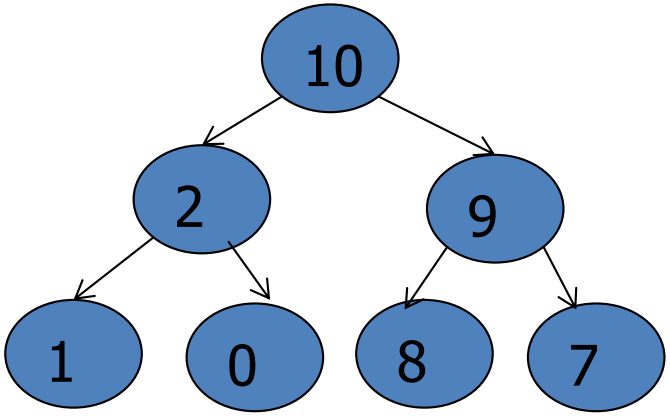
Heap (Max Heap)

- Definition:
  - A binary tree
  - The value of each node is **greater than** or equal to the values stored in each of its children. (Heap-order property)
  - The tree is completely filled, with the possible exception of the bottom level, and the leaves in the bottom level are all in the leftmost positions. (Structure property)

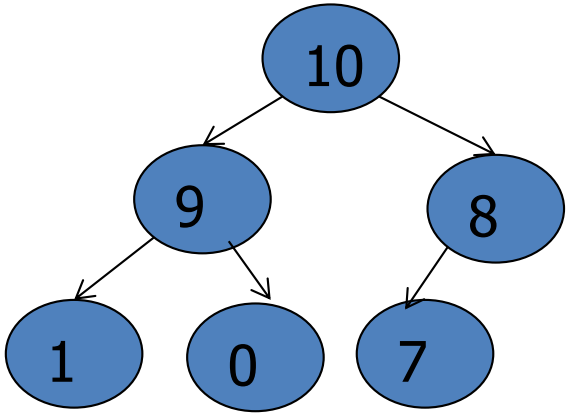
Min Heap

- Definition:
  - A binary tree
  - The value of each node is less than the values stored in each of its children.
  - Same structure property as max heap.

Example of Heaps



[10 2 9 1 0 8 7]



[10 9 8 1 0 7]

- Due to the structure property, the number of levels are
  - $O(\log_2 n)$

Array Organization

- Heaps are often stored in arrays due to its regularity.
  - Sequential location from the top to bottom, left to right.
  - Example: [10 2 9 1 0 8 7]

Values in the Array

- Values start at subscript 0 (Used in later codes):
    - Root is at [0]
    - Left child: [2*parent's position +1]
    - Right child: [2*parent's position +2]
    - Parent: [(either child's position-1)/2]
    - Next free location: [number of elements]
- Some implementations use values start at subscript 1, in which case:
    - Root is at [1]
    - Left child: [2*parent's position]
    - Right child: [2*parent's position +1]
    - Parent: [either child's position/2]
    - Next free location: [number of elements + 1]

Operations

- Enqueue
- Dequeue
- buildHeap

Enqueue (into a max heap)

Insert(e)

- Put *e* at the end of heap
- while *e* is not in the root and *e* > parent(*e*)
  - Swap *e* with its parent

Dequeue (from a max heap)

1. Extract the element from the root
2. Put the element from the last leaf in its place;
3. Remove the last leaf;
4. Move down the element in the root

//pseudo code for moving down (or percolate down)

```
p = root;
While p is not a leaf and p < any of its children
   swap p with the larger child
```

Build Heap with Array

- Two approaches (bottom-up or top-down) with different time complexity.

Bottom-up approach (Heapify)

- Pretend it's a heap and fix the heap-order property.
- Start from the last nonleaf node at [n/2-1] to root (n is the number of nodes),
- move down (i.e. Check if it is less than one of its children and repeat until reach the leaf)

- Algorithm:

  For i = index of the last nonleaf node down to 0

  moveDown(data, i, n-1)

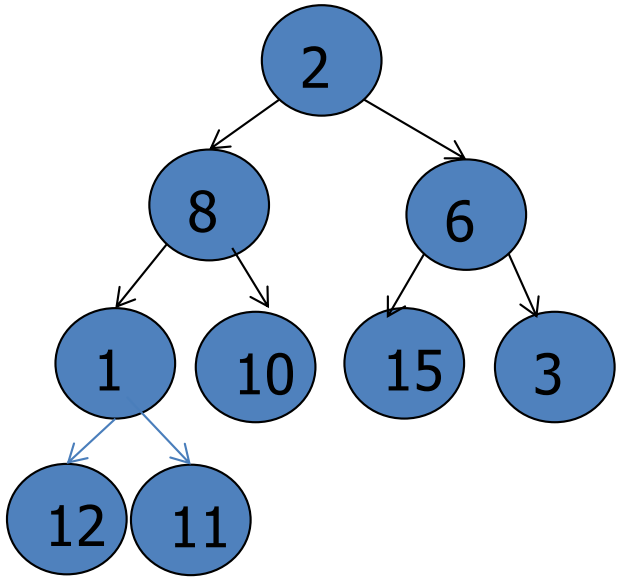
moveDown()

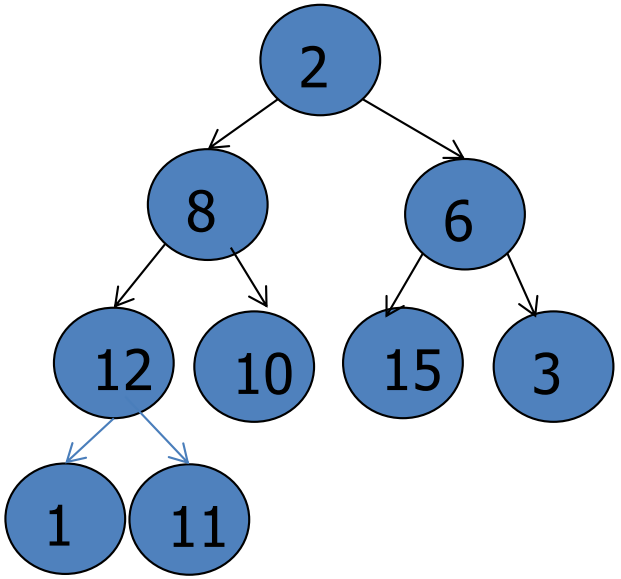
```
template<class T>
void moveDown(T data[], int first, int last)
{
   int larger = 2*first +1;   //the left child
   while (larger <=last)    //beyond   the last level
   {
      if (larger < last &&  data[larger] < data[larger+1]) //first has 2 children and the right is larger
            larger ++;
     if  (data[first] < data[larger])     //if necessary
     {
         swap(data[first], data[larger]);
         first = larger;          // first is the position of previous larger child, move down for next iteration
         larger = 2*first +1;
      }
     else
          larger = last + 1;   // just to exit the loop
   }
}
```

Discussion

- Make sure not to assume that there are always two children. – Extra test in the code.
- Time Complexity for moveDown()
  - The worst-case for moveDown() is O(logN)
  - The average time is also O(logN)

Example: Transforming [2 8 6 1 10 15 3 12 11] into a heap

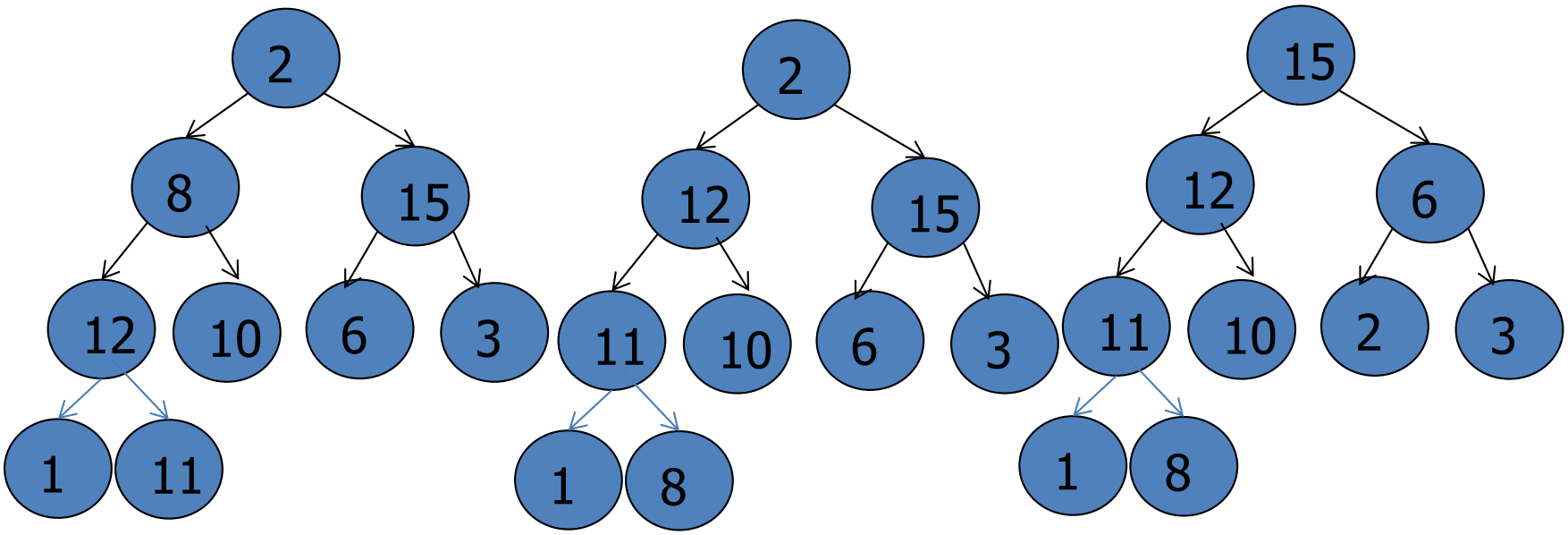




->Move down 1 (swap 1 and 12)

-> Move down 6

[2 8 6 12 10 15 3 1 11]



-> Move down 8 -> Move down 2

[2 8 15 12 10 6 3 1 11] [2 12 15 11 10 6 3 1 8] [15 12 6 11 10 2 3 1 8]

Linear Complexity in Worst Case

- Complexity of Heapify an array in a bottom-up way:
  - worst case: O(n)
  - Why?

  - Assume *the **height of the node** is defined as steps it needs to reach the leaf level*.
  - For a complete tree of height h (perfect with total nodes 2^h-1), the sum of the total heights of the nodes is 2^h-1-h.

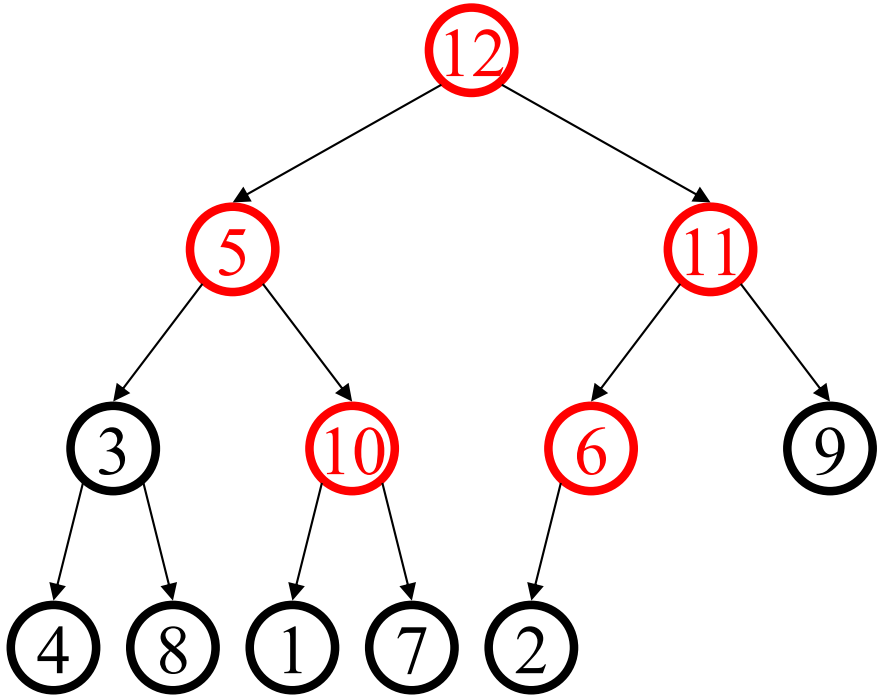
    Example: h = 3, nodes 7
    - Total heights: 2 + 1 + 1 = 4 (= 7-3)

  - Since every node at most moves down its height, the total number of swaps is linear to the total number of nodes!
  - It is an upperbound for trees that are not perferct.

Another Example

Build a min-heap from the following array:

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



Build heap: Top-down Approach

- Top-down approach
  - Repeatedly perform enqueue()
  - Worst Case Complexity is O(nlgn)
    - Slower than the Bottom-up approach
    - Why?
      - Enqueue() is O(logn) worst case. So: building heap: O(nlgn)
- The average case of the two approaches for building heap are similar: O(n).

HeapSort

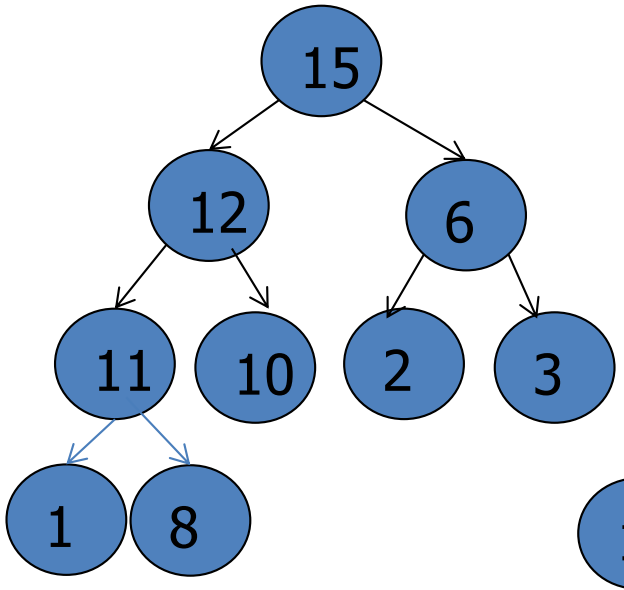
1. Heapifies the list of numbers.
2. The element at the beginning of the list is then swapped with the element at the end of the list. The list is made 1 element shorter. The new list is then heapified.
3. The process continues until the entire list is sorted.

Pseudo Code

- Heapsort
  - Transform into a heap
  - For i from n-1 down to 2 (n is the size)
    - Swap the root with the element in position *i*
    - Fix heap property

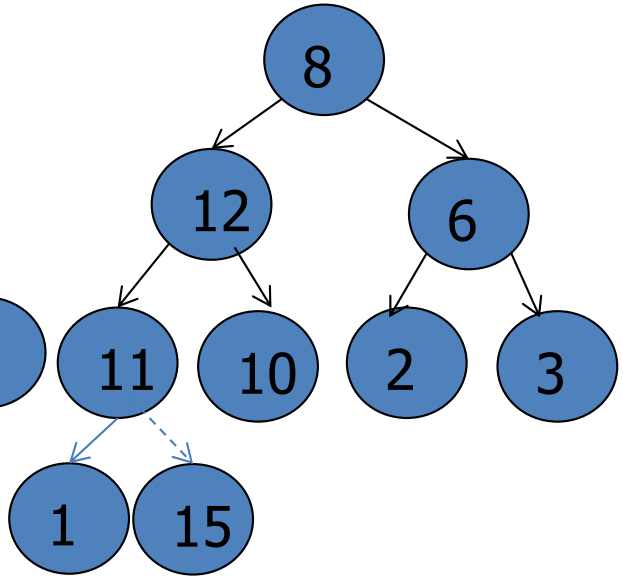
Example: sort the array [2 8 6 1 10 15 3 12 11]

1. heapify



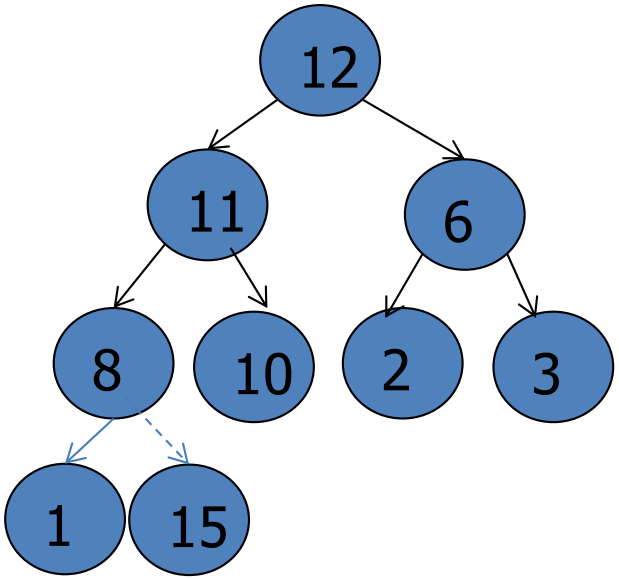
[15 12 6 11 10 2 3 1 8]

2. Swap 15,8



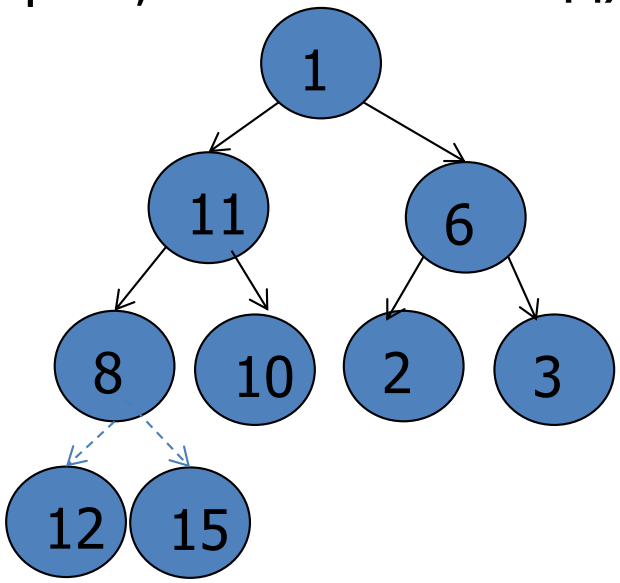
[8 12 6 11 10 2 3 1 15]

3. Fix heap-order



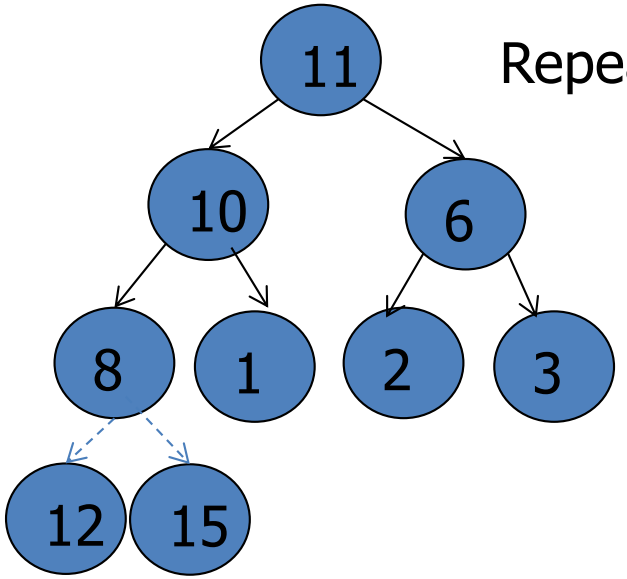
[12 11 6 8 10 2 3 1 15]

swap 12,1



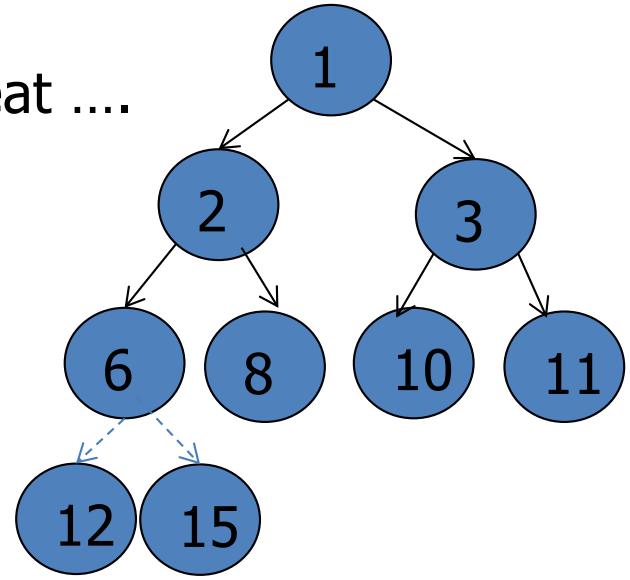
[1 11 6 8 10 2 3 12 15]

Fix heap-order



[11 10 8 1 2 3 12 15]

Repeat ….



[1 2 3 6 8 10 11 12 15]

Algorithm

```
template <class T>
void heapsort(T data[], int size)
{
     for (int i = size/2 -1; i >= 0; --i) //create the heap
          moveDown(data, i, size-1);
     for (int i = size-1; i>= 1; -- i)
     {
          swap(data[0], data[i]);
          moveDown(data, 0, i-1);
     }
}
```

Heapsort Complexity

- Worst case: O(NlgN)
- Best case: O(N)

- Why?
  - 1st step for construction, O(N) for bottom-up approach
  - n-1 swaps,
  - O(nlgn) to restore the heap property,
- So total: O(N) + O(nlgn) + n-1 = O(nlgN)

- The best case happens when all the elements are identical – movedown() is called but no moves are necessary.
- Ironically, the worst case of heap sort happens when the heap is already sorted.