Heaps

1 Heaps

Review—Tree Definitions

- **Full binary Tree** a binary tree of height h with no missing nodes. All leaves have the same depth and every non-leaf has two children.
- **Complete binary Tree** a binary tree of height h that is full to level $h-1$ and has level h filled in from left to right.
- **Balanced Binary Tree** a binary tree in which the left and right subtrees of any node have heights that differ by at most 1.

What is a Heap?

A heap is a certain kind of *complete* binary tree.

Each node in a heap contains a key, and these keys are organized in a particular manner. Notice that this is *not* a binary search tree, but the keys have some semblance of order.

This is a useful property because the largest (smallest) node is always at the top. Because of this, a heap can easily implement a priority queue (where we need quick access to the highest (or lowest) priority item).

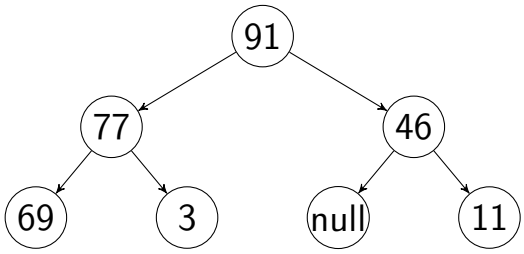
The operations for both insertion and deletion are $O(logN)$.

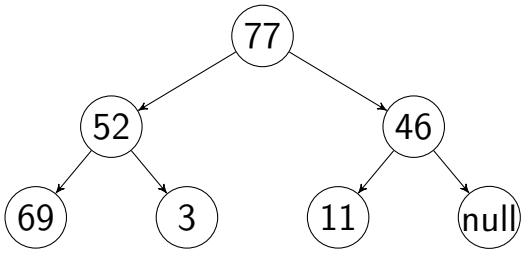
Heap Storage Rules (*Max*-Heap)

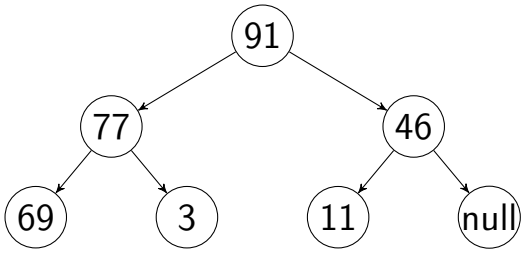
1. The entry contained by a node is greater than or equal to the entries of the node's children.
2. The tree is a complete tree, so that every level except the deepest must contain as many nodes as possible. At the deepest level, all the nodes are as far left as possible.

A *Min*-Heap can be defined similarly.

Which tree is a *max*-heap?







The top tree is not a heap because it is not complete—recall that a complete tree must have the nodes on the bottom level as far left as possible.

The middle tree is not a heap because one of the nodes (52) has a value that is smaller than its child (69).

The bottom tree is a max-heap.

Heap Implementation

Since a heap is a complete binary tree, and a complete binary tree is more easily implemented using an array than with pointers. If the maximum size of a heap is known in advance, then an array implementation can use a fixed-size array.

If there are n nodes, only the first n positions of an array, A, are used.

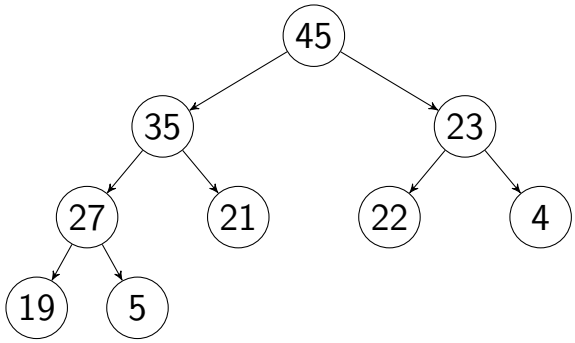
Useful parent-child node relationships:

- The left child of a node is: $2i + 1$
- The right child of a node is: $2i + 2$
- The parent of a node is: $(i - 1)/2$

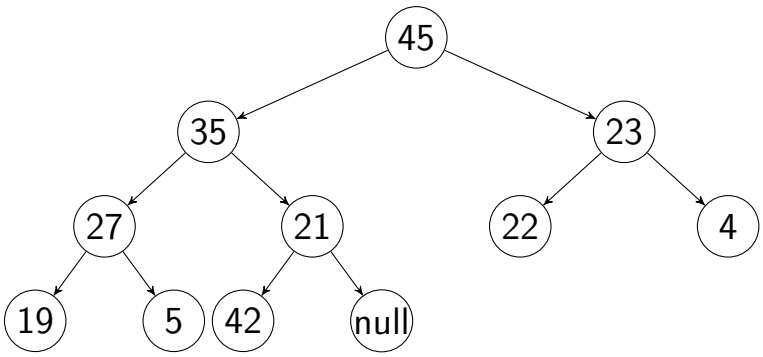
where i is the index of the node in the array.

Adding and removing entries from a heap often requires the heap to be *reheapified*.

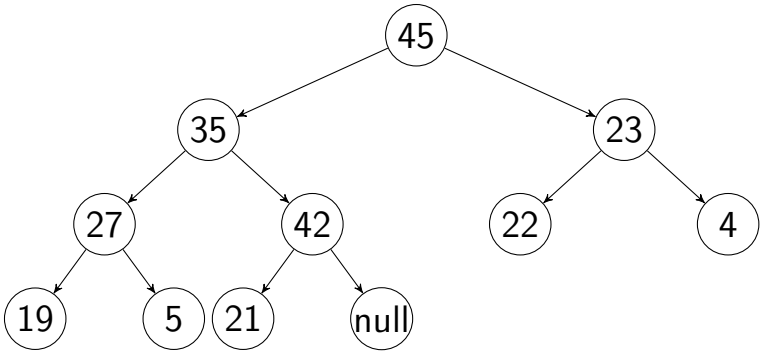
Consider adding an entry with a priority of 42 to the following heap:

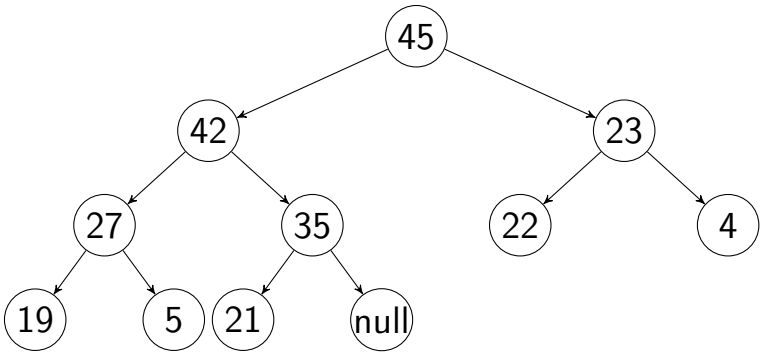


Insert 42 (rightmost leaf)



Reheapification





Pseudocode for Adding an Entry

1. Place the new entry in the heap in the first available location.
   This keeps the structure as a complete binary tree, but it might no longer be a heap since the new entry might have a higher priority than its parent.

2. *while* (the new entry's priority is higher than its parent)
   Swap the new entry with its parent.

Pseudocode for Deleting the Top Entry

1. Move the last node of the tree into the root.
2. Move the out-of-place node downward, swapping with its *larger* child until the new node reaches an acceptable location.

Code For Insertion (Pascal)

Data Structures and Algorithms, 1983
Alfred V. Aho, Bell Laboratories, Murray Hill, New Jersey
John E. Hopcroft, Cornell University, Ithaca, New York
Jeffrey D. Ullman, Stanford University, Stanford, California

```
type

    processtype = record
      id: integer;
      priority: integer
    end;


    PRIORITYQUEUE = record
      contents: array[1..maxsize] of processtype;
      last: integer
    end;
```

C/C++ Declarations

```
struct processtype {
    int id;
    int priority;
};

typedef struct PRIORITYQUEUE {
    struct processtype contents[maxsize];
    int last;
};
```

Priority from a *processtype* record:

```
function p ( a: processtype ) : integer;
begin
    return (a.priority)
end;
```

```
procedure INSERT ( x: processtype;
                   var A: PRIORITYQUEUE );
    var
      i : integer;
      temp : processtype;

begin
      if A.last >= maxsize then
        error ('priority queue is full')

      else begin
        A.last := A.last + 1;
        A.contents [A.last] := x;
        i := A.last; { i is index of current position of x }
        while (i > 1) and
          (p(A.contents [i]) < p(A.contents [i div 2])) do
        begin { push x up the tree by exchanging it with
              its parent of larger priority. Recall p computes
              the priority of a processtype element }
          temp := A.contents [i];
          A.contents [i] := A.contents [i div 2];
          A.contents [i div 2] := temp;
          i := i div 2
        end
      end
end; { INSERT }
```