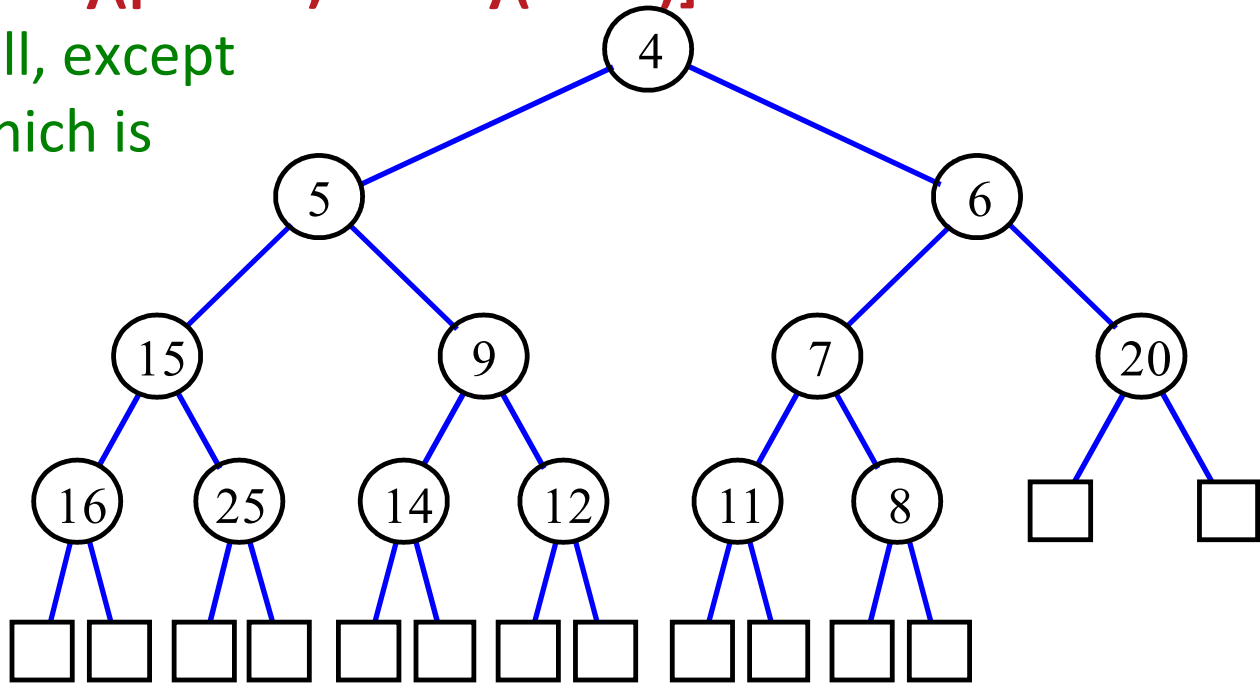
# Heaps in C

CSCI2100 Data Structures Tutorial 7

# Heaps

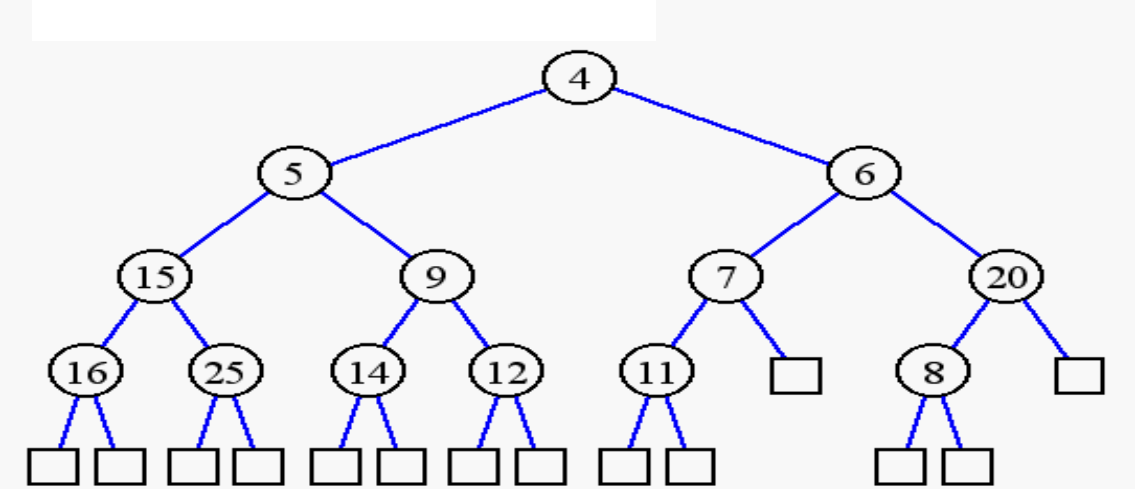
- A *heap* is a binary tree T that stores a key-element pairs at its internal nodes
- It satisfies two properties:
  - **MinHeap: key(parent) >= key(child)**
  - **[OR MaxHeap: key(parent) <= key(child)]**
  - all levels are full, except the last one, which is left-filled

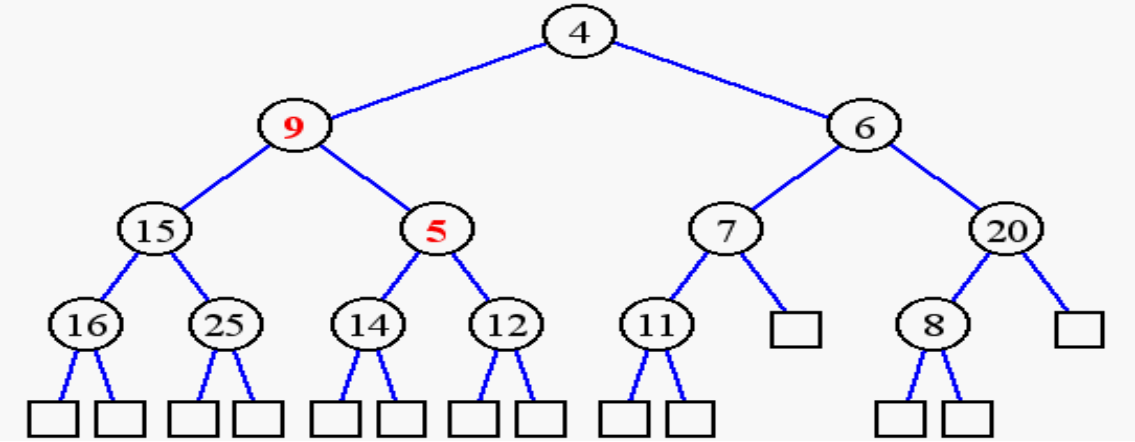


# What are Heaps Useful for?

- To implement priority queues
- Priority queue = a queue where all elements have a “priority” associated with them
- Remove in a priority queue removes the element with the smallest priority
  - insert
  - removeMin

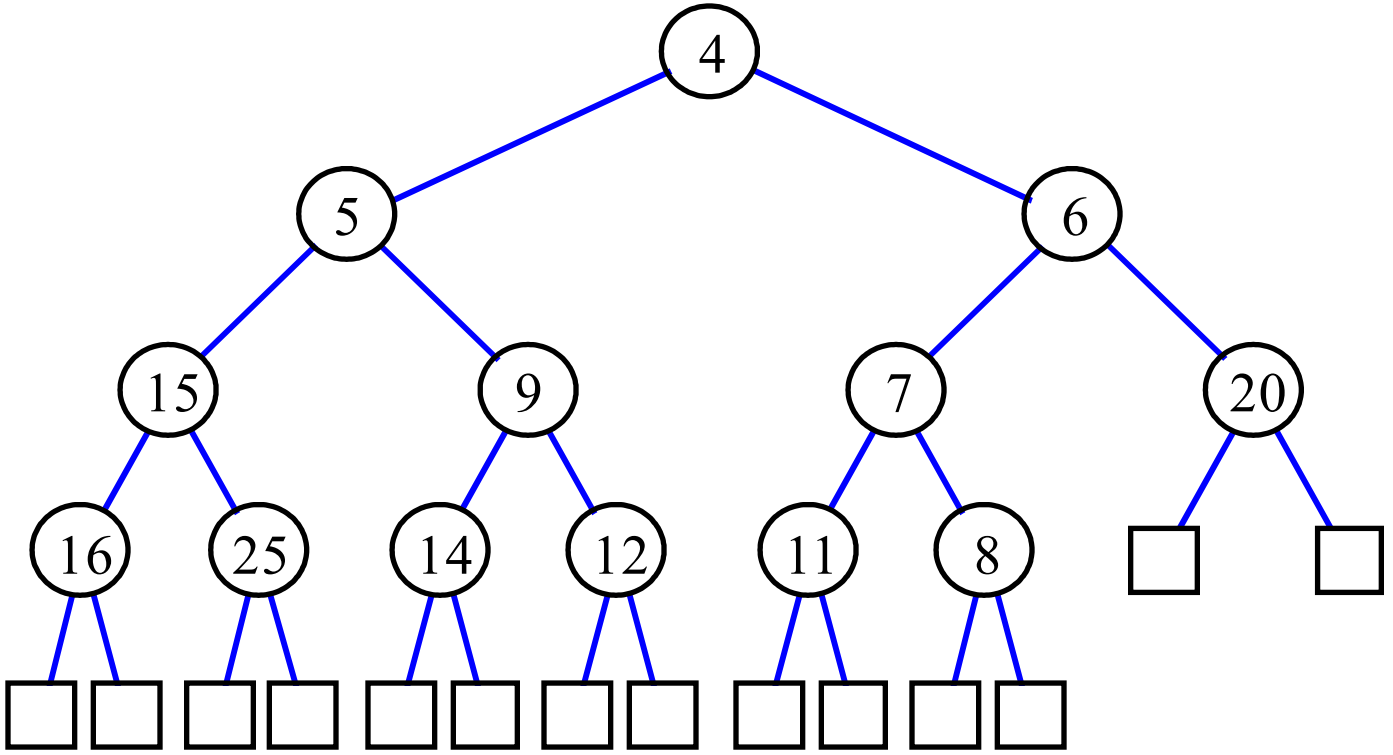
# Heap or Not a Heap?



# Heap Properties

- A heap T storing n keys has height h = ⌊logn⌋, which is O(log n)



# ADT for Min Heap

objects: n > 0 elements organized in a binary tree so that the value in each node is at least as large as those in its children

method:

Heap Create(MAX_SIZE)::= create an empty heap that can hold a maximum of max_size elements

Boolean HeapFull(heap, n)::= if (n==max_size) return TRUE
else return FALSE

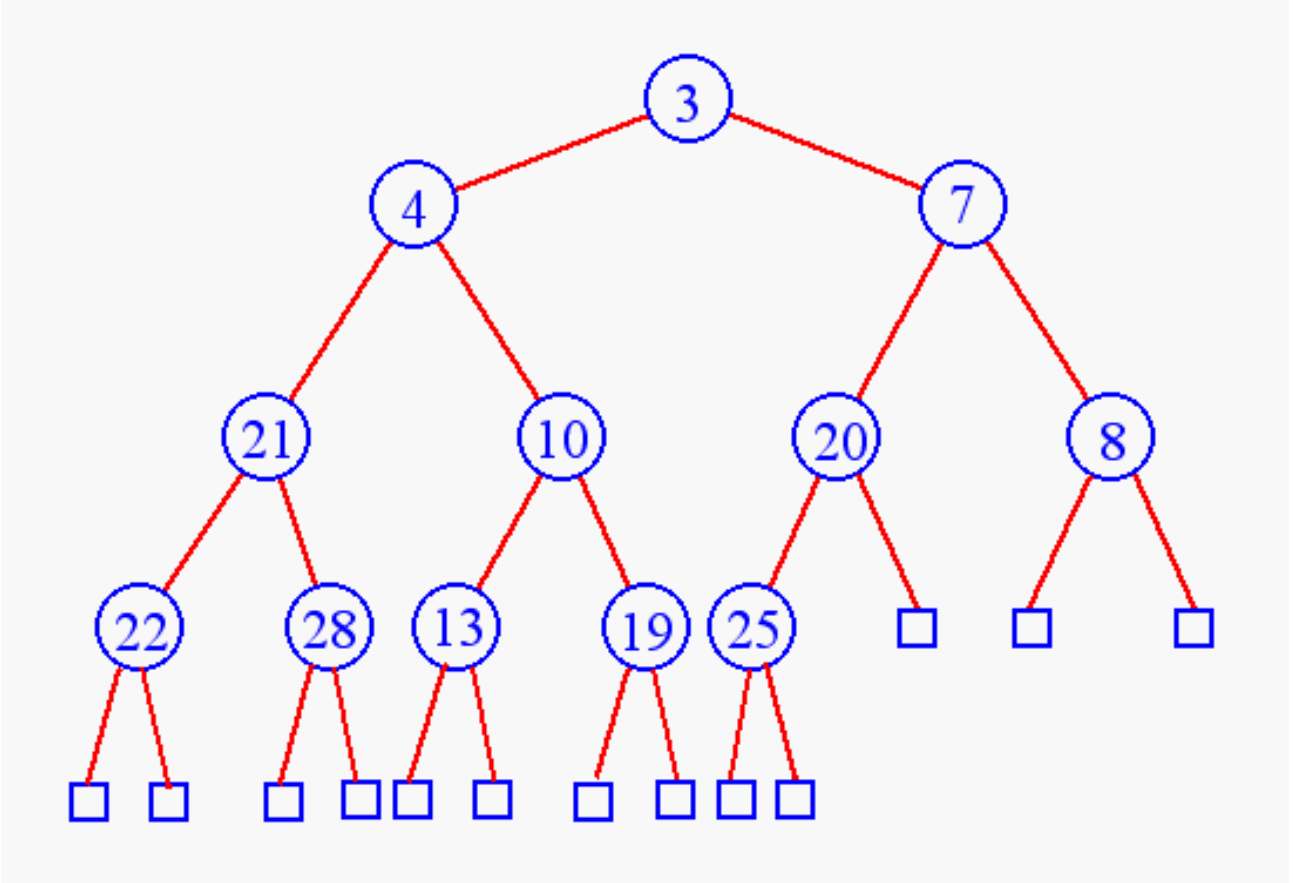
Heap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert item into heap and return the resulting heap
else return error

Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE
else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one instance of the smallest element in the heap and remove it from the heap
else return error

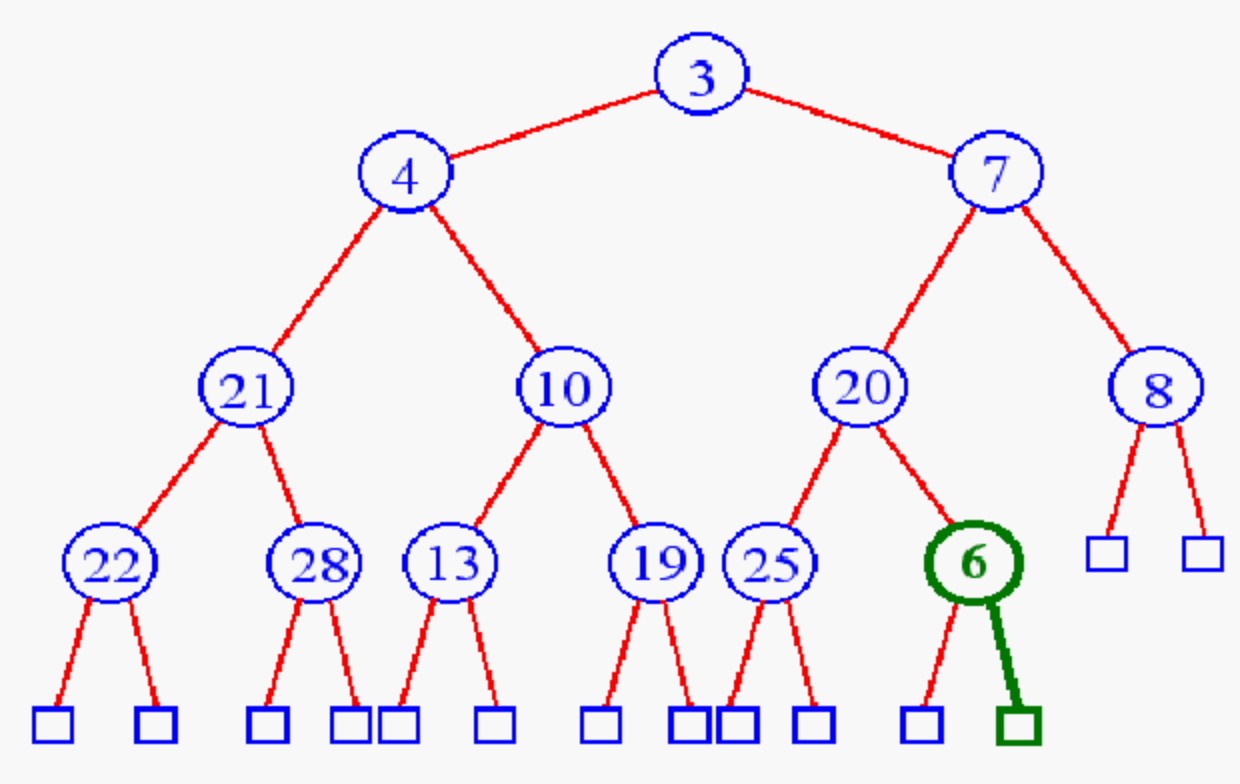
# Heap Insertion

- Insert 6



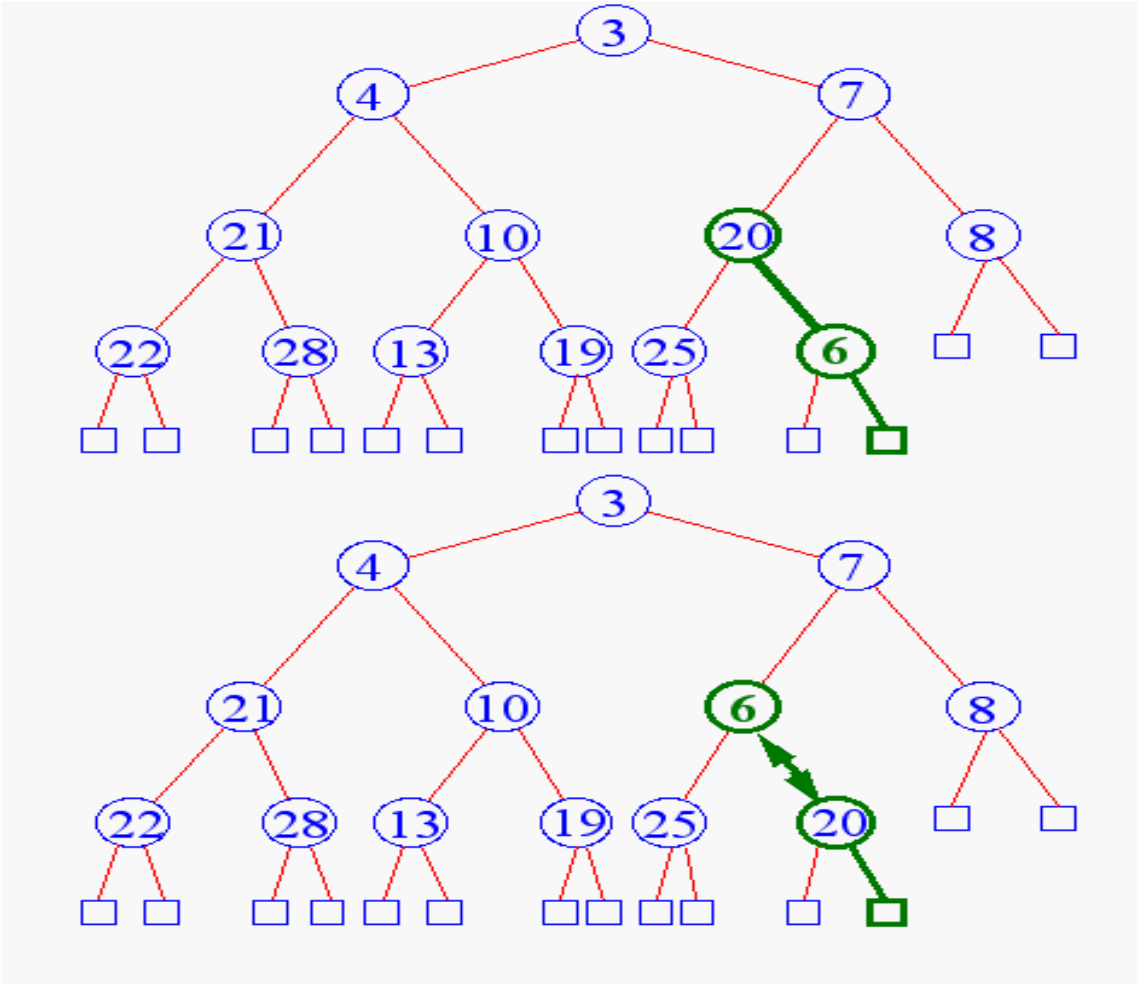
# Heap Insertion

- Add key in next available position

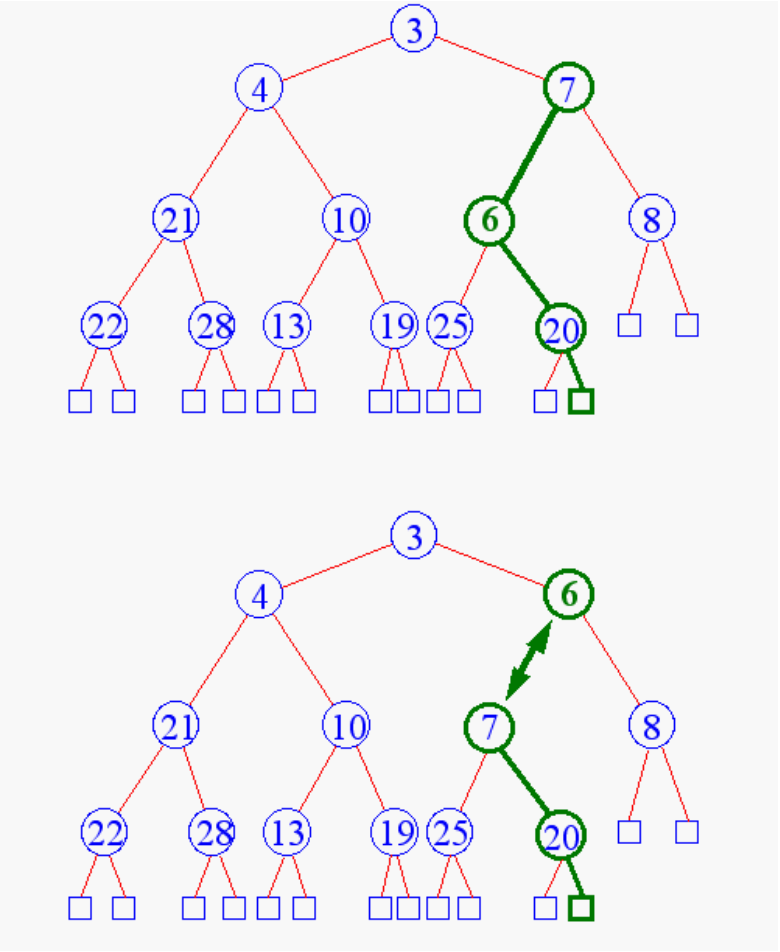


# Heap Insertion

- Begin bottom-up

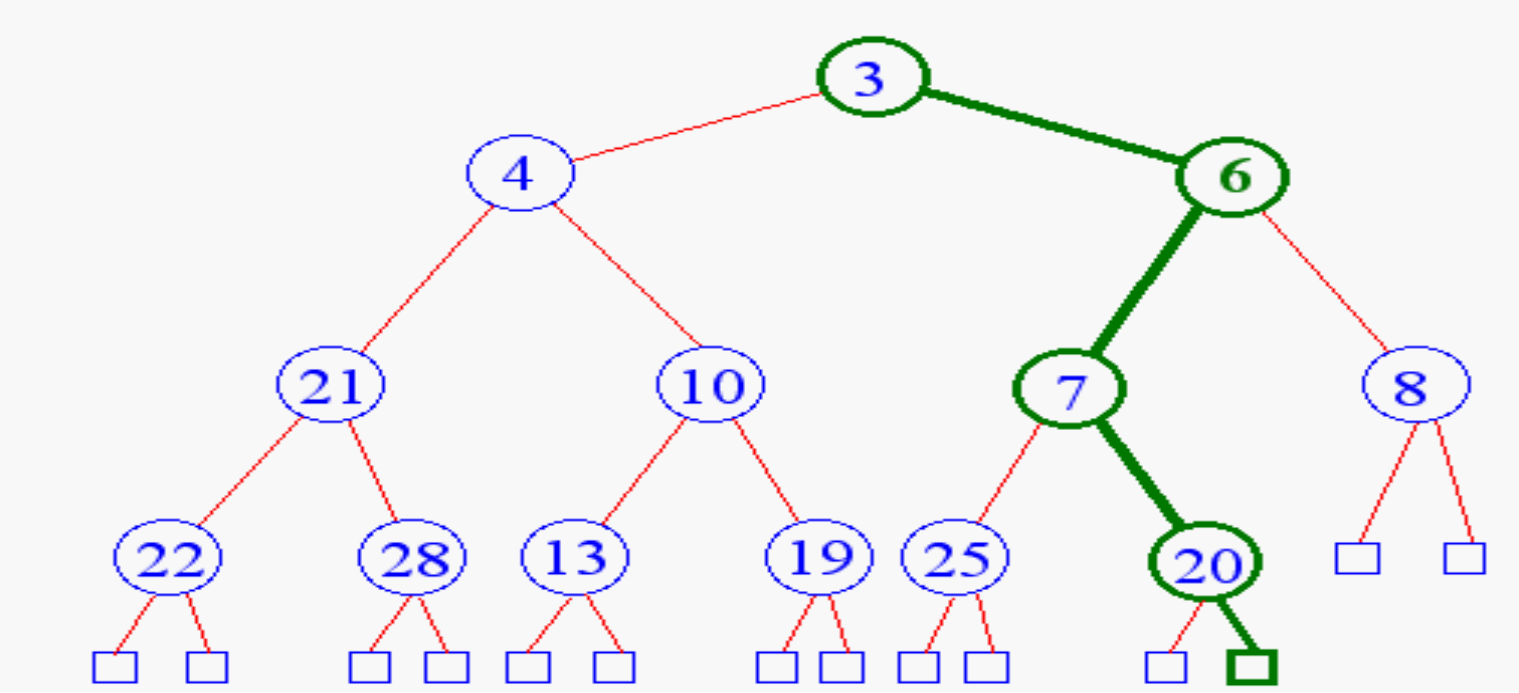


# Heap Insertion



# Heap Insertion

- Terminate bottom-up when
  - reach root
  - key child is greater than key parent

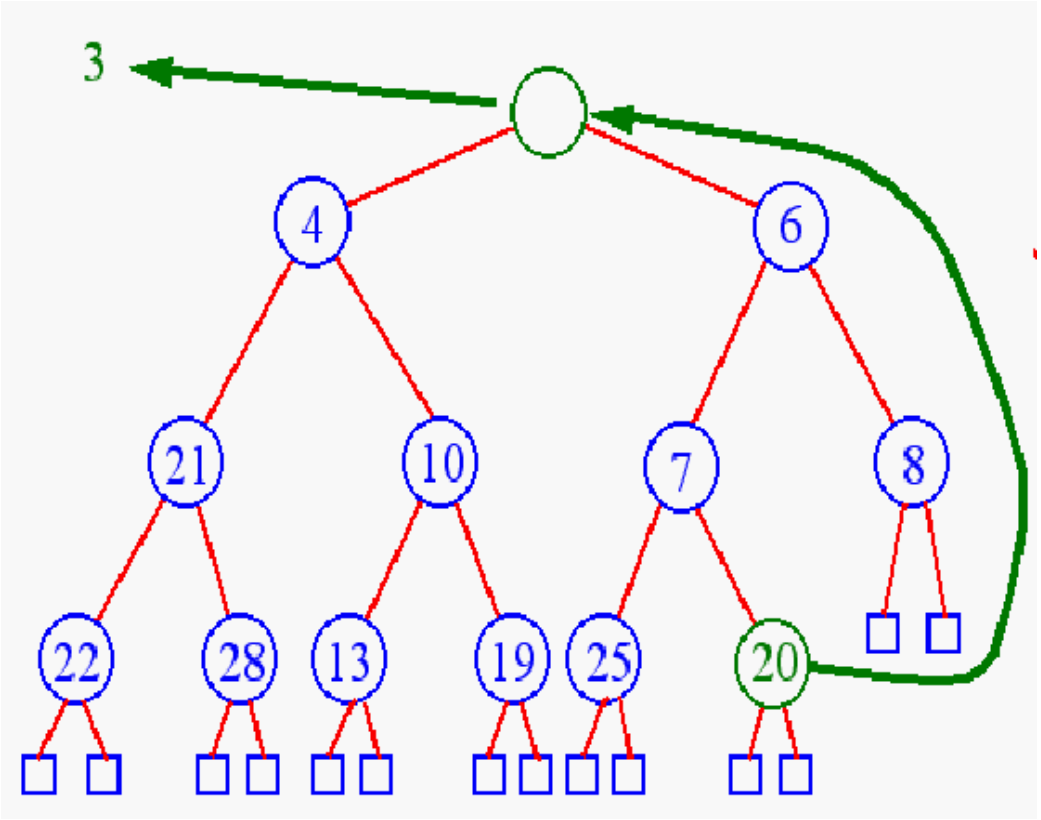


# Heap Removal

- Remove element

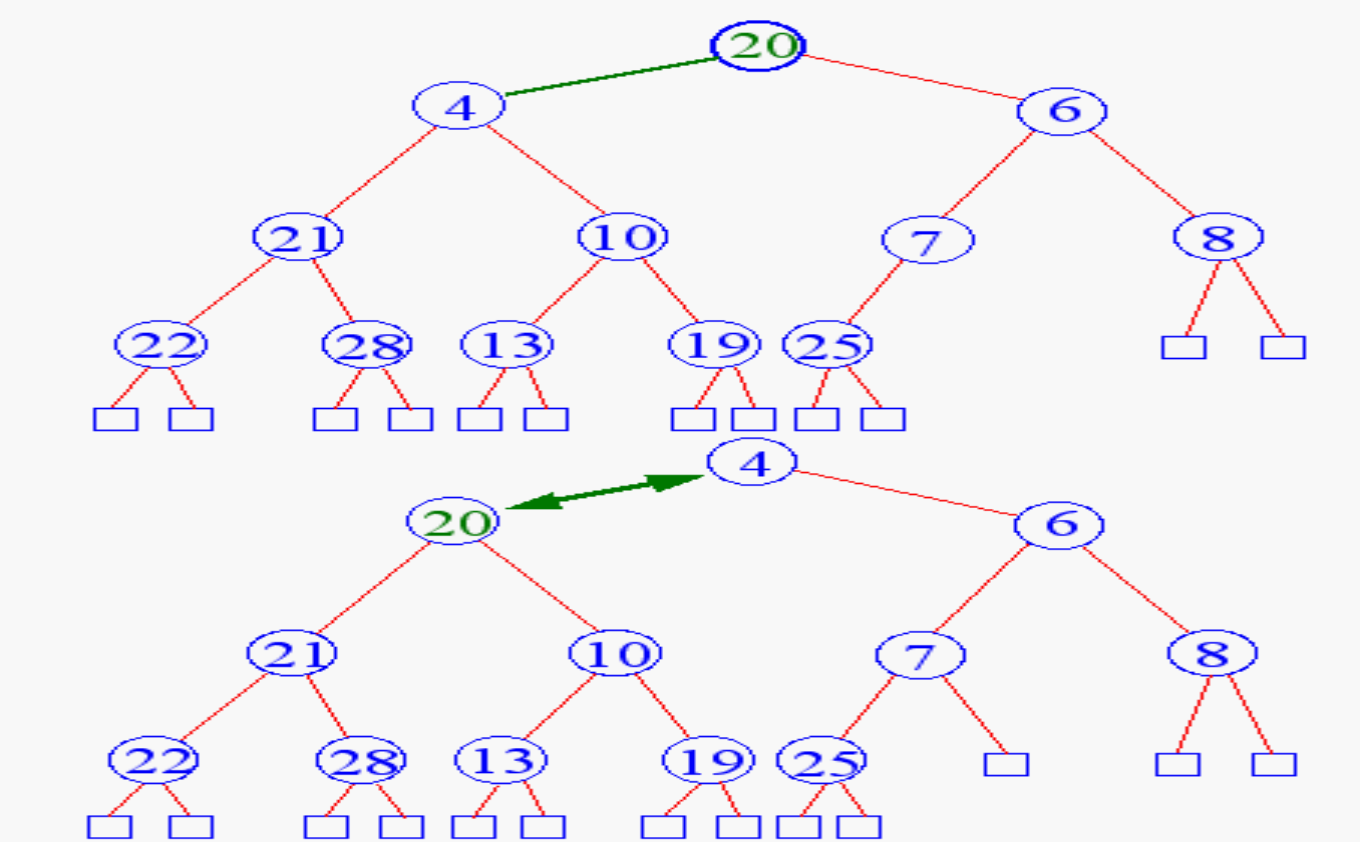
from priority queues?

removeMin( )

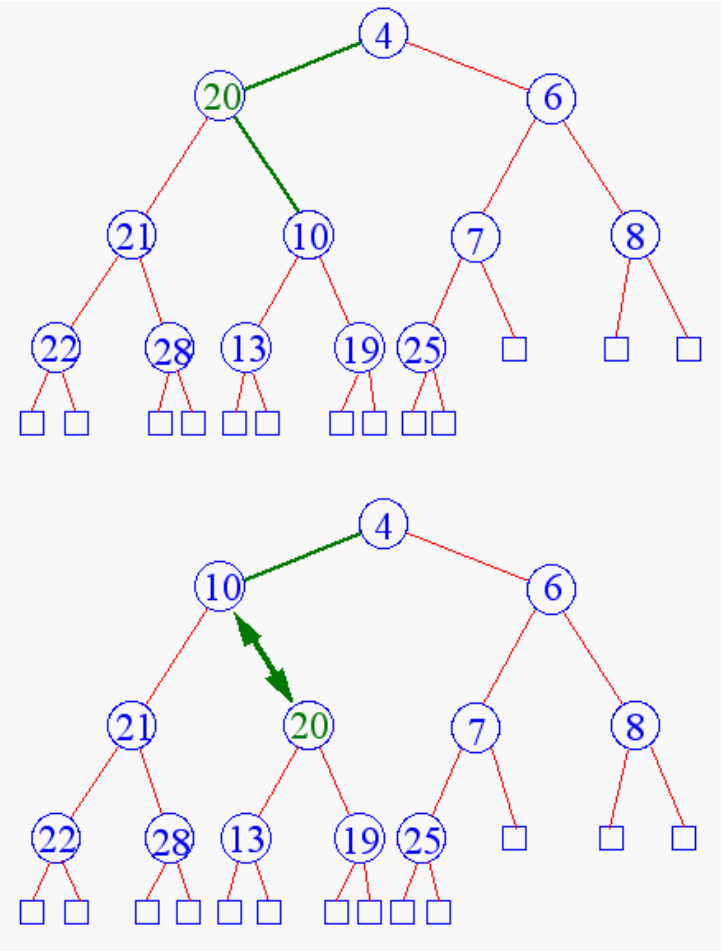


# Heap Removal

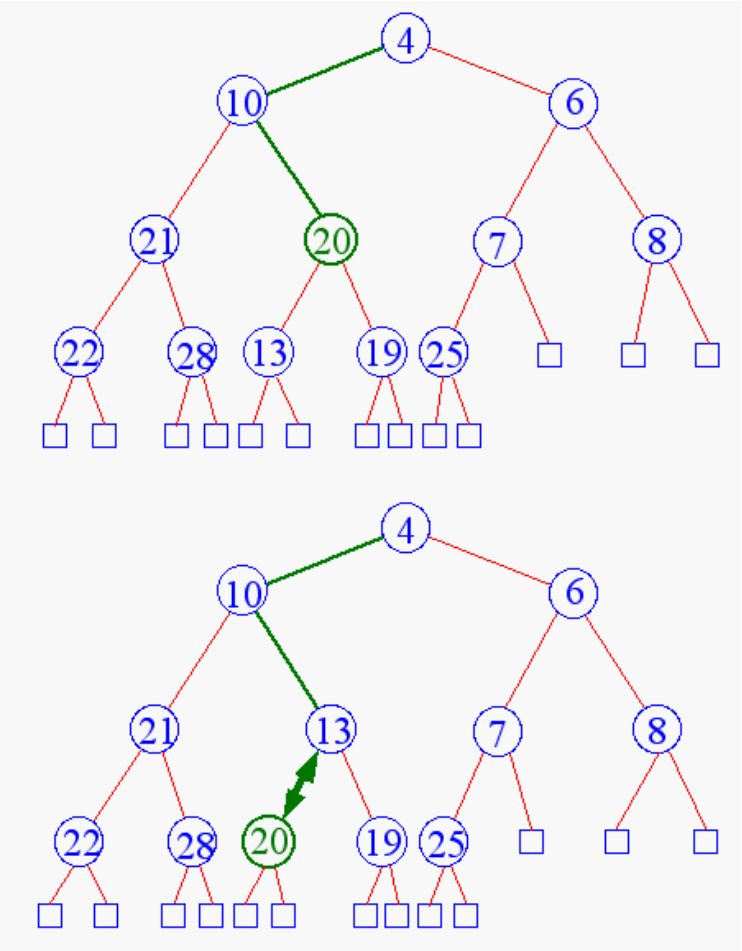
- Begin top-down



# Heap Removal

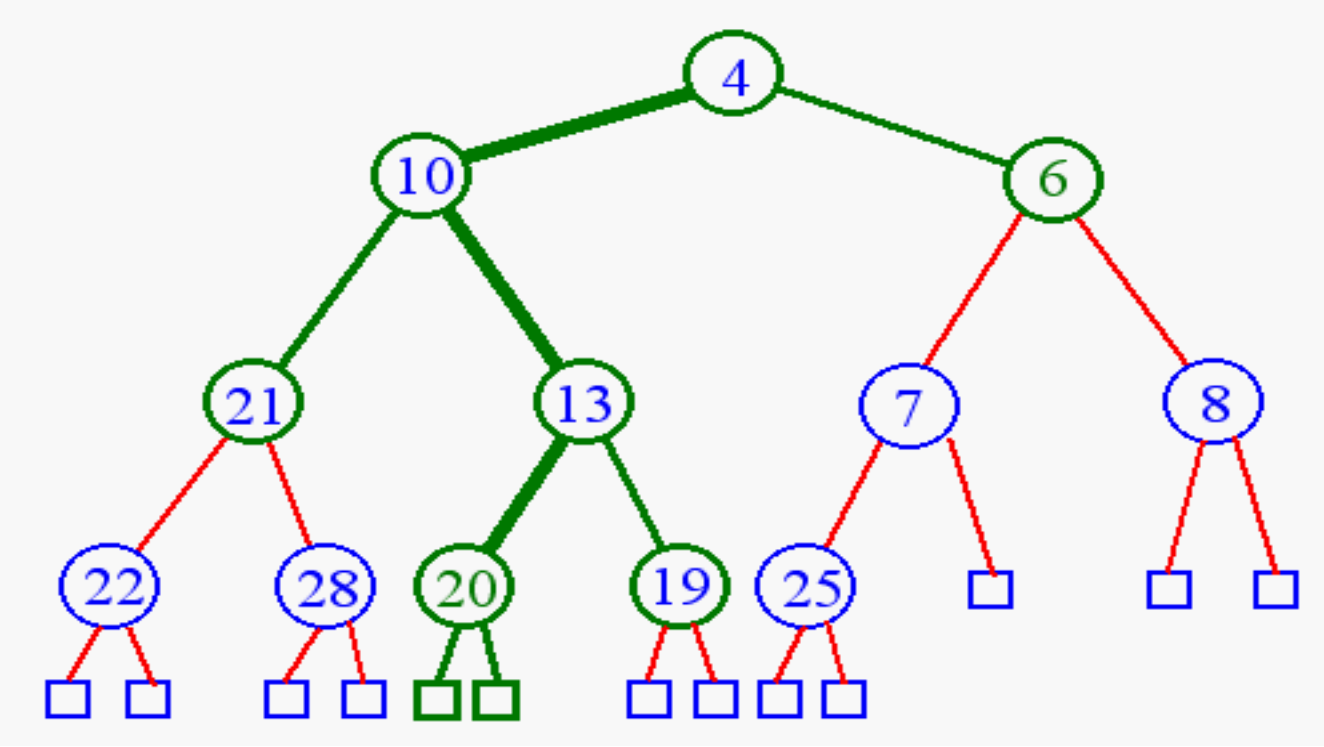


# Heap Removal



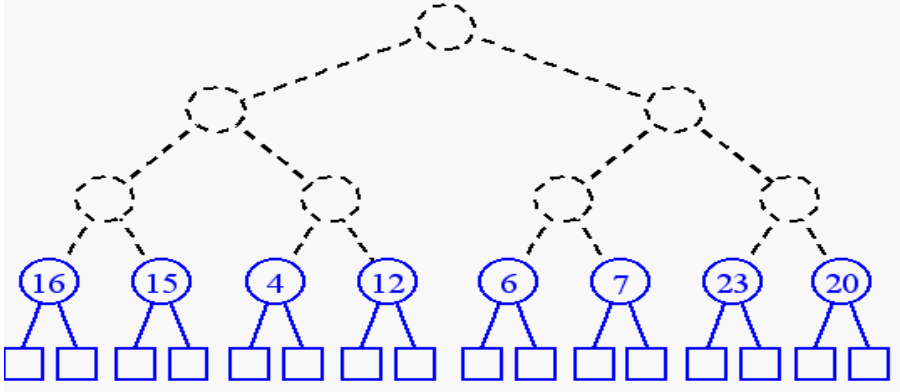
# Heap Removal

- Terminate top-down when
  - reach leaf level
  - key parent is smaller than key child

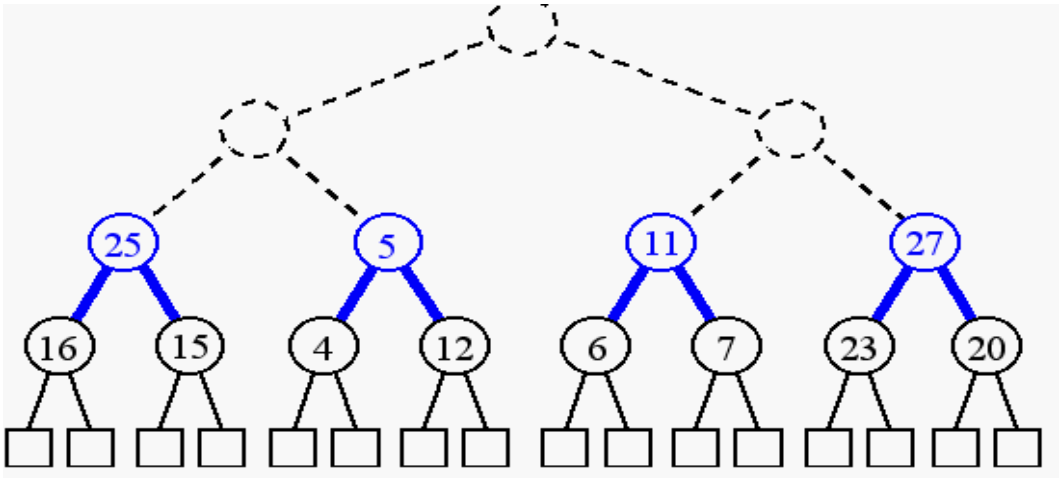


# Building a Heap

- build (n + 1)/2 trivial one-element heaps

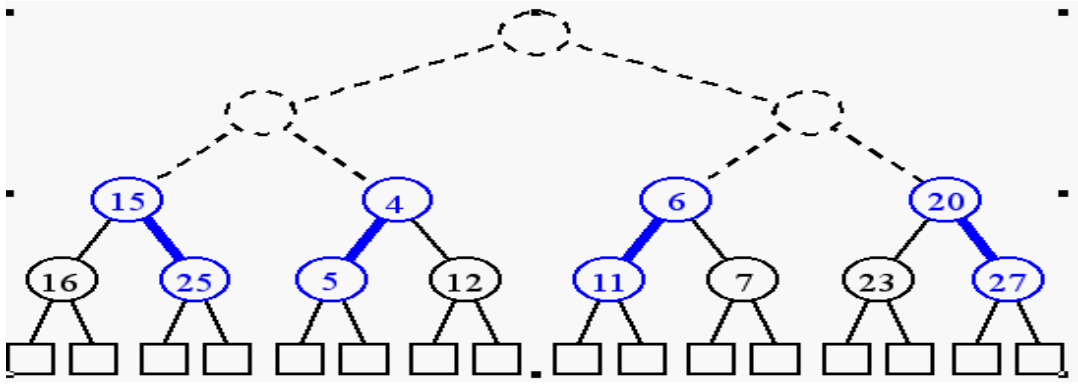



- build three-element heaps on top of them

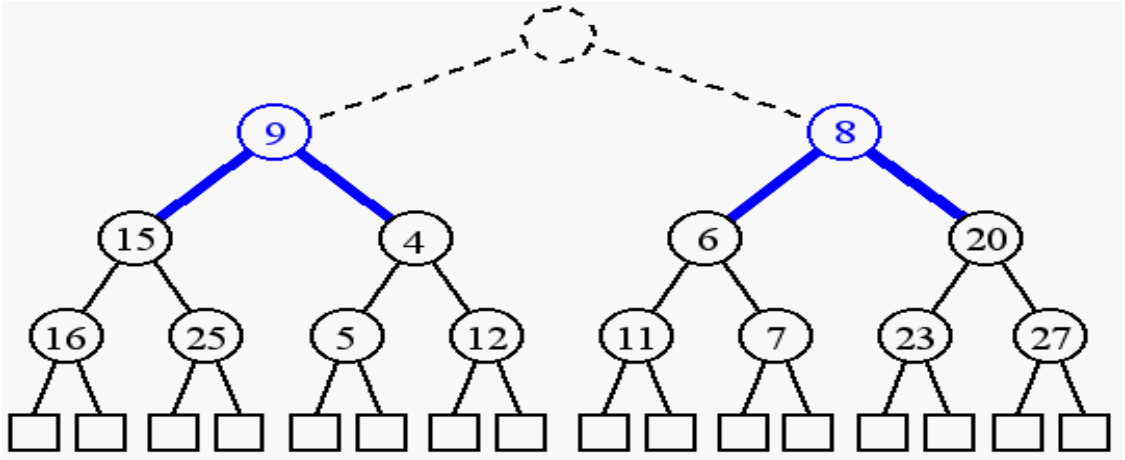


# Building a Heap

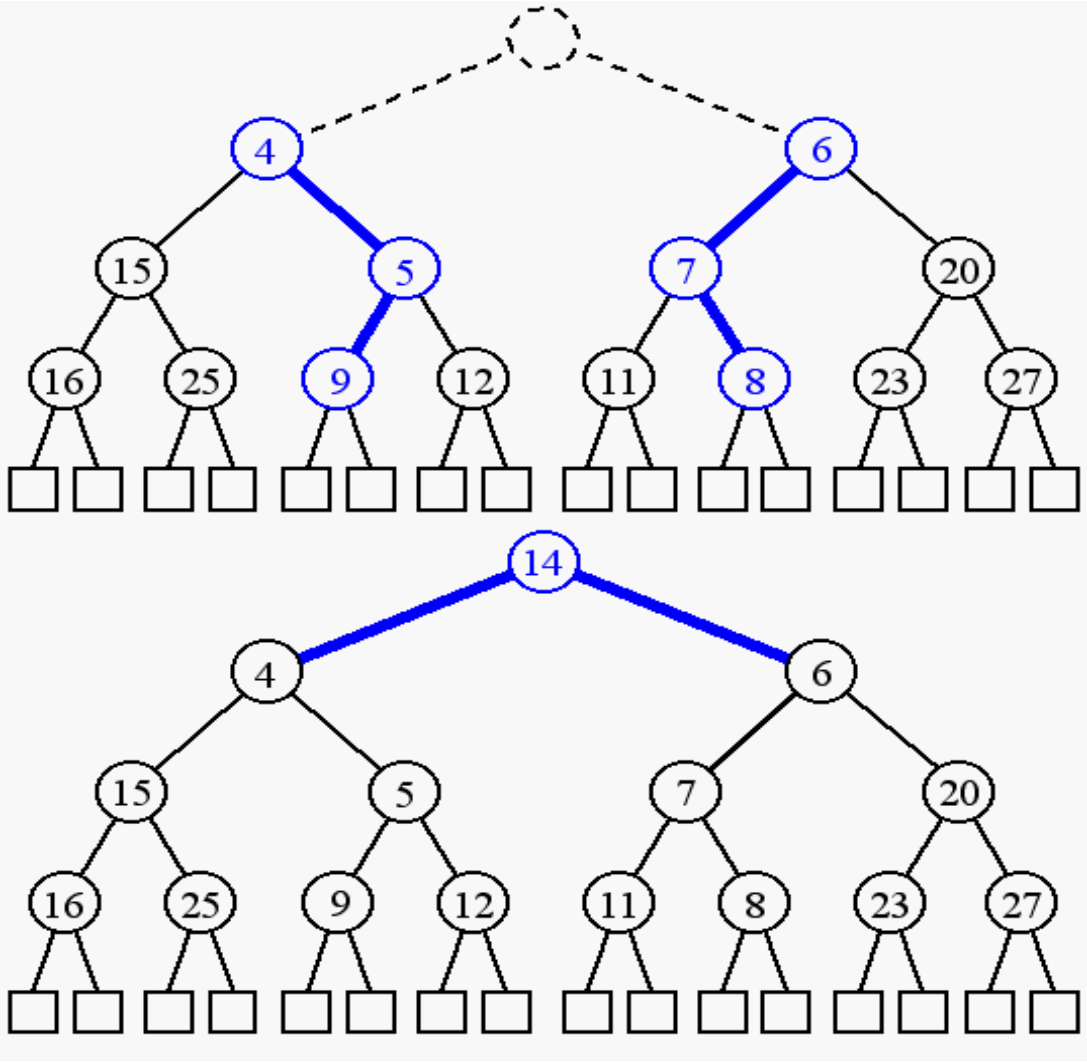
- Top-down to preserve the order property



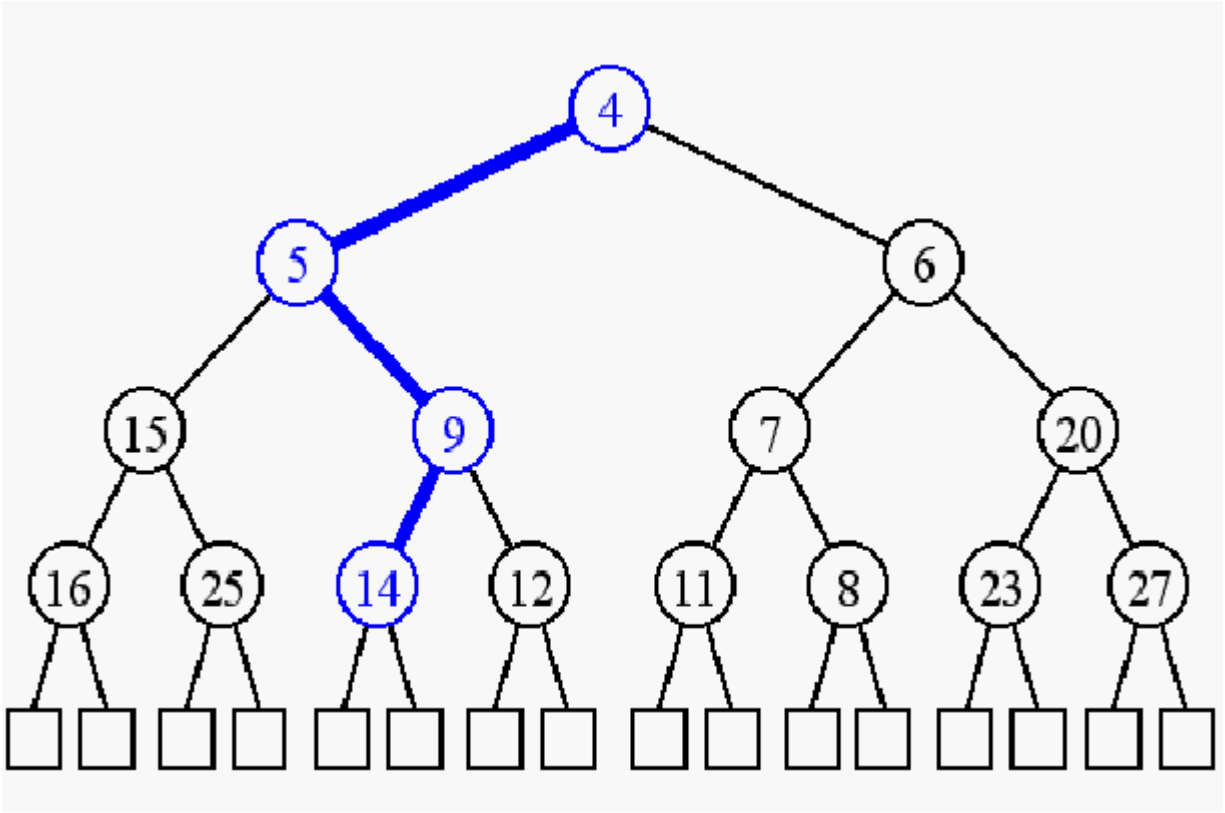

- Now form seven-element heaps



# Building a Heap

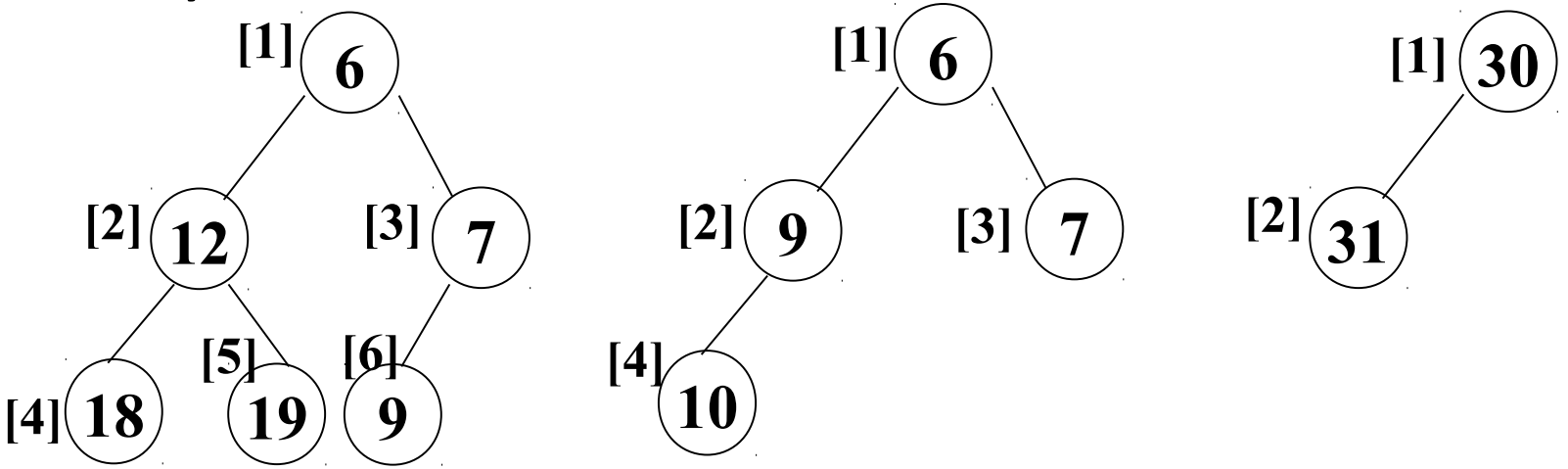


# Building a Heap



# Heap Implementation

- Using arrays
- Parent = k ; Children = 2k , 2k+1
- Why is it efficient?



# Insertion into a Heap

```
void insertHeap(element *heap, element item, int n)
{
  int i;
  if (HEAP_FULL(heap, n)) {
    fprintf(stderr, "the heap is full.\n");
    exit(1);
  }
  i = ++n;
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

$O(\log_2 n)$

# Deletion from a Heap

```
element deleteHeap(element *heap, int n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(heap, n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
   highest key */
  item = heap[1];
  /* use last element in heap to adjust heap */
  temp = heap[n--];
  parent = 1;
  child = 2;
```

# Deletion from a Heap (cont'd)

```
  while (child <= n) {
    /* find the larger child of the current
       parent */
    if ((child < n)&&
        (heap[child].key<heap[child+1].key))
      child++;
    if (temp.key >= heap[child].key) break;
    /* move to the next lower level */
    heap[parent] = heap[child];
    parent = child;
    child *= 2;
  }
  heap[parent] = temp;
  return item;
}
```

# Heap Sorting

- Step 1: Build a heap
- Step 2: removeMin( )

- Running time for build a heap?
- For index $\leftarrow \lfloor n/2 \rfloor$ downto 1 Do
  - Downheap(index)

- Hint: O(N) with observation, there're at most $\lfloor n/2^h \rfloor$ nodes with height $h$ $(0 \leq h \leq \lfloor \log n \rfloor)$

# A quick start tutorial for GDB

```c
/* test.c */
/* Sample program to debug. */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char **argv)
{
    if (argc != 3)
        return 1;
    int a = atoi (argv[1]);
    int b = atoi (argv[2]);
    int c = a + b;
    printf ("%d\n", c);
    return 0;
}
```

# A quick start tutorial for GDB

- Compile with the -g option:
  - gcc -g -o test test.c
- Load the executable, which now contain the debugging symbols, into gdb:
  - gdb  test

# A quick start tutorial for GDB

- Now you should find yourself at the gdb prompt. There you can issue commands to gdb.
- Say you like to place a breakpoint at line 11 and step through the execution, printing the values of the local variables - the following commands sequences will help you do this:

# A quick start tutorial for GDB

```
(gdb) break test.c:11
Breakpoint 1 at 0x401329: file test.c, line 11.
(gdb) set args 10 20
(gdb) run
Starting program: c:\Documents and Settings\VMathew\Desktop/test.exe 10 20
[New thread 3824.0x8e8]

Breakpoint 1, main (argc=3, argv=0x3d5a90) at test.c:11
(gdb) n
(gdb) print a
$1 = 10
(gdb) n
(gdb) print b
$2 = 20
(gdb) n
(gdb) print c
$3 = 30
(gdb) c
Continuing.
30

Program exited normally.
(gdb)
```

# Commands all you need to start:

```
break file:lineno - sets a breakpoint in the file at lineno.
set args - sets the command line arguments.
run - executes the debugged program with the given command line arguments.
next (n) and step (s) - step program and step program until it
                        reaches a different source line, respectively.
print - prints a local variable
bt -  print backtrace of all stack frames
c - continue execution.
```

- Type help at the (gdb) prompt to get a list and description of all valid commands.

# Further GDB guides

- Peter's GDB tutorial http://dirac.org/linux/gdb/
- Tutorial on using the GDB debugger (Video) http://www.youtube.com/watch?v=k-zAgbDq5pk