

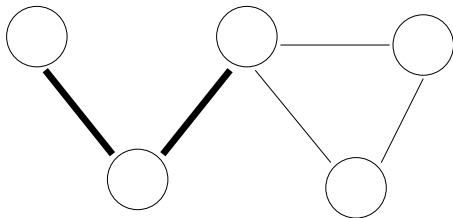
# CSC 263 Lecture 9

July 27, 2004

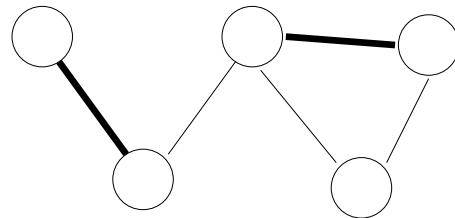
For another reference, see chapters 23 and 6 of the textbook (CLRS).

## 1 Minimum Cost Spanning Trees (MCSTs)

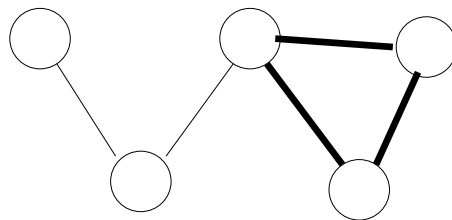
Let  $G = (V, E)$  be a connected, undirected graph with edge weights  $w(e)$  for each edge  $e \in E$ . A *tree* is a subset of edges  $A \subset E$  such that  $A$  is connected and contains no cycles. The following diagram shows a graph with three different subsets  $A$  (the thick edges are in  $A$ , the thin ones are not). One is a tree and the other two aren't.



Tree

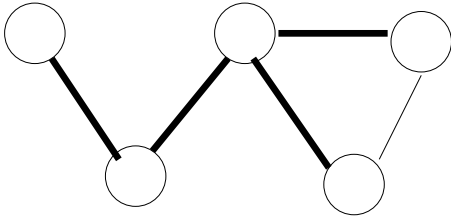


Not a tree (not connected)



Not a tree (has a cycle)

A *spanning tree* is a tree  $A$  such that every vertex  $v \in V$  is an endpoint of at least one edge in  $A$ . Notice that any spanning tree must contain  $n - 1$  edges, where  $|V| = n$  (proof by induction on  $n$ ).

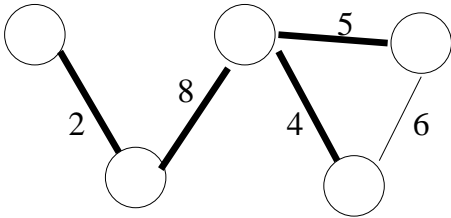


Spanning Tree

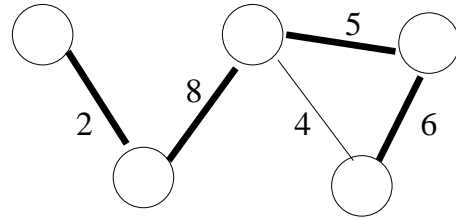
A *minimum cost spanning tree* is a spanning tree  $A$  such that

$$w(A) = \sum_{e \in A} w(e)$$

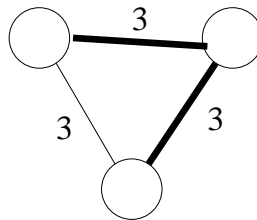
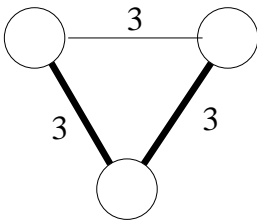
is less than or equal to  $w(B)$ , for all other spanning trees  $B$ .



Minimum Cost Spanning Tree



Spanning tree, but not minimum cost



Two different MCSTs on the same graph

Finding MCSTs is important in practice: imagine you have a network of computers that are connected by various links. Some of these links are faster, or more reliable, than others. You might want to pick a minimal set of links that connects every computer (in other words, a spanning tree) such that these links are overall the best (they have minimum cost). Once you have found these links, you never have to use the remaining slower, or less reliable, links.

We will look at two algorithms for constructing MCSTs. The first is Prim's Algorithm.

### 1.1 Prim's Algorithm

Prim's algorithm uses a Priority Queue ADT. This operates on a set  $S$  where each element  $x \in S$  has a priority  $p(x)$  which comes from a well-ordered universe (usually the natural numbers). There are three operations on this set:

- INSERT( $S, x$ ): insert an element  $x$  in the set  $S$ .
- ISEMPY( $S$ ): return true if  $S$  is empty.
- EXTRACT-MIN( $S$ ): remove and return an element  $x \in S$  with minimum priority.

In addition, we will need the operation DECREASE-PRIORITY( $x, p$ ) will set the priority of  $x$ , which is in the queue, to be  $p$  ( $p$  is less than  $x$ 's current priority).

```

PRIM-MST( $G=(V,E), w:E \rightarrow Z$ )
   $A := \{\}$ ;
  initialize a priority queue  $Q$ ;
  for all  $v$  in  $V$  do
    priority[ $v$ ] := infinity;
     $p[v] := \text{NIL}$ ;
    INSERT( $Q, v$ );
  end for
  pick some arbitrary vertex  $s$  in  $V$  and let priority[ $s$ ] := 0;
  while ( not ISEMPY( $Q$ ) ) do
     $u := \text{EXTRACT-MIN}(Q)$ ;
    if  $p[u] \neq \text{NIL}$  then  $A := A \cup \{(p[u], u)\}$ ;
    for each  $v$  in adjacency-list[ $u$ ] do
      if  $v$  in  $Q$  and  $w(u, v) < \text{priority}[v]$  then
        DECREASE-PRIORITY( $v, w(u, v)$ );
         $p[v] := u$ ;
      end if
    end for
  end while
END PRIM-MST

```

Prim's algorithm also grows an MCST  $A$  starting with an empty set. Even though  $A$  is technically a set of edges, we view it as a set of vertices too. In general, these vertices will be the endpoints of the edges in  $A$ . When we start the algorithm, however, we consider the vertex  $s$  to be in  $A$  even though there are no edges in  $A$ . From now on, we just keep adding edges to  $A$  by finding the "lightest" edge that has one endpoint in  $A$  and the other endpoint outside of  $A$ .

## 1.2 Correctness

The correctness of Prim's algorithm is given by the following theorem.

**Theorem:** If  $G = (V, E)$  is a connected, undirected, weighted graph,  $A$  is a subset of some MCST of  $G$ , and  $e$  is any edge of minimum weight with one endpoint in  $A$  and one endpoint outside of  $A$ , then  $A \cup \{e\}$  is a subset of some MCST of  $G$ .

**Proof:** Let  $T$  be a MCST of  $G$  that contains  $A$  as a subset. If  $e$  is in  $T$ , then we are done. Otherwise, we construct a different MCST  $T'$  that contains  $e$ . If we add edge  $e$  to  $T$ , we create a cycle in the resulting graph ( $T \cup \{e\}$  is not a tree anymore). This cycle must contain another edge  $e'$  with one endpoint in  $A$  and one endpoint outside of  $A$  (otherwise, every endpoint of  $A$  is already in  $T$  which means that edge  $e$  cannot exist). Since we picked  $e$  to have minimum weight among such edges, it must be the case that  $w(e) \leq w(e')$ . Now, let  $T' = T \cup \{e\} - \{e'\}$ .  $T'$  is connected, and it is acyclic (since we removed one edge from the only cycle in  $T \cup \{e\}$ ), so it is a spanning

tree of  $G$  that contains  $A$  as a subset. Moreover,  $w(T') - w(T) = w(e) - w(e') \leq 0$  so  $T'$  has total weight no greater than  $T$ , i.e.,  $T'$  is an MCST that contains  $A \cup \{e\}$  as a subset.

## 2 Priority Queues

Priority queues are very useful. Some of their applications are:

- Job scheduling in operating systems
- Printer queues
- Event-driven simulation algorithms
- Greedy algorithms

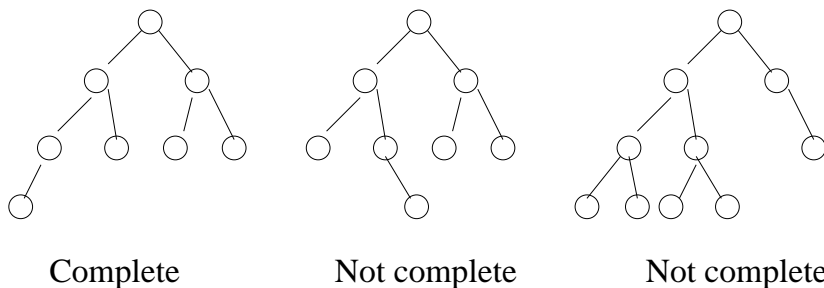
There are several possible data structures for implementing priority queues:

- Unsorted list: takes time  $\Theta(n)$  for **EXTRACT-MIN** in the worst-case.
- Sorted list (by priorities): takes time  $\Theta(n)$  for **INSERT** in worst-case.
- Red-Black tree (key-values are priorities): **INSERT** and **EXTRACT-MIN** take time  $\Theta(\log n)$ .
- Direct addressing: if the universe  $U$  of priorities is small and the priorities are all distinct, then we can store an element with priority  $k$  in the  $k$ th cell of an array. **INSERT** takes time  $\Theta(1)$ . **EXTRACT-MIN** requires time  $\Theta(|U|)$  in the worst-case (have to look at each location to find the first nonempty one).

## 3 Heaps

We will look at one particular data structure for priority queues in depth. They are called *heaps* and are defined as follows: a heap is a binary tree  $T$  of elements with priorities such that

1.  $T$  is *complete*: this means that every level of the tree is full except perhaps the bottom one, which fills up from left to right. For example:

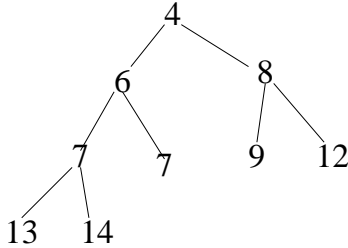


2. For each node  $x$  in  $T$ , if  $x$  has a left-child, then  $p(x) \leq p(\text{left}(x))$  and if  $x$  has a right-child, then  $p(x) \leq p(\text{right}(x))$ .

We can conclude a few immediate facts about heaps from the definition. First of all, the root has minimum priority. Secondly, every subtree of a heap is also a heap (in particular, an empty tree is a heap). Finally, since heaps are complete, if a heap contains  $n$  nodes, then its height  $h$  is  $\Theta(\log n)$ .

### 3.1 Storing heaps

Traditionally, a heap is stored by using an array  $A$  together with an integer *heapsize* that stores the number of elements currently in the heap (or the number of nonempty entries in  $A$ ). The following conventions are used to store the nodes of the tree in the array: the root of the tree is stored at  $A[1]$ , the two children of the root are stored at  $A[2]$  and  $A[3]$ , the four grandchildren of the root are stored at  $A[4]$ ,  $A[5]$ ,  $A[6]$ ,  $A[7]$ , etc. In general, if element  $x$  is stored at  $A[i]$ , then  $left(x)$  is stored at  $A[2i]$  and  $right(x)$  is stored at  $A[2i + 1]$ .



$A = [4,6,8,7,7,9,12,13,14]$

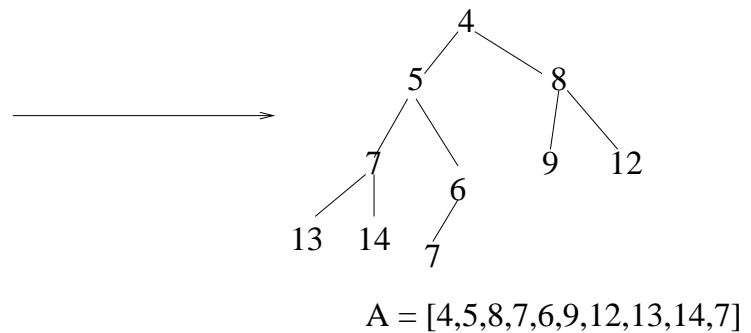
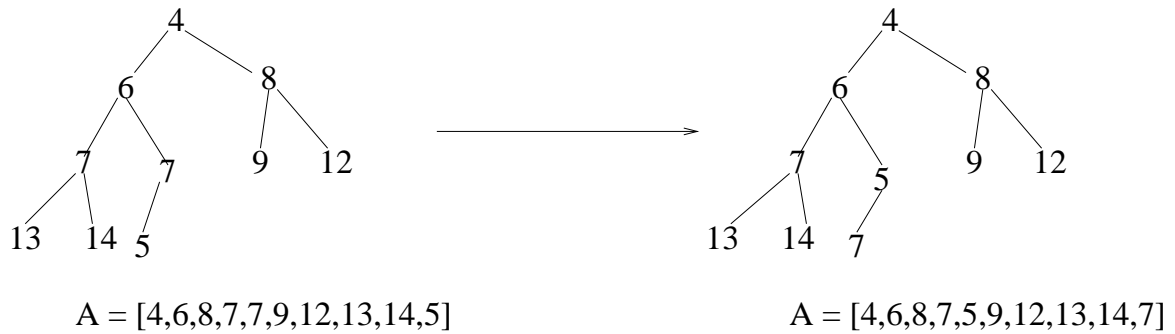
If the size of the array is close to the number of elements in the heap, then this data structure is extremely space-efficient because we don't have to store any pointers. We can use a dynamic array to ensure that this is true (recall that the amortized cost of managing a dynamic array is small).

### 3.2 Implementing priority queues

We can perform the priority queue operations on a heap as follows:

- **INSERT:** Increment *heapsize* and add the new element at the end of the array. The result might violate the heap property, so "percolate" the element up (exchanging it with its parent) until its priority is no smaller than the priority of its parent.

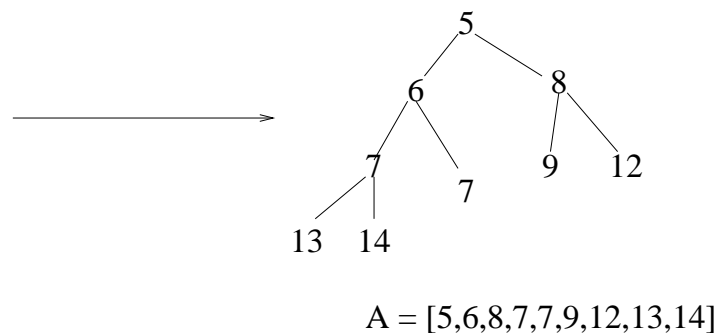
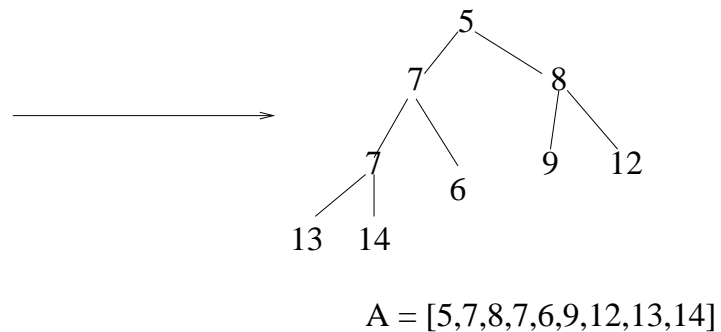
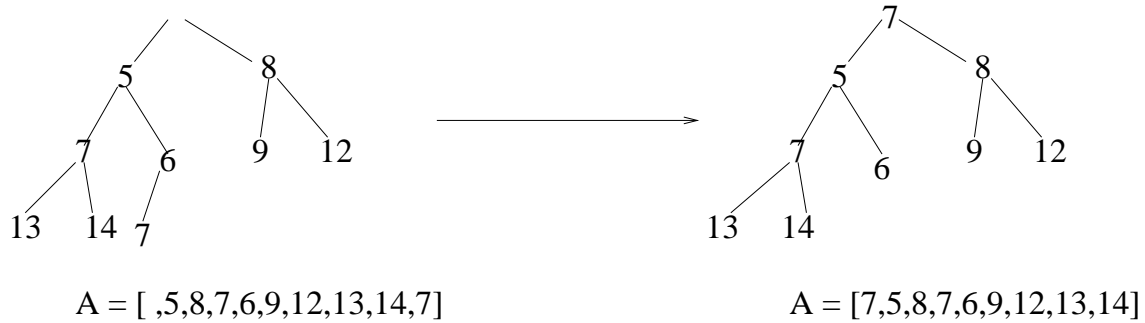
For example, if we perform **INSERT**(5) on the previous heap, we get the following result (showing both the tree and the array for each step of the operation):



In the worst-case, we will have to move the new element all the way to the root, which takes time  $\Theta(\text{height of heap}) = \Theta(\log n)$ .

- **EXTRACT-MIN:** Decrement *heapsize* and remove the first element of the array. In order to be left with a valid heap, move the last element in the array to the first position (so the heap now has the right "shape"), and percolate this element down until its priority is no greater than the priorities of both its children. Do this by exchanging the element with its child of lowest priority at every step.

For example, if we perform **EXTRACT-MIN** on the previous heap, we get the following result (showing both the tree and the array for each step of the operation):



As with INSERT, we may wind up moving the last element from the root all the way down to a leaf, which takes  $\Theta(\text{height of heap}) = \Theta(\log n)$  in the worst-case.

The "percolating down" of an element that we just described for EXTRACT-MIN is a very useful operation for heaps. In fact, it's so useful that it already has a name: If  $x$  is the element initially stored at  $A[i]$ , and assuming that the left and right subtrees of  $x$  are heaps. Then HEAPIFY( $A, i$ ) percolates  $x$  downwards until the subtree of the element now stored at  $A[i]$  is a heap.

```

HEAPIFY(A, i)
  smallest := i;
  if ( 2i <= heapsize and A[2i] < A[i] ) then

```

```

    smallest := 2i;
endif
if ( 2i+1 <= heapsize and A[2i+1] < A[smallest] ) then
    smallest := 2i+1;
endif
if ( smallest /= i ) then
    swap A[i] and A[smallest];
    HEAPIFY(A,smallest);
endif
END

```

The running time, as with EXTRACT-MIN, is  $\Theta(\log n)$ .

### 3.3 Building heaps

If we start with an array  $A$  of elements with priorities, whose only empty slots are at the far right, then we can immediately view  $A$  as a complete binary tree.  $A$ , however, is not necessarily a heap unless the elements are ordered in a certain way. There are several options for making  $A$  into a heap:

1. Sort  $A$  from lowest priority element to highest. Clearly  $A$  will now obey part 2 of the heap definition (actually, every sorted array is a heap, but every heap is not necessarily a sorted array). This takes time  $\Theta(n \log n)$  if we use, say, the guaranteed fast version of quicksort.
2. We can simply make a new array  $B$  and go through every element of  $A$  and INSERT it into  $B$ . Since INSERT takes time  $\Theta(\log n)$  and we do it for each of the  $n$  elements of  $A$ , the whole thing takes time  $\Theta(n \log n)$ .
3. The most efficient way is to use HEAPIFY: notice that every item in the second half of  $A$  corresponds to a leaf in the tree represented by  $A$ , so starting at the "middle" element (i.e., the first nonleaf node in the tree represented by  $A$ ), we simply call HEAPIFY on each position of the array, working back towards position 1.

```

BUILD-HEAP(A)
    heapsize := size(A);
    for i := floor(heapsize/2) downto 1 do
        HEAPIFY(A,i);
    end for
END

```

Because each item in the second half of the array is already a heap (it's a leaf), the preconditions for HEAPIFY are always satisfied before each call. For example, if  $A = [1,5,7,6,2,9,4,8]$ , then BUILD-HEAP(A) makes the following sequence of calls to HEAPIFY (you can check the result of each one by tracing it):

```

HEAPIFY( [1,5,7,6,2,9,4,8], 4 ) = [1,5,7,6,2,9,4,8]
HEAPIFY( [1,5,7,6,2,9,4,8], 3 ) = [1,5,4,6,2,9,7,8]
HEAPIFY( [1,5,4,6,2,9,7,8], 2 ) = [1,2,4,6,5,9,7,8]
HEAPIFY( [1,2,4,6,5,9,7,8], 1 ) = [1,2,4,6,5,9,7,8]

```

Since we make  $O(n)$  calls to `HEAPIFY` and since each one takes  $O(\log n)$  time, we immediately get a bound of  $O(n \log n)$ . But in fact, we can do better by analyzing more carefully: basically, we call `HEAPIFY` on each subtree of height  $\geq 1$  and `HEAPIFY` runs in time proportional to the height of that subtree. So we can estimate the total running time as follows:

$$O\left(\sum_{h=1}^{\log n} h \times \text{number of subtrees of height } h\right).$$

The sum goes to  $\log n$  because the height of the whole tree is  $\Theta(\log n)$ . A tree with  $n$  nodes contains at most  $\lceil n/2^{h+1} \rceil$  nodes of height  $h$  (why?), so it contains at most the same number of subtrees of height  $h$ . Therefore, the running time is:

$$O\left(\sum_{h=1}^{\log n} h \times \lceil n/2^{h+1} \rceil\right) = O\left(n \sum_{h=1}^{\infty} h/2^h\right) = O(n).$$

The last equation comes from the fact that  $\sum_{h=1}^{\infty} h/2^h \leq 2$  (p. 44 of CLR). So `BUILD-HEAP` runs in time  $O(n)$ .

### 3.4 Complexity of Prim's Algorithm

So what is the running time of Prim's Algorithm? It turns out that using the above ideas, you can implement `DECREASE-PRIORITY` in time  $O(\log n)$ . The while loop of Prim's runs at most once for every vertex in the graph. If  $u$  is the current vertex selected from the priority queue, then the algorithm analyzes each edge going from  $u$  to outside  $A$ . Since  $u$  is always added to  $A$  and never becomes the current vertex again, we consider each edge at most once. When we consider an edge, we might decrease the priority of one of its endpoints, which takes time  $O(\log n)$ . Therefore, the loop takes time at most  $O(m \log n)$ . Since the initial building of the heap can be done faster than this, the worst-case running time is  $O(m \log n)$ .