Data Structures for Minimum Spanning Trees: Graphs & Heaps

Slides by Carl Kingsford

Jan. 17, 2014

KT 2.5,3.1
Recall Prim’s Algorithm

1: \# distToT[u] is distance from current tree to u
2: for \( u \in V \) do distToT[u] \( \leftarrow \infty \)
3: \( u \leftarrow s \)
4: while \( u \neq \text{null} \) do
5: \# We put \( u \) in the tree, so distance is \(-\infty\)
6: distToT[u] \( \leftarrow -\infty \)
7: \# Each of \( u \)'s neighbors \( v \) are now incident to the current tree
8: for \( v \in \text{NEIGHBORS}(u) \) do
9: \# If the distance is smaller than before, we have to update
10: if \( d(u,v) < \text{distToT}[v] \) then
11: distToT[v] \( \leftarrow d(u,v) \)
12: parent[v] \( \leftarrow u \)
13: \( u \leftarrow \text{CLOSESTVERTEX}(\text{distToT}) \)
14: return parent
RAM = Symbols + Pointers
(for our purposes)

RAM is an array of bits, broken up into words, where each word has an address.

We may interpret the bits as numbers, or letters, or other symbols.

We may interpret bits as a memory address (pointer).

We can store and manipulate arbitrary symbols (like letters) and associations between them.
Storing a list

Store a list of numbers such as 3, 7, 32, 8:

- Records located anywhere in memory
- Don’t have to know the size of data at the start
- Pointers let us express relationships between pieces of information
What is a data structure anyway?

It’s an agreement about:

- how to store a collection of objects in memory,
- what operations we can perform on that data,
- the algorithms for those operations, and
- how time and space efficient those algorithms are.

Data structures → Data structuring:

How do we organize information in memory so that we can find, update, add, and delete portions of it efficiently?

Often, the key to a fast algorithm is an efficient data structure.
Abstract data types (ADT)

ADT specifies permitted operations as well as time and space guarantees.

Example Graph ADT (without time/space guarantees):

- `G.add_vertex(u)` — adds a vertex to graph G.
- `G.add_edge(u, v, d)` — adds an edge of weight d between vertices u and v.
- `G.has_edge(u, v)` — returns True iff edge \{u,v\} exists in G.
- `G.neighbors(u)` — gives a list of vertices adjacent to u in G.
- `G.weight(u,v)` — gives the weight of edge u and v
Representing Graphs

Adjacency matrix:

\[
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 \\
\end{pmatrix}
\]

Adjacency list:
Representing Graphs

Adjacency matrix:

\[
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0
\end{pmatrix}
\]

How long does \texttt{G.neighbors(u)} take?

Adjacency list:

1 → 2 → 3
2 → 1 → 5
3 → 1 → 4 → 5
4 → 3
5 → 2 → 3
Representing Graphs

Adjacency matrix:

\[
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 \\
\end{pmatrix}
\]

How long does \( G.\text{neighbors}(u) \) take?

How would you implement \( G.\text{weight}(u,v) \)?
Implementing \texttt{ClosestVertex}:

Priority Queue ADT & d-Heaps
Priority Queue ADT

A priority queue, also called a heap, holds items $i$ that have keys $\text{key}(i)$.

They support the following operations:

- $H.\text{insert}(i)$ — put item $i$ into heap $H$
- $H.\text{deletemin}()$ — return item with smallest key in $H$ and delete it from $H$
- $H.\text{makeheap}(S)$ — create a heap from a set $S$ of items
- $H.\text{findmin}()$ — return item with smallest key in $H$
- $H.\text{delete}(i)$ — remove item $i$ from $H$
ClosestVertex via Heaps

Items: vertices that are on the “frontier” (incident to the current tree)

Key(v) = distToT[v]

ClosestVertex: return H.deletemin()
Priority Queue ADT

- Efficiently support the following operations on a set of keys:
  - findmin: return the smallest key
  - deletemin: return the smallest key & delete it
  - insert: add a new key to the set
  - delete: delete an arbitrary key

- Would like to be able to do findmin faster (say in time independent of the # of items in the set).
When scheduler asks “What should I run next?” it could $\text{findmin}(H)$. 
Plane Sweep: Process points left to right:

Store points in a priority queue, ordered by their $x$ coordinate.
Plane Sweep: Process points left to right:

Store points in a priority queue, ordered by their $x$ coordinate.
Plane Sweep: Process points left to right:

Store points in a priority queue, ordered by their $x$ coordinate.
Plane Sweep: Process points left to right:

Store points in a priority queue, ordered by their $x$ coordinate.
Plane Sweep: Process points left to right:

Store points in a priority queue, ordered by their $x$ coordinate.
Heap-Ordered Trees

- The keys of the children of $u$ are $\geq$ the key($u$), for all nodes $u$.
- (This “heap” has nothing to do with the “heap” part of computer memory.)
Heap-Ordered Trees

- The keys of the children of $u$ are $\geq$ the key($u$), for all nodes $u$.
- (This “heap” has nothing to do with the “heap” part of computer memory.)

Along each path, keys are monotonically non-decreasing.
Heap – Find min

The minimum element is always the root
Heap – Insert

```
2
/  \
8   3
/   /  \
12  9   7
       \\
        10
```
1. Add node as a leaf
   (we’ll see where later)
Heap – Insert

1. Add node as a leaf (we’ll see where later)

2. “sift up:” while current node is < its parent, swap them.
Heap – Insert

1. Add node as a leaf (we’ll see where later)

2. “sift up:” while current node is < its parent, swap them.
Heap – Insert

1. Add node as a leaf
   (we’ll see where later)

2. “sift up:” while current node is < its parent, swap them.
Heap – Delete($i$)
Heap – Delete($i$)

1. need a pointer to node containing key $i$
Heap – Delete($i$)

1. need a pointer to node containing key $i$

2. replace key to delete $i$ with key $j$ at a leaf node
   (we’ll see how to find a leaf soon)
Heap – Delete($i$)

1. need a pointer to node containing key $i$

2. replace key to delete $i$ with key $j$ at a leaf node (we’ll see how to find a leaf soon)

3. Delete leaf
Heap – Delete(\(i\))

1. need a pointer to node containing key \(i\)

2. replace key to delete \(i\) with key \(j\) at a leaf node (we’ll see how to find a leaf soon)

3. Delete leaf

4. If \(i > j\) then sift up, moving \(j\) up the tree.

If \(i < j\) then “sift down”: swap current node with smallest of children until its bigger than all of its children.
Running Times

• \textit{findmin} takes constant time \([O(1)]\)

• \textit{insert, delete} take time \(\propto\) tree height plus the time to find the leaves.

• \textit{deletemin}: same as delete

• Q1: How do we find leaves used in \textit{insert} and \textit{delete}?
  – \textit{delete}: use the last inserted node.
  – \textit{insert}: choose node so tree remains complete.

• Q2: How do we ensure the tree has low height?
Store Heap in a Complete Tree

```
    2
   /  
  8   3
 / 
12  9
/ 
15 21
```

```
Store Heap in a Complete Tree

```
        2
       / \
      8   3
     / \   /
    12 9  7 10
   / \  /   /
  15 21 A  7 10
```
Store Heap in a Complete Tree
Store Heap in a Complete Tree
Store Heap in a Complete Tree
Store Heap in a Complete Tree
Store Heap in a Complete Tree
Arrays ⇔ Complete Binary Trees
Store Heap in a Complete Tree

left(i): \(2i\) if \(2i \leq n\) otherwise None
right(i): \((2i + 1)\) if \(2i + 1 \leq n\) otherwise None
parent(i): \([i/2]\) if \(i \geq 2\) otherwise None
Store Heap in a Complete Tree

left(i): $2i$ if $2i \leq n$ otherwise None
right(i): $(2i + 1)$ if $2i + 1 \leq n$ otherwise None
parent(i): $\lfloor i/2 \rfloor$ if $i \geq 2$ otherwise None
Store Heap in a Complete Tree

left(i): \(2i\) if \(2i \leq n\) otherwise None

right(i): \((2i + 1)\) if \(2i + 1 \leq n\) otherwise None

parent(i): \([i/2]\) if \(i \geq 2\) otherwise None
Store Heap in a Complete Tree

left(i): $2i$ if $2i \leq n$ otherwise None
right(i): $(2i + 1)$ if $2i + 1 \leq n$ otherwise None
parent(i): $\lfloor i/2 \rfloor$ if $i \geq 2$ otherwise None
Store Heap in a Complete Tree

- left(i): $2i$ if $2i \leq n$ otherwise None
- right(i): $(2i + 1)$ if $2i + 1 \leq n$ otherwise None
- parent(i): $\lfloor i/2 \rfloor$ if $i \geq 2$ otherwise None
Store Heap in a Complete Tree

left(i): $2i$ if $2i \leq n$ otherwise None
right(i): $(2i + 1)$ if $2i + 1 \leq n$ otherwise None
parent(i): $\lfloor i/2 \rfloor$ if $i \geq 2$ otherwise None
Store Heap in a Complete Tree

left(i): \(2i \) if \(2i \leq n\) otherwise None
right(i): \((2i + 1)\) if \(2i + 1 \leq n\) otherwise None
parent(i): \(\lceil i/2 \rceil\) if \(i \geq 2\) otherwise None
**d-Heaps – Don’t have to use binary trees**

- What about complete non-binary trees (e.g. every node has $d$ children)?
  - *insert* takes $O(\log_d n)$ [because height $O(\log_d n)$]
  - *delete* takes $O(d \log_d n)$ [why?]

- Can still store in an array.

- If you have few deletions, make $d$ bigger so that tree is shorter.

- Can tune $d$ to fit the relative proportions of inserts / deletes.
d-Heap Runtime Summary

- *findmin* takes $O(1)$ time
- *insert* takes $O(\log_d n)$ time
- *delete* takes $O(d \log_d n)$ time
- *deletemin* takes time $O(d \log_d n)$
- *makeheap* takes $O(n)$ time

Reason: height of a complete binary tree with $n$ nodes is about $\log n$. 
One more operation needed for Prim’s

- H.decreaseKey($u,j$): reduce the key for item $u$ by $j$. 
One more operation needed for Prim’s

- H.decreaseKey(u,j): reduce the key for item u by j.

Why is this needed?
One more operation needed for Prim’s

• H.decreaseKey(u,j): reduce the key for item u by j.

Why is this needed?

How can we implement it?
One more operation needed for Prim’s

• H.decreaseKey(u,j): reduce the key for item u by j.

Why is this needed?

How can we implement it?

1. Reduce the key by j
2. siftup to put it in the right place.
3. Takes time proportional to the height of the tree $\approx \log n$
Make Heap – Create a heap from $n$ items

- $n$ inserts take time $\propto n \log n$.
- Better:
  - put items into array arbitrarily.
  - for $i = n \ldots 1$, siftdown($i$).
- Each element trickles down to its correct place.

By the time you sift level $i$, all levels $i + 1$ and greater are already heap ordered.
Make Heap – Time Bound

There are at most \( \frac{n}{2^h} \) items at height \( h \).

Siftdown for all height \( h \) nodes is \( \approx \frac{hn}{2^h} \) time.

Total time
\[
\approx \sum_h \frac{hn}{2^h} \quad [\text{sum of time for each height}]
\]
\[
= n \sum_h \frac{h}{2^h} \quad [\text{factor out the n}]
\]
\[
\approx n \quad [\text{sum bounded by const}]
\]
Heapsort – Another application of Heaps

Given unsorted array of integers

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>12</th>
<th>10</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Heapsort – Another application of Heaps

Given unsorted array of integers

```
8  2  12  10  7  15  21  9  3
1  2  3  4  5  6  7  8  9
```

makeheap – O(n)
Now first position has smallest item.

```
2  3  12  8  7  15  21  9  10
1  2  3  4  5  6  7  8  9
```

Swap first & last items.
Heapsort – Another application of Heaps

Given unsorted array of integers

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>12</td>
<td>10</td>
<td>7</td>
<td>15</td>
<td>21</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

makeheap – O(n)
Now first position has smallest item.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>21</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Swap first & last items.
Heapsort – Another application of Heaps

Given unsorted array of integers

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>12</th>
<th>10</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

makeheap – O(n)
Now first position has smallest item.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Delete last item from heap.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Heapsort – Another application of Heaps

Given unsorted array of integers

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>12</th>
<th>10</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

makeheap – O(n)

Now first position has smallest item.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Delete last item from heap.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Heapsort – Another application of Heaps

Given unsorted array of integers

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>12</td>
<td>10</td>
<td>7</td>
<td>15</td>
<td>21</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

makeheap – O(n)
Now first position has smallest item.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>21</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Delete last item from heap.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>21</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

siftdown new root key down
Heapsort – Another application of Heaps

Given unsorted array of integers

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>12</th>
<th>10</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

makeheap – O(n)

Now first position has smallest item.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Delete last item from heap.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

siftdown new root key down

<table>
<thead>
<tr>
<th>3</th>
<th>7</th>
<th>12</th>
<th>8</th>
<th>10</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Heapsort – Another application of Heaps

Given unsorted array of integers

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>12</th>
<th>10</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

makeheap – O(n)
Now first position has smallest item.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Delete last item from heap.

<table>
<thead>
<tr>
<th>10</th>
<th>3</th>
<th>12</th>
<th>8</th>
<th>7</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

siftdown new root key down

<table>
<thead>
<tr>
<th>3</th>
<th>7</th>
<th>12</th>
<th>8</th>
<th>10</th>
<th>15</th>
<th>21</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>