Data Structures and Algorithms for Engineers

Module 6: Trees

Lecture 6: Priority queues, binary heap, heapsort.

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Priority Queues

- Many applications require algorithms to process items in a specific order (e.g., relative importance).
  - One option: Use a list, sort it, and process in the resultant order
- Priority queues are more flexible
  - They allow new elements to be added at arbitrary intervals
  - More efficient to add the new element in a priority queue than to add to a list and re-sort

Priority Queues

Main priority queue operations

– *Insert (Q, x)*

  Given an item x with a key k, insert it into the priority queue Q

– Find_Minimum(Q) or Find_Maximum(Q)

  Return a pointer to the item whose key value is smaller / larger than any other key in the priority queue

– Delete_Minimum(Q) or DeleteMaximum(Q)

  Remove the item from the priority queue Q *whose key is minimum or maximum*

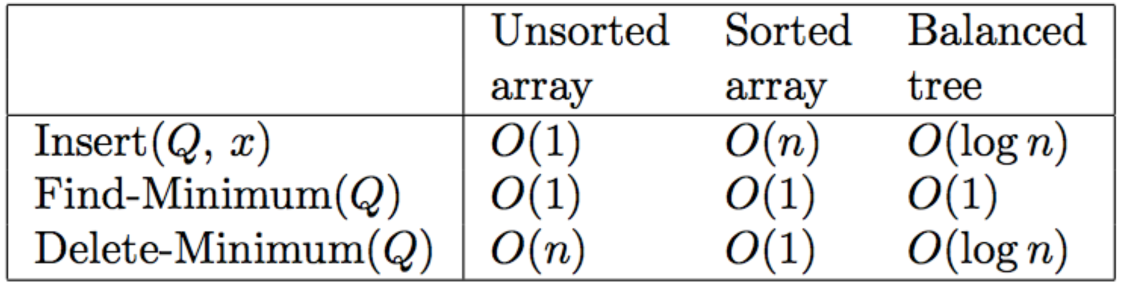
Priority Queues

Possible implementations

- Unsorted array
- Sorted array
- Balanced binary search tree

Priority Queues

Possible implementations



	Unsorted array	Sorted array	Balanced tree
Insert(Q, x)	$O(1)$	$O(n)$	$O(\log n)$
Find-Minimum(Q)	$O(1)$	$O(1)$	$O(1)$
Delete-Minimum(Q)	$O(n)$	$O(1)$	$O(\log n)$

- How do we get $O(1)$ for Find-Minimum(Q) in all three cases?
- Use a variable to store a pointer/index to the minimum entry in each of these structures
  - Update the pointer on each insertion (if necessary)
  - Update it on each deletion, requiring the new minimum to be found (but the cost of this can be folded into the cost of the deletion because it is the same cost as the deletion)

Binary Heap

- Favoured implementation of priority queue
- Heaps maintain partial order on the set of elements
  - Weaker than sorted order (& so it is efficient)
  - Stronger than random order (& so min/max element can be quickly identified)
  - "Heap" refers to being "top of the heap", i.e., what's on top dominates what is underneath
    - Dominate: is greater than (or less than) everything under it

Binary Heap

Heap-labelled tree is a binary tree

- Keys in each node dominate the keys of each of its children
- Min-heap … less than its children
- Max-heap … greater than its children

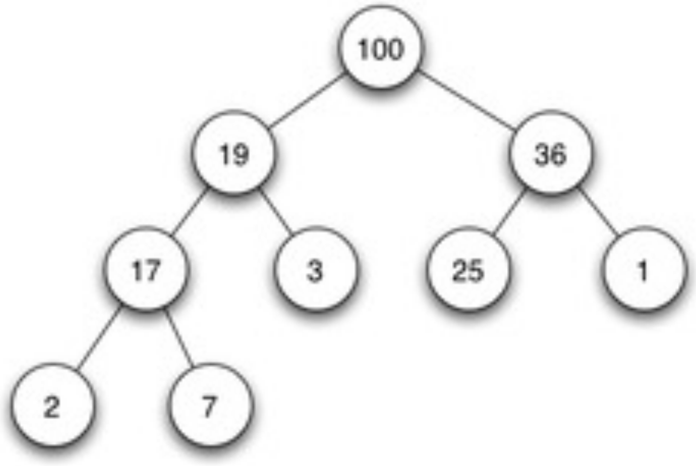
Binary Heap

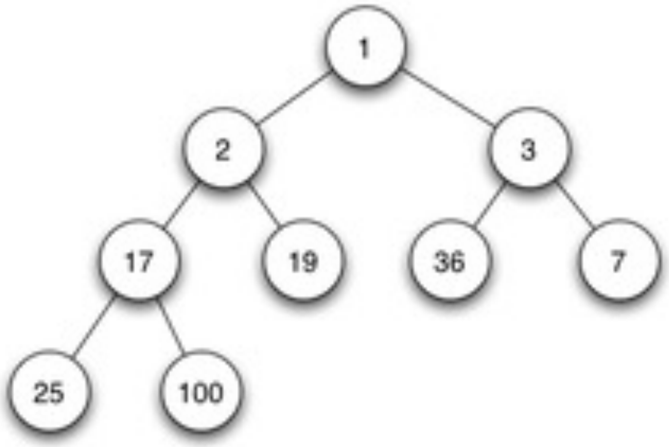
A binary heap is a binary tree that satisfies two special shape and heap properties:

- all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right

- each node is "greater than or equal to" each of its children (in the case of a max-heap) according to some comparison predicate which is fixed for the entire data structure

Binary Heap





Max-heap

Min-heap

Binary Heap

The order of siblings is not specified

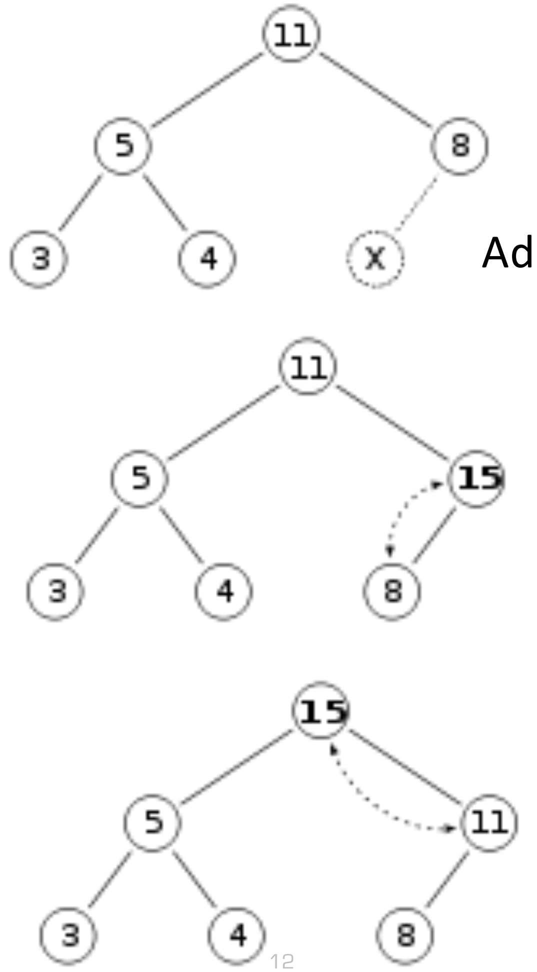
- Two children can be freely interchanged
- As long as it doesn't violate the shape and heap properties

Binary Heap

Adding to a heap

- Algorithm: upheap / heapify-up / sift-up
    - Add element to bottom level
    - Compare the added element with its parent;
      if they are in correct order, stop
    - If not, swap the element with its parent and return to previous step
- O(log n)

Binary Heap



Add 15 to max-heap

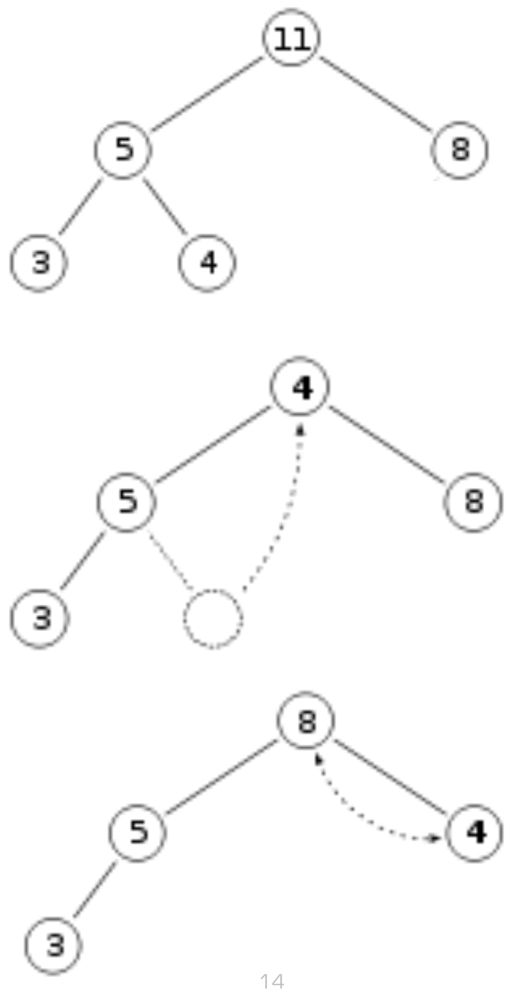
Binary Heap

Deleting the root from a heap

- Effectively extracting the maximum element in a max-heap (or extracting the minimum element in min-heap)
- Algorithm: downheap / heapify-down / sift-down
  - Replace root with last element on the bottom level
  - Compare the swapped element with
    - The larger child (max-heap)
    - The smaller child (min-heap)
  - if they are in correct order, stop
  - If not, swap the element with the child and return to previous step
- O(log n)

Binary Heap

Delete Root from max-heap



Binary Heap

Implementation as a binary tree data structure

Problem: finding adjacent element on last level

- Find it algorithmically (takes time)
- Use additional links between siblings:
  threading the tree (takes space)

Binary Heap

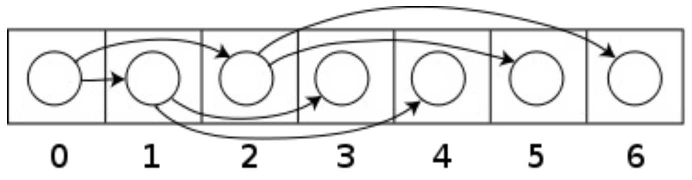
Implementation as an array

- Represent a binary tree WITHOUT any pointers by using an array of keys and a mapping function

- Use formula to find parents and children of a node

  - Node at index i has children at indices $2i+1$ and $2i+2$

  - Node at index i has parent at index $(i-1)/2$



Binary Heap

```
/* Implementation of a min-heap priority queue                       */
/*                                                                    */
/* To simplify matters, assume the array starts at index 1            */
/* Thus, left child of key at index k is located at index 2k          */
/*  and right child is located at index 2k+1                          */
/*  and parent of key at index k is located at index floor(k/2)       */
/*                                                                    */
/* Exercise: rewrite the code so that array starts at index 0         */


typedef struct {
   item_type q[PQ_SIZE+1];       /* body of queue */
   int n;                        /* number of queue elements */
} priority_queue;
```

Binary Heap

```
pq_parent(int n){
   if (n == 1) return(-1);
   else return((int) n/2);       /* implicitly take floor(n/2) */
}

pq_young_child(int n)
{
        return(2 * n);
}
```

Binary Heap

```
/* Construct the heap incrementally                                    */
/*                                                                     */
/* insert a new element at the left-most open location in the array */
/* i.e. at index (n+1) of a previously n-element heap.                 */
/*                                                                     */
/* This guarantees the desired balance but not necessarily the       */
/* dominance ordering of the keys: new key might be less than its    */
/* parent in a min-heap or greater than its parent in a max-heap     */
/*                                                                     */
/* To satisfy this condition, we recursively bubble up the new key  */
/* to its correct position (i.e. upheap / heapify-up / sift-up)      */
/*                                                                     */
/* Since the heap binary tree is almost full, its height is log n    */
/* and consequently each insertion takes at most O(log n) time       */
/* Thus and initial heap of n elements can be constructed in         */
/* O(n log n) time through n insertions                               */
```

Binary Heap

```
pq_insert(priority_queue *q, item_type x){
  if (q->n >= PQ_SIZE)
         printf("Warning: priority queue overflow insert x=%d\n",x);
  else {
         q->n = (q->n) + 1;
         q->q[ q->n ] = x;
         bubble_up(q, q->n);
   }
}

bubble_up(priority_queue *q, int p) {
    if (pq_parent(p) == -1) return; /* at root of heap, no parent */
    if (q->q[pq_parent(p)] > q->q[p]) {
            pq_swap(q,p,pq_parent(p));
            bubble_up(q, pq_parent(p));
   }
}
```

Binary Heap

```
pq_init(priority_queue *q) {
   q->n = 0;
}

make_heap(priority_queue *q, item_type s[], int n)
   int i; /* counter */

   pq_init(q);

   for (i=0; i<n; i++)
      pq_insert(q, s[i]);
}
```

Binary Heap

```
/* Extract the minimum                                                   */
/*                                                                       */
/* Identify and delete the dominant element:                             */
/* i.e. the element at the root of the tree, i.e the first element      */
/*                                                                       */
/* Deleting this root element leaves a hole in the array                 */
/* which we fill by moving the element from the right-most leaf         */
/* (at the nth position of the array) into the first position           */
/*                                                                       */
/* Again, this satisfies the shape condition but not necessarily        */
/* the dominance condition.                                              */
/* So, we recursively bubble down the new root                           */
/* to its correct position (i.e. heapify)                                */
/*                                                                       */
/* Root deletion is achieved in O(log n) time                            */
```

Binary Heap

```
item_type extract_min(priority_queue *q) {

   int min = -1;                     /* minimum value */

   if (q->n <= 0)
      printf("Warning: empty priority queue.\n");
   else {
        min = q->q[1];

        q->q[1] = q->q[ q->n ];
        q->n = q->n - 1;
        bubble_down(q,1);
   }

   return(min);
}
```

Binary Heap

```
bubble_down(priority_queue *q, int p) {

   int c;         /* child index              */
   int i;         /* counter                  */
   int min_index; /* index of lightest child */

   c = pq_young_child(p);

   min_index = p;

   /* find the index of the smallest of parent and children */
   for (i=0; i<=1; i++)
      if ((c+i) <= q->n) { // check that the child is there
         if (q->q[min_index] > q->q[c+i])
            min_index = c+i;
       }

   if (min_index != p) {          // if the smallest of the three nodes
       pq_swap(q,p,min_index);    // is NOT the parent, we need to swap
       bubble_down(q, min_index);// and repeat with that child
   }
}
```

Heapsort

- The heap priority queue can be used to create a very efficient sorting algorithm: heapsort
- Construct the priority queue: $O(n \log n)$
- Repeatedly extract the minimum: $O(n \log n)$
- Overall complexity is $O(n \log n)$ worst-case
- This is the best that can be expected from any sorting algorithm
- Also, it is an in-place sort, meaning it uses no extra memory in addition the array containing the elements is to be sorted

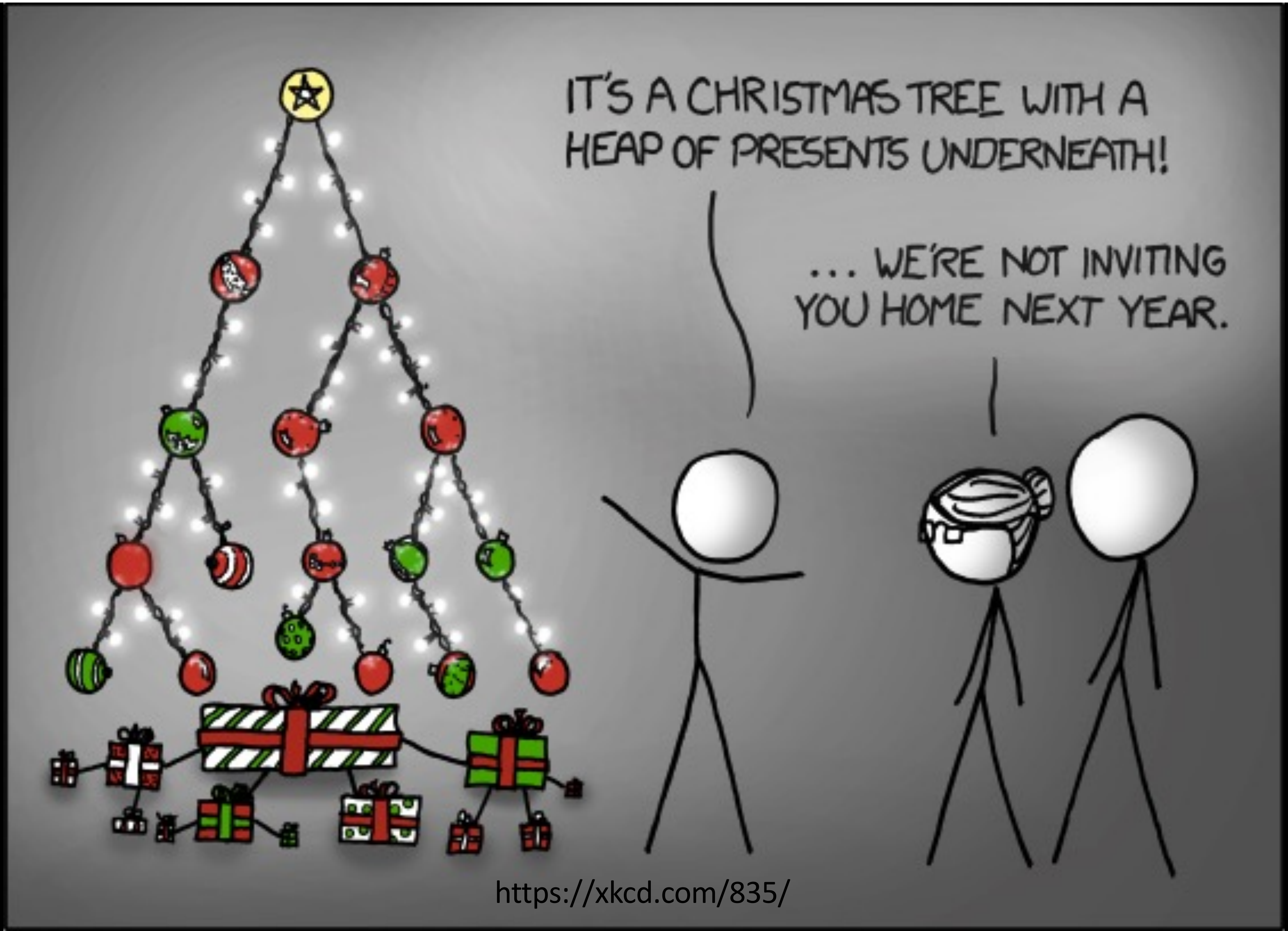
Heapsort

```
heapsort(item_type s[], int n) {

   int i;            /* counters */
   priority_queue q; /* heap for heapsort */

   make_heap(&q,s,n);

   for (i=0; i<n; i++)
      s[i] = extract_min(&q);
}
```


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

https://xkcd.com/835/