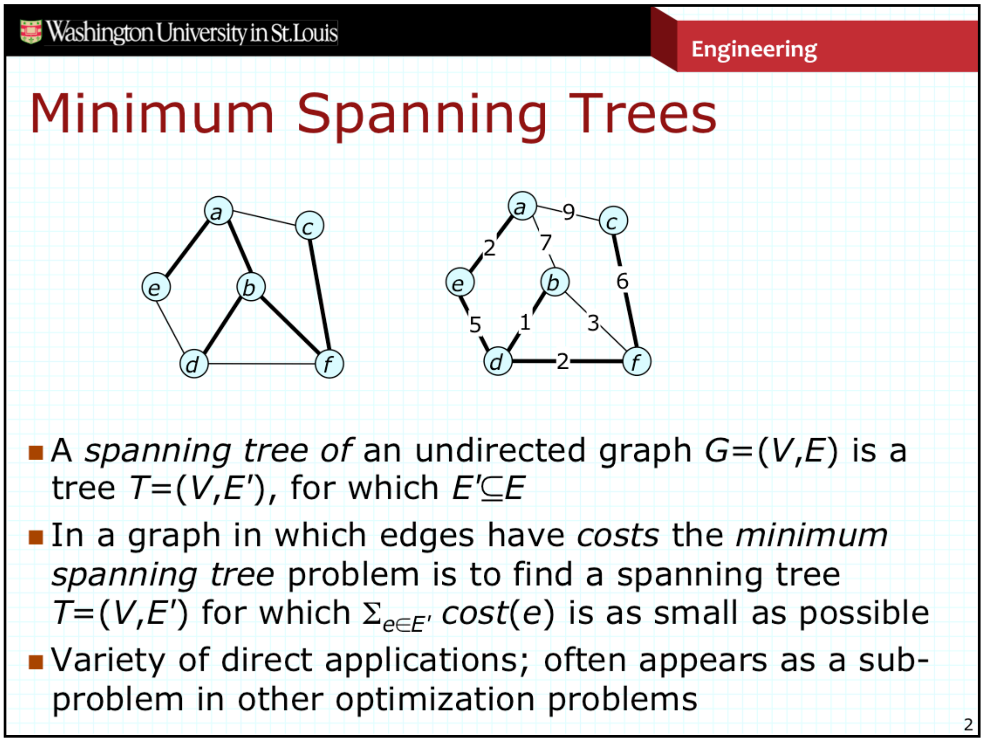
Washington University in St.Louis

Engineering

Minimum Spanning Trees and *d*-Heaps

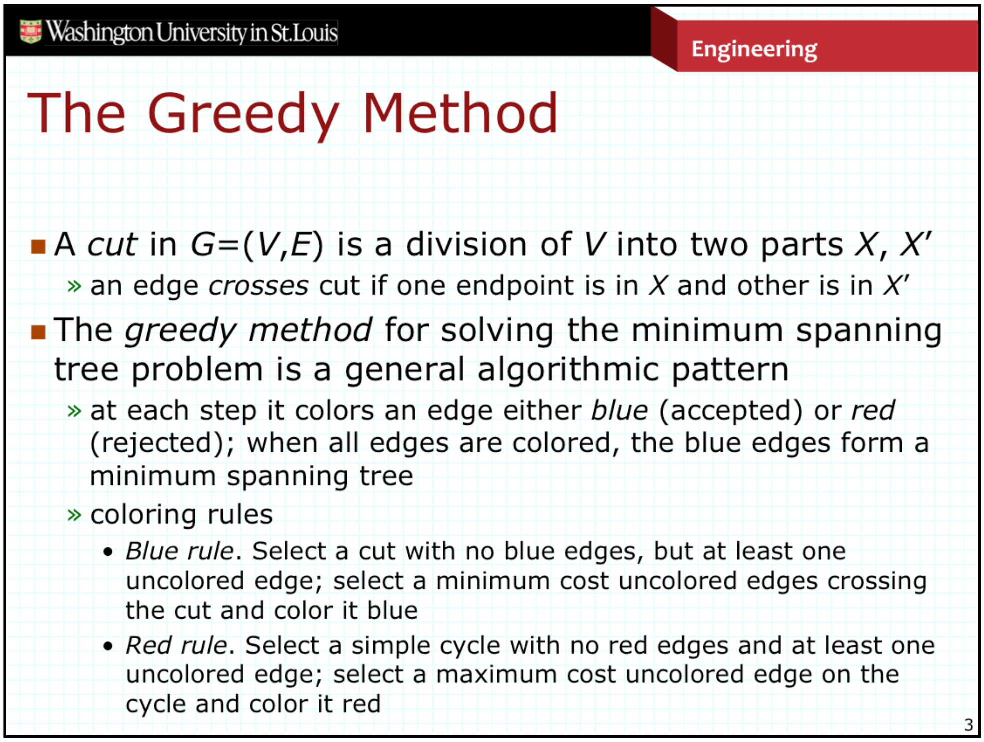
Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst



Minimum Spanning Trees

- A *spanning tree of* an undirected graph $G=(V,E)$ is a tree $T=(V,E')$, for which $E' \subseteq E$
- In a graph in which edges have *costs* the *minimum spanning tree* problem is to find a spanning tree $T=(V,E')$ for which $\Sigma_{e \in E'}\, cost(e)$ is as small as possible
- Variety of direct applications; often appears as a sub-problem in other optimization problems



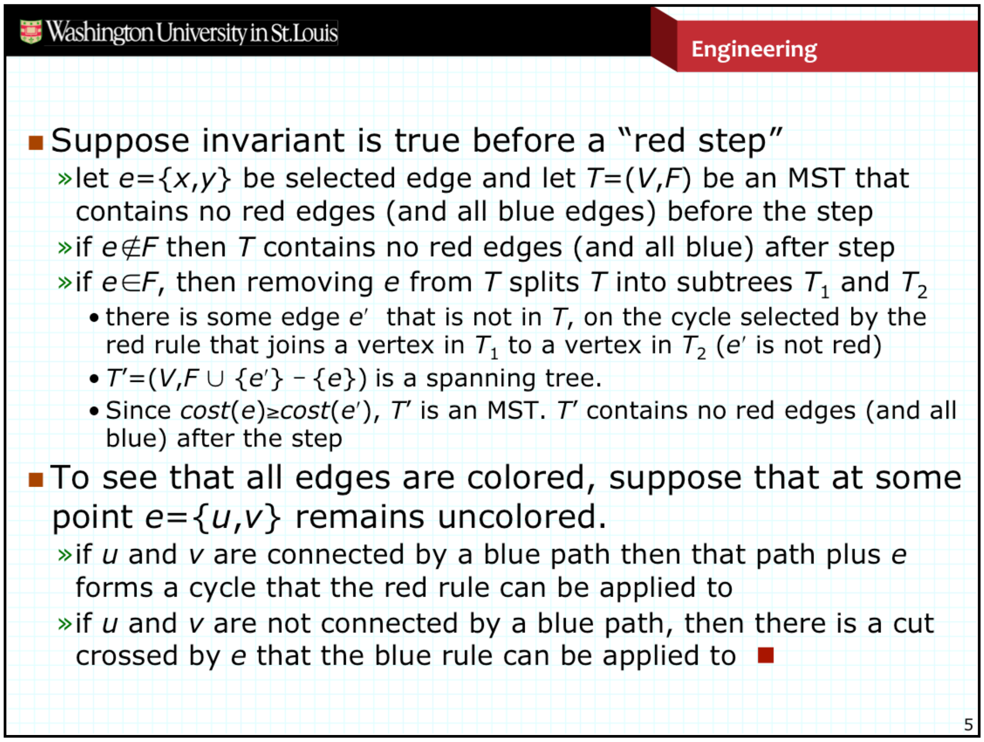
The Greedy Method

- A *cut* in *G*=(*V*,*E*) is a division of *V* into two parts *X*, *X′*
  - » an edge *crosses* cut if one endpoint is in *X* and other is in *X′*
- The *greedy method* for solving the minimum spanning tree problem is a general algorithmic pattern
  - » at each step it colors an edge either *blue* (accepted) or *red* (rejected); when all edges are colored, the blue edges form a minimum spanning tree
  - » coloring rules
    - *Blue rule*. Select a cut with no blue edges, but at least one uncolored edge; select a minimum cost uncolored edges crossing the cut and color it blue
    - *Red rule*. Select a simple cycle with no red edges and at least one uncolored edge; select a maximum cost uncolored edge on the cycle and color it red

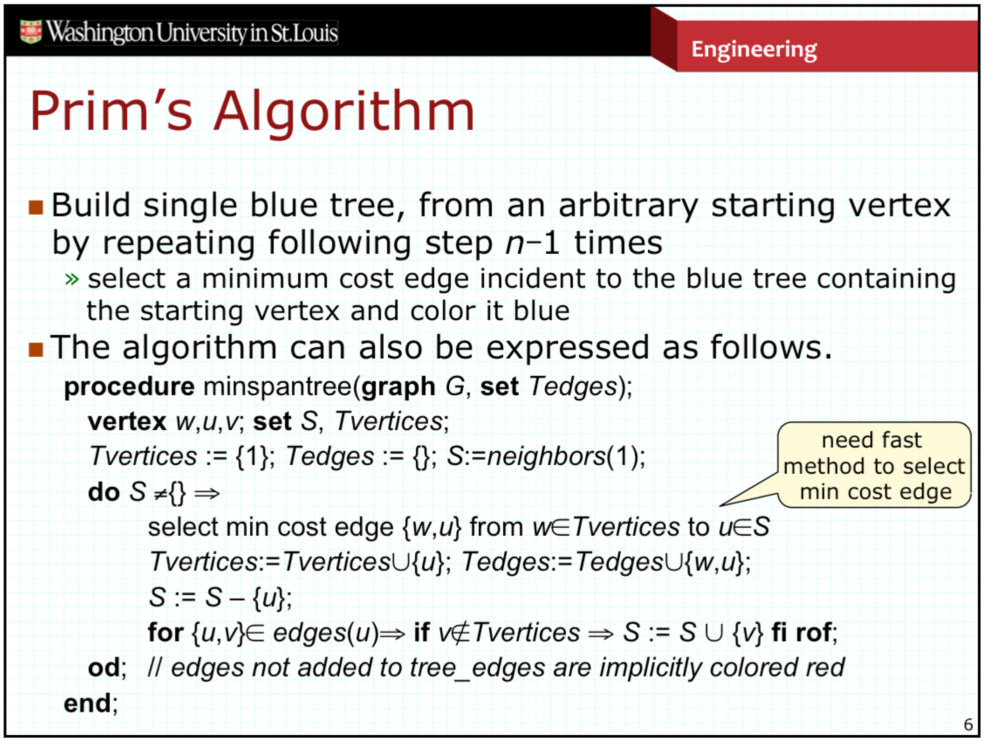
Correctness of Greedy Method

- Greedy method maintains *color invariant*
  - »there is an MST containing all the blue edges and no red ones
- *Theorem 6.1* (*Tarjan*)**.** Greedy method colors all edges of a connected graph and maintains color invariant

  Proof. Suppose invariant is true before a "blue step"
  - »let $e=\{x,y\}$ be selected edge, let $T=(V,F)$ be an MST containing all blue edges (and no red ones) before the step
  - »if $e \in F$, T contains all blue edges (and no red ones) after the step
  - »if $e \notin F$, there is some other edge e' on simple path from x to y in T that is also in the cut selected by the blue rule (e' is not blue)
    - $T'=(V,F \cup \{e\} - \{e'\})$ is a spanning tree
    - since e' is not blue and $cost(e) \leq cost(e')$, T' is an MST and T' contains all the blue edges (and no red ones) after the step



- Suppose invariant is true before a "red step"
  - » let $e=\{x,y\}$ be selected edge and let $T=(V,F)$ be an MST that contains no red edges (and all blue edges) before the step
  - » if $e \notin F$ then T contains no red edges (and all blue) after step
  - » if $e \in F$, then removing e from T splits T into subtrees T_1 and T_2
    - there is some edge e' that is not in T, on the cycle selected by the red rule that joins a vertex in T_1 to a vertex in T_2 (e' is not red)
    - $T'=(V,F \cup \{e'\} - \{e\})$ is a spanning tree.
    - Since $cost(e) \geq cost(e')$, T' is an MST. T' contains no red edges (and all blue) after the step
- To see that all edges are colored, suppose that at some point $e=\{u,v\}$ remains uncolored.
  - » if u and v are connected by a blue path then that path plus e forms a cycle that the red rule can be applied to
  - » if u and v are not connected by a blue path, then there is a cut crossed by e that the blue rule can be applied to ■

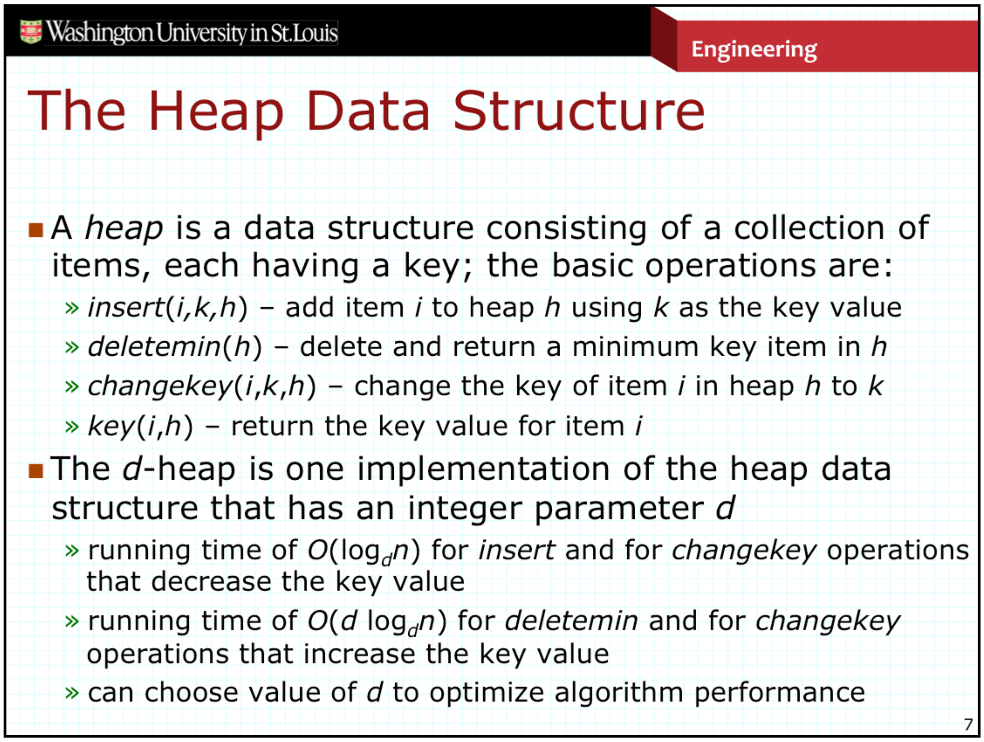


Prim’s Algorithm

- Build single blue tree, from an arbitrary starting vertex by repeating following step $n-1$ times
  - » select a minimum cost edge incident to the blue tree containing the starting vertex and color it blue
- The algorithm can also be expressed as follows.

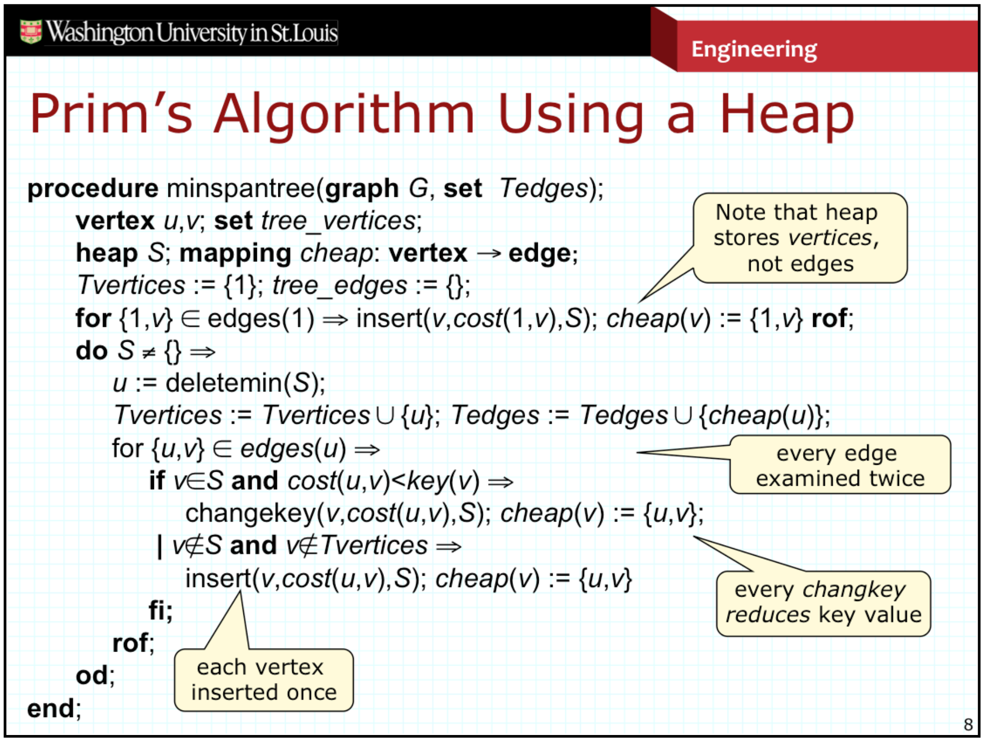
```
procedure minspantree(graph G, set Tedges);
    vertex w,u,v; set S, Tvertices;
    Tvertices := {1}; Tedges := {}; S:=neighbors(1);
    do $S \neq \{\} \Rightarrow$
            select min cost edge {w,u} from $w \in$ Tvertices to $u \in S$
            Tvertices:=Tvertices$\cup${u}; Tedges:=Tedges$\cup${w,u};
            $S := S - \{u\}$;
            for $\{u,v\} \in$ edges(u)$\Rightarrow$ if $v \notin$ Tvertices $\Rightarrow$ $S := S \cup \{v\}$ fi rof;
    od;   // edges not added to tree_edges are implicitly colored red
end;
```

need fast method to select min cost edge



The Heap Data Structure

- A *heap* is a data structure consisting of a collection of items, each having a key; the basic operations are:
  - » *insert*(*i*,*k*,*h*) – add item *i* to heap *h* using *k* as the key value
  - » *deletemin*(*h*) – delete and return a minimum key item in *h*
  - » *changekey*(*i*,*k*,*h*) – change the key of item *i* in heap *h* to *k*
  - » *key*(*i*,*h*) – return the key value for item *i*
- The *d*-heap is one implementation of the heap data structure that has an integer parameter *d*
  - » running time of $O(\log_d n)$ for *insert* and for *changekey* operations that decrease the key value
  - » running time of $O(d \log_d n)$ for *deletemin* and for *changekey* operations that increase the key value
  - » can choose value of *d* to optimize algorithm performance



Prim’s Algorithm Using a Heap

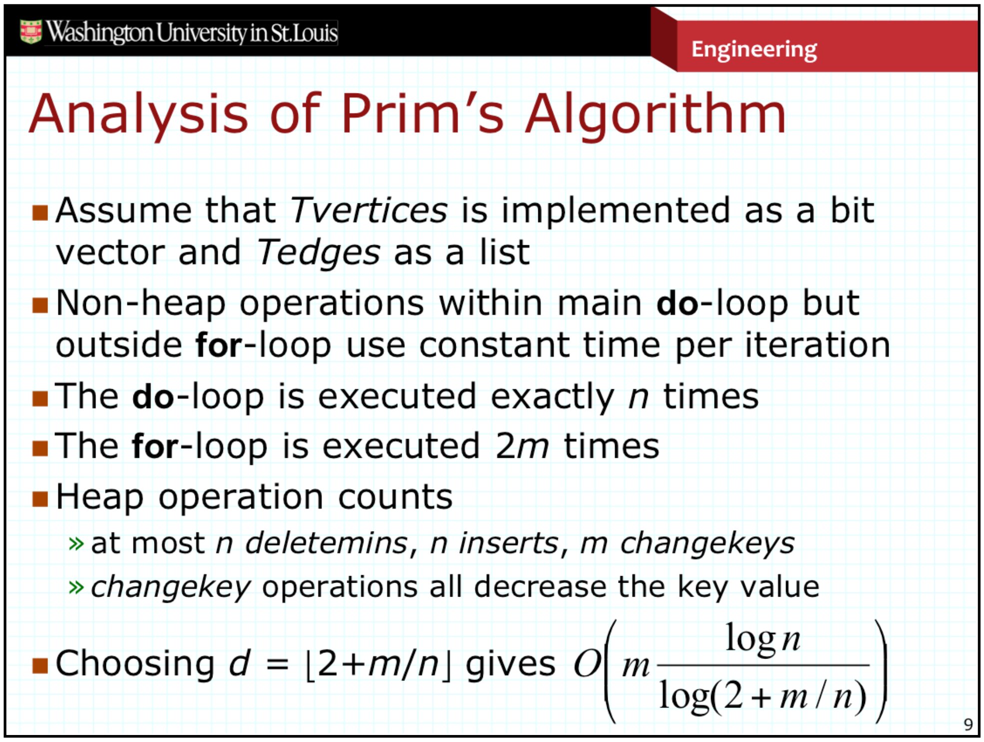
```
procedure minspantree(graph G, set Tedges);
    vertex u,v; set tree_vertices;
    heap S; mapping cheap: vertex → edge;
    Tvertices := {1}; tree_edges := {};
    for {1,v} ∈ edges(1) ⇒ insert(v,cost(1,v),S); cheap(v) := {1,v} rof;
    do S ≠ {} ⇒
        u := deletemin(S);
        Tvertices := Tvertices ∪ {u}; Tedges := Tedges ∪ {cheap(u)};
        for {u,v} ∈ edges(u) ⇒
            if v∈S and cost(u,v)<key(v) ⇒
                changekey(v,cost(u,v),S); cheap(v) := {u,v};
            | v∉S and v∉Tvertices ⇒
                insert(v,cost(u,v),S); cheap(v) := {u,v}
            fi;
        rof;
    od;
end;
```

Note that heap stores *vertices*, not edges

every edge examined twice

every *changkey reduces* key value

each vertex inserted once

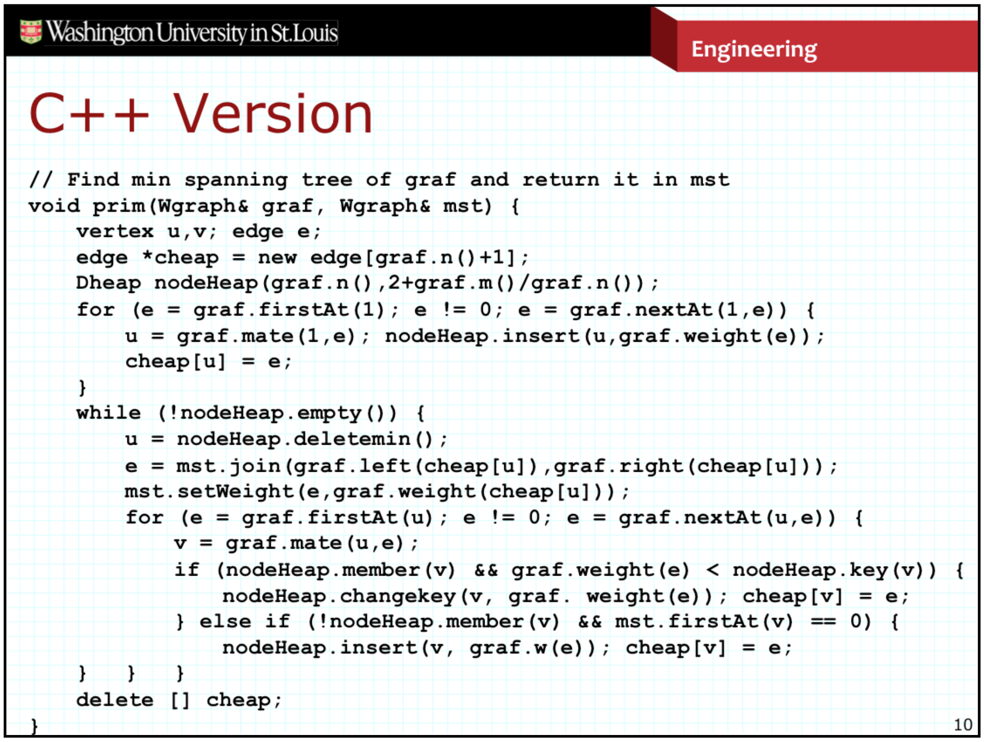


Washington University in St.Louis

Engineering

Analysis of Prim's Algorithm

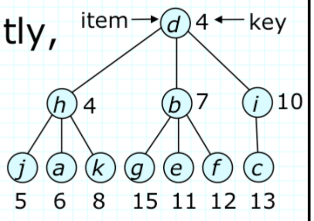
- Assume that *Tvertices* is implemented as a bit vector and *Tedges* as a list
- Non-heap operations within main **do**-loop but outside **for**-loop use constant time per iteration
- The **do**-loop is executed exactly *n* times
- The **for**-loop is executed 2*m* times
- Heap operation counts
  - at most *n deletemins*, *n inserts*, *m changekeys*
  - *changekey* operations all decrease the key value
- Choosing $d = \lfloor 2+m/n \rfloor$ gives $O\left(m \frac{\log n}{\log(2+m/n)} \right)$



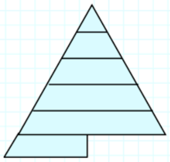
C++ Version

```
// Find min spanning tree of graf and return it in mst
void prim(Wgraph& graf, Wgraph& mst) {
    vertex u,v; edge e;
    edge *cheap = new edge[graf.n()+1];
    Dheap nodeHeap(graf.n(),2+graf.m()/graf.n());
    for (e = graf.firstAt(1); e != 0; e = graf.nextAt(1,e)) {
        u = graf.mate(1,e); nodeHeap.insert(u,graf.weight(e));
        cheap[u] = e;
    }
    while (!nodeHeap.empty()) {
        u = nodeHeap.deletemin();
        e = mst.join(graf.left(cheap[u]),graf.right(cheap[u]));
        mst.setWeight(e,graf.weight(cheap[u]));
        for (e = graf.firstAt(u); e != 0; e = graf.nextAt(u,e)) {
            v = graf.mate(u,e);
            if (nodeHeap.member(v) && graf.weight(e) < nodeHeap.key(v)) {
                nodeHeap.changekey(v, graf. weight(e)); cheap[v] = e;
            } else if (!nodeHeap.member(v) && mst.firstAt(v) == 0) {
                nodeHeap.insert(v, graf.w(e)); cheap[v] = e;
    }   }   }
    delete [] cheap;
}
```

d-Heaps



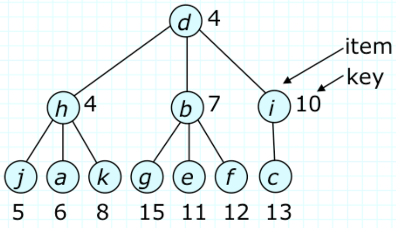
- Heaps can be implemented efficiently, using *heap-ordered* tree
  - each tree node contains one *item* with a real-valued *key*
  - key of each node ≥key of its parent
- A *d*-heap is *heap-shaped*, heap-ordered *d*-ary tree
  - let *T* be an infinite *d*-ary tree, with vertices numbered in breadth-first order
  - a subtree of *T* is *heap-shaped* if its vertices have consecutive numbers 1,2,...,*n*



- The depth of a *d*-heap with *n* vertices is $\leq\lceil \log_d n \rceil$

Implementing *d*-Heaps as Arrays

- *D*-heap can be stored in an array in breadth-first order
  - allows indices for parents and children to be calculated directly, eliminating the need for pointers



	1	2	3	4	5	6	7	8	9	10	11
h	*d*	*h*	*b*	*i*	*j*	*a*	*k*	*g*	*e*	*f*	*c*

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
key	6	7	13	4	11	12	15	4	10	5	8

- If i is index of item x, then $\lceil (i-1)/d \rceil$ is index of $p(x)$; indices of children of x are in range $[d(i-1)+2 \,..\, di+1]$
- When key of item is decreased, restore heap-order, by repeatedly swapping the item with its parent
  - similarly, for increasing an item's key

d-Heap Operations

```
item function findmin(heap $h$);
  return if $h=\{\} \Rightarrow$ null | $h \neq \{\} \Rightarrow h(1)$ fi;
end;

procedure siftup(item $i$, integer $x$, modifies heap $h$);
  integer $p$;
  $p := \lceil (x-1)/d \rceil$;
  do $p \neq 0$ and $key(h(p)) > key(i) \Rightarrow$
     $h(x) := h(p)$; $x := p$; $p := \lceil (p-1)/d \rceil$ ;
  od;
  $h(x) := i$;
end;

procedure insert(item $i$; modifies heap $h$);
  siftup($i$, $|h| + 1$, $h$);
end;
```

insert *i* at position *x* or above

at most one iteration per level in heap

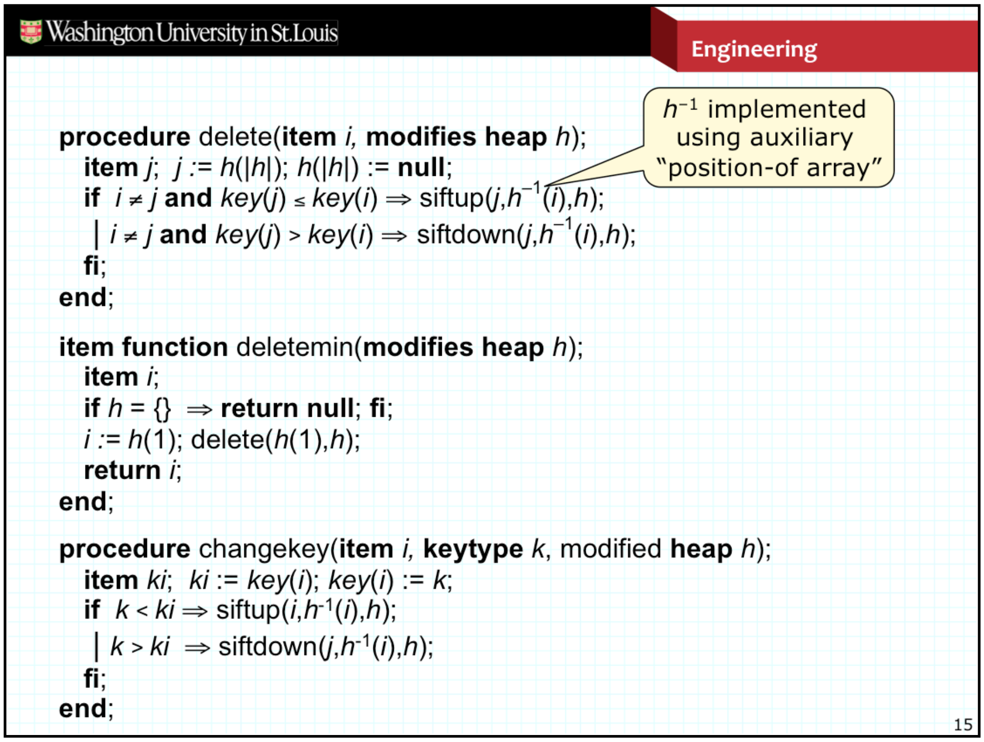
```
procedure siftdown(item $i$, integer $x$, modifies heap $h$);
   integer $c$;
   $c$ := minchild($x$,$h$);
   do $c \neq 0$ and $key(h(c)) < key(i)$ ⇒
        $h(x) := h(c)$; $x := c$; $c$ := minchild($x$,$h$);
   od;
   $h(x) := i$;
end;
```

insert *i* at position *x* or below

at most one iteration per level in heap

```
integer function minchild(integer $x$, heap $h$);
   integer $i$, $minc$;
   $minc := d(x-1) + 2$;
   if $minc > |h|$ ⇒ return 0 fi;
   $i := minc + 1$;
   do $i \leq \min\{|h|, dx+1\}$ ⇒
        if $key(h(i)) < key(h(minc))$ ⇒ $minc := i$ fi;
        $i := i + 1$;
   od;
   return $minc$;
end;
```

at most *d* iterations



h^{-1} implemented using auxiliary “position-of array”

```
procedure delete(item $i$, modifies heap $h$);
    item $j$;  $j$ := $h(|h|)$; $h(|h|)$ := null;
    if  $i \neq j$ and $key(j) \leq key(i)$ ⇒ siftup($j,h^{-1}(i),h$);
     | $i \neq j$ and $key(j) > key(i)$ ⇒ siftdown($j,h^{-1}(i),h$);
    fi;
end;

item function deletemin(modifies heap $h$);
    item $i$;
    if $h$ = {}  ⇒ return null; fi;
    $i$ := $h(1)$; delete($h(1),h$);
    return $i$;
end;

procedure changekey(item $i$, keytype $k$, modified heap $h$);
    item $ki$;  $ki$ := $key(i)$; $key(i)$ := $k$;
    if  $k < ki$ ⇒ siftup($i,h^{-1}(i),h$);
     | $k > ki$  ⇒ siftdown($j,h^{-1}(i),h$);
    fi;
end;
```

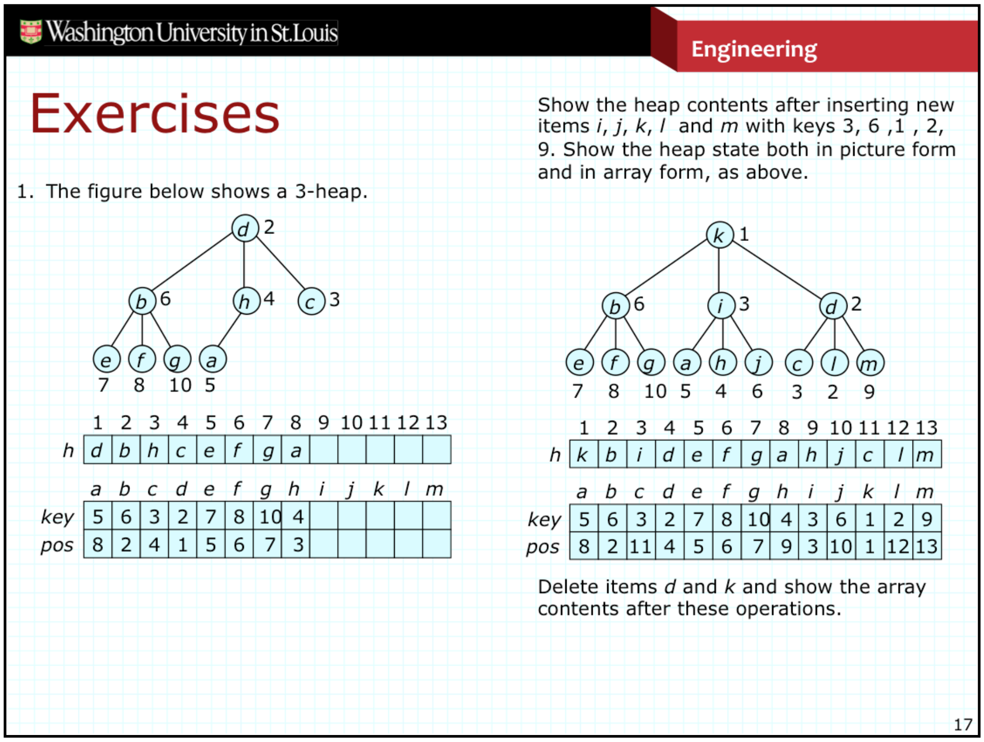
Analysis of *d*-Heap Operations

```
heap function makeheap(set of item $s$);
    integer $j$; heap $h$;
    $h := \{\}$;
    for $i \in s \Rightarrow j := |h|+1;\ h(j) = i;$ rof;
    $j = \lceil (|h|-1)/d \rceil$;
    do $j>0 \Rightarrow$ siftdown($h(j),j,h$); $j = j-1$; od;
    return $h$;
end;
```

- Each execution of *siftup* (and hence *insert*) takes $O(\log_d n)$ time, while each execution of *siftdown* takes $O(d \log_d n)$ time
- Time for *changekey* depends on whether keys increase or decrease
    - if keys always decrease, can make *changekey* faster using a large d
- The running time for *makeheap* is $O(f)$ where

$$f(n) = \frac{n}{d}d + \frac{n}{d^2}2d + \frac{n}{d^3}3d + \cdots$$

which is $O(n)$



Exercises

1. The figure below shows a 3-heap.

	1	2	3	4	5	6	7	8	9	10	11	12	13
h	*d*	*b*	*h*	*c*	*e*	*f*	*g*	*a*					

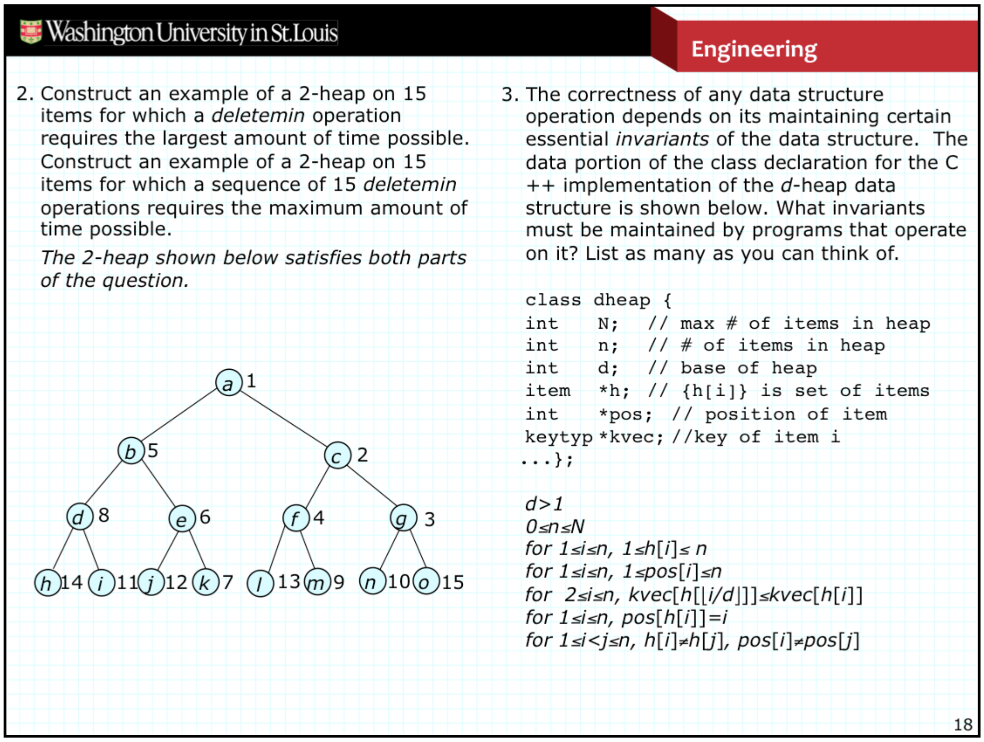
	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
key	5	6	3	2	7	8	10	4					
pos	8	2	4	1	5	6	7	3					

Show the heap contents after inserting new items *i*, *j*, *k*, *l* and *m* with keys 3, 6 ,1 , 2, 9. Show the heap state both in picture form and in array form, as above.

	1	2	3	4	5	6	7	8	9	10	11	12	13
h	*k*	*b*	*i*	*d*	*e*	*f*	*g*	*a*	*h*	*j*	*c*	*l*	*m*

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
key	5	6	3	2	7	8	10	4	3	6	1	2	9
pos	8	2	11	4	5	6	7	9	3	10	1	12	13

Delete items *d* and *k* and show the array contents after these operations.



2. Construct an example of a 2-heap on 15 items for which a *deletemin* operation requires the largest amount of time possible. Construct an example of a 2-heap on 15 items for which a sequence of 15 *deletemin* operations requires the maximum amount of time possible.

   The 2-heap shown below satisfies both parts of the question.

3. The correctness of any data structure operation depends on its maintaining certain essential *invariants* of the data structure. The data portion of the class declaration for the C++ implementation of the *d*-heap data structure is shown below. What invariants must be maintained by programs that operate on it? List as many as you can think of.

```
class dheap {
int    N;   // max # of items in heap
int    n;   // # of items in heap
int    d;   // base of heap
item   *h;  // {h[i]} is set of items
int    *pos; // position of item
keytyp *kvec; //key of item i
...};
```

$d>1$

$0 \leq n \leq N$

for $1 \leq i \leq n$, $1 \leq h[i] \leq n$

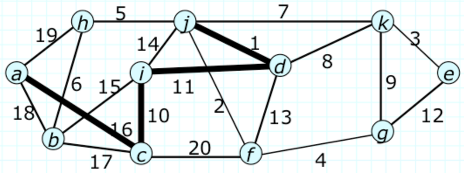
for $1 \leq i \leq n$, $1 \leq pos[i] \leq n$

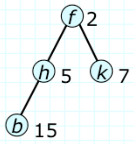
for $2 \leq i \leq n$, $kvec[h[\lfloor i/d \rfloor]] \leq kvec[h[i]]$

for $1 \leq i \leq n$, $pos[h[i]]=i$

for $1 \leq i < j \leq n$, $h[i] \neq h[j]$, $pos[i] \neq pos[j]$

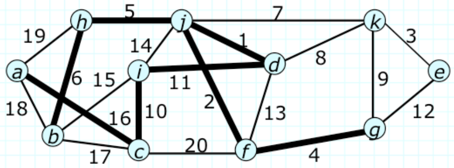
4. The figure below shows an intermediate state in the execution of Prim’s algorithm. The heap and the *cheap* mapping are shown below (d=2).

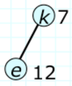




	a	b	c	d	e	f	g	h	i	j	k
cheap		*bi*	*ac*	*di*		*fj*		*hj*	*ci*	*dj*	*jk*

Show the state after four more steps have been performed.





	a	b	c	d	e	f	g	h	i	j	k
cheap		*bh*	*ac*	*di*	*eg*	*fj*	*fg*	*hj*	*ci*	*dj*	*jk*