Discussion Session 7: Priority Queues and Disjoint Sets

CS 225: Data Structures
& Software Principles

Agenda

- **Priority Queues/Heaps**
  - Operations on Heaps
  - Implementations of Heaps
  - More about heaps (BuildHeap, HeapSort)
- Disjoint Sets
  - Operations on Disjoint Sets
  - Up Trees
    - Union
    - Find
  - Array Implementation

By the end of this class, you

- Need to
  - Understand priority queue operations using heaps
  - Be able to manually simulate heap operations
  - Be able to manually simulate up-tree operations for disjoint sets
  - Be able to implement the Disjoint Set ADT using arrays
- Ought to be able to implement a Heap
- Might want to think about analysis of the functions discussed.

Priority Queues

- **Definition**: A Priority Queue is a set ADT of such pairs (K,I) supporting the following operations where **K** $\in$ **key**, a set of linearly ordered key values and **I** is associated with some information of type **Element.**
  - MakeEmptySet( )
  - IsEmptySet(S)
  - Insert(K,I,S): Add pair (K,I) to set S
  - FindMin(S): Return an **element** I such that (K,I) $\in$ S and K is minimal with respect to the ordering
  - DeleteMin(S): Delete an element (K,I) from S such that K is minimal and return I

Priority Queues

- The elements have an intrinsic **priority**; we can Insert elements and Remove them in order of their priority, independent of the time sequence in which they were inserted
- Normally, the item with lowest key value is considered to have highest priority
- Priority Queues are structures that are optimized for finding the minimal (= highest priority) element in the tree using FindMin( )

Applications of Priority Queues

- Scheduling
- Sorting
- Merging sorted lists
- Selection

Priority Queues

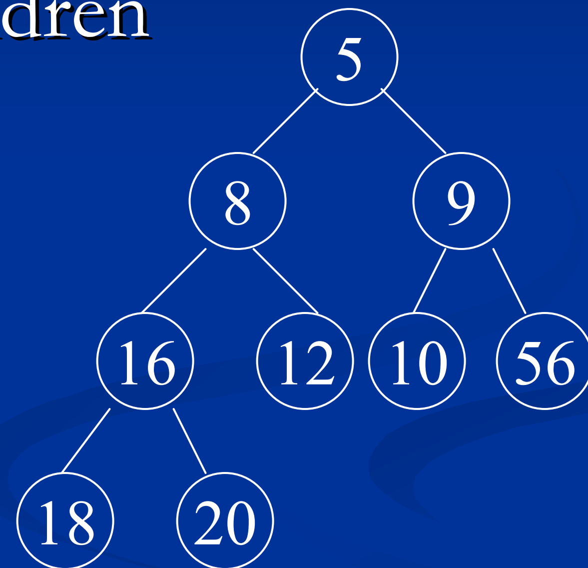
- Priority Queues can be implemented
  - using balanced trees
  - as a **heap**
- A **double-ended Priority Queue** is one which supports *FindMax* and *DeleteMax* operations along with *FindMin* and *DeleteMin* operations

Priority Queues

- **Partially ordered tree**
  - It is a Key tree of elements such that the priority of each node is greater than or equal to that of each of its children
  - The highest priority element of the tree is located at the root.
  - Tree requires reordering when the highest priority element is deleted or when a new element is inserted into the tree

Priority Queues

- In other words, a node's key value (priority) is <= the key value of its children



- No conclusion can be drawn about the relative order of the items in the left and right sub-trees of a node

Heaps

- **Heap**
  - An **implicitly represented complete** partially ordered Key tree is called a heap
- Run-time Analysis
  - **FindMin(S)** takes O(1) time
  - **Insert(K,I,S)** takes O(log n) time
    - Actually, Average case is O(1), but we don't use this in 225
  - **DeleteMin(S)** takes O(log n) time
- Operations on heaps to be discussed
  - Insert
  - DeleteMin

Heaps

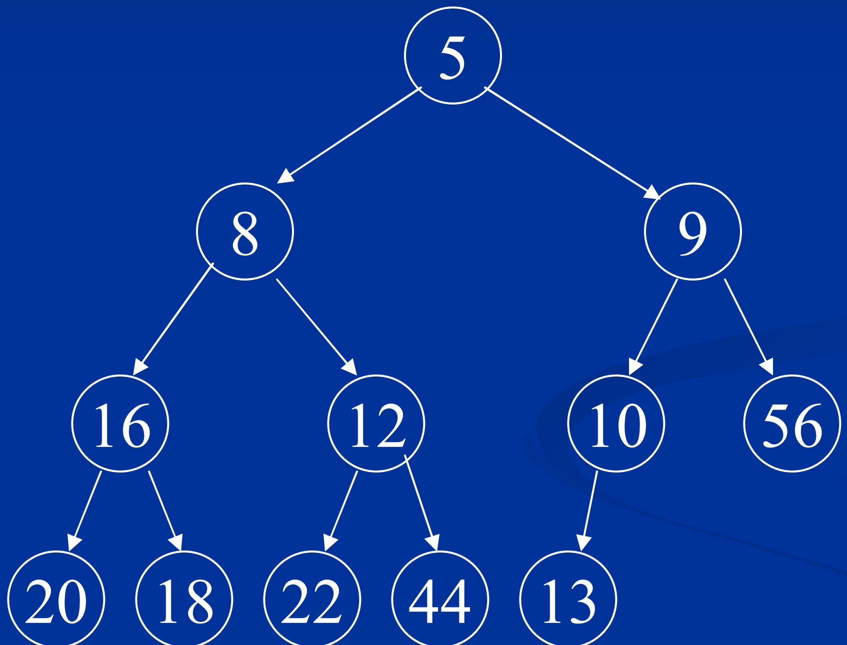
- Insert
  - Step 1: Append the new element in its natural position as a new leaf node in the tree and call that node **x**
  - Step 2:

    if(**x**'s parent's priority > **x**'s priority)
    - Swap contents of **x** and **x**'s parent and reassign **x** to **x**'s parent
    - Repeat from Step 2

    else
    - Algorithm TERMINATES

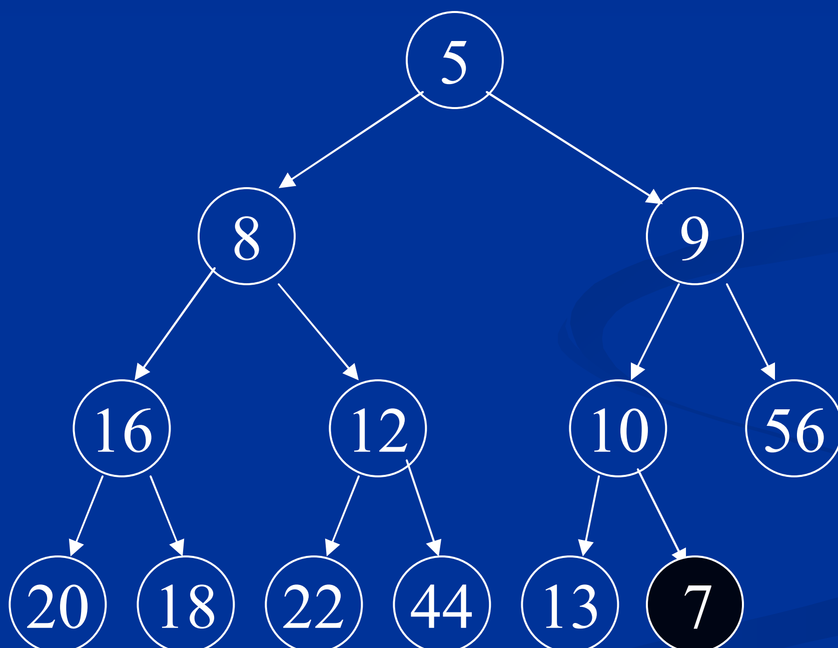
Heaps

- Given tree

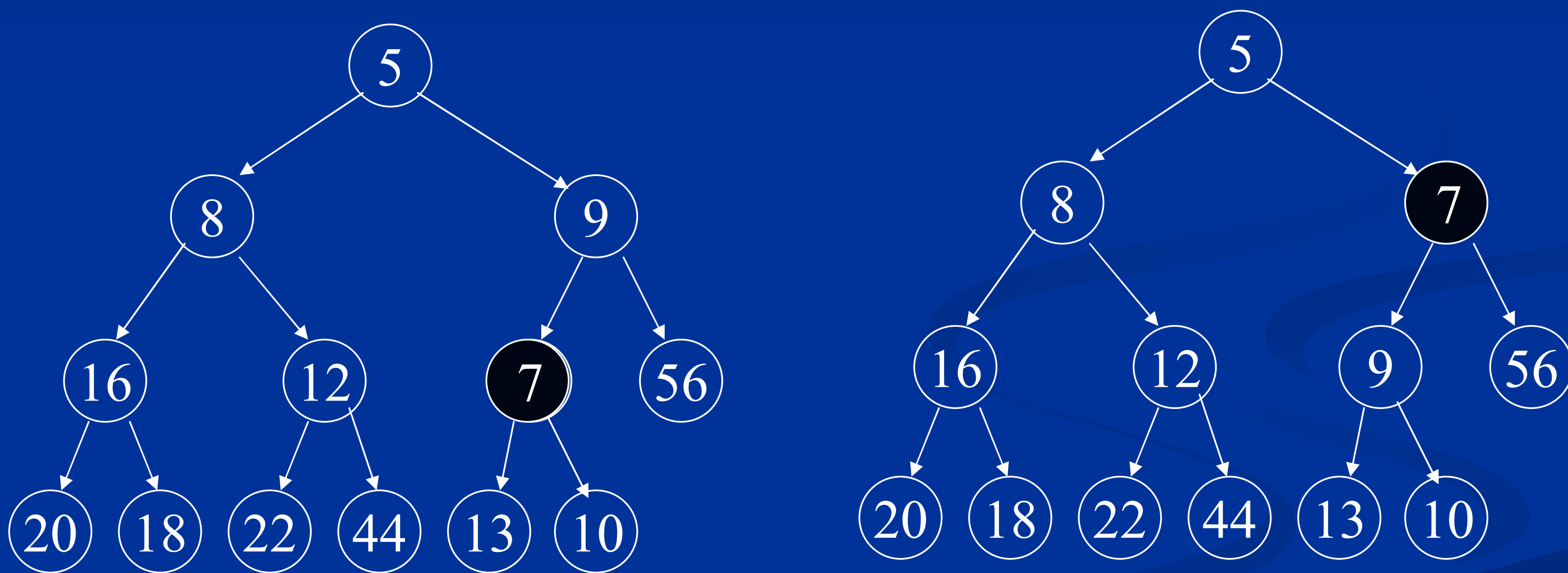


Heaps

- Insert node of priority 7



Heaps



Heaps

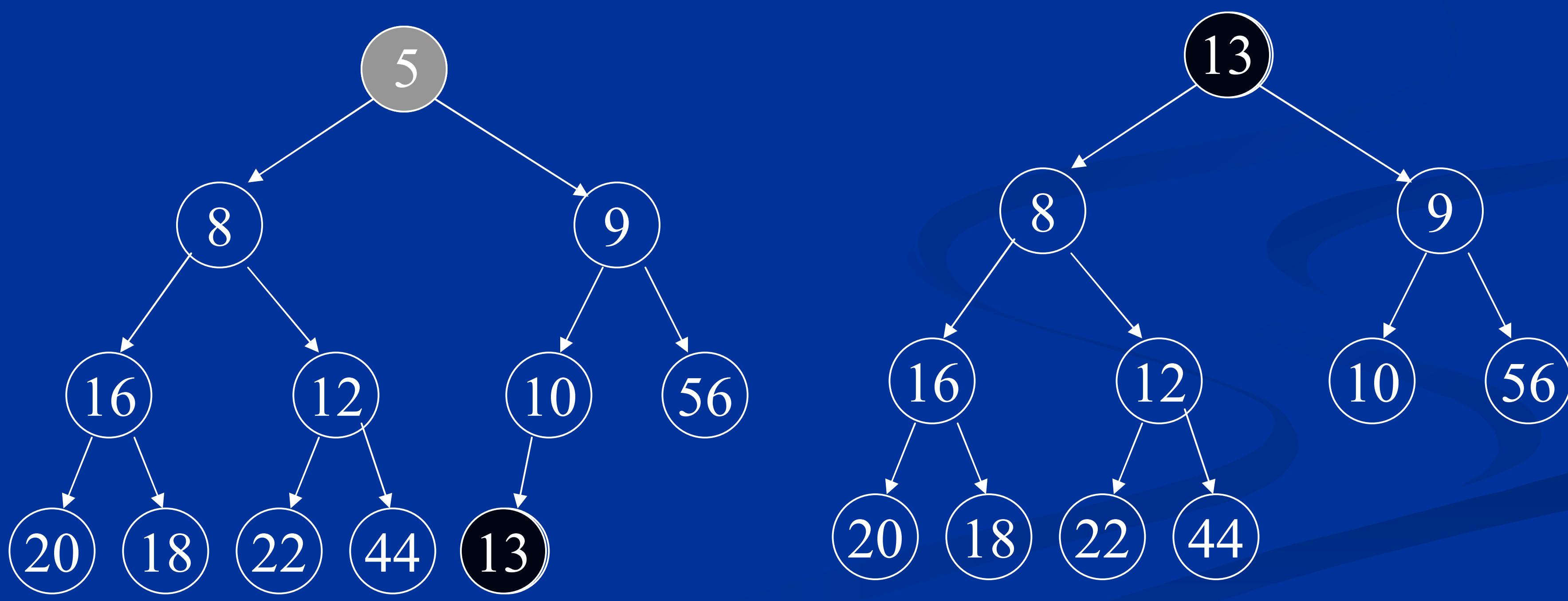
- DeleteMin
  - Step 1: **minimum_value** = contents of the root node. Find the rightmost leaf on the lowest level of the tree and call that node **x** and copy its contents to the root node and delete the node **x**
  - Step 2: Set **x** to point to root node
  - Step 3:

    if (**x**'s priority >= priority of any one of **x**'s children)
    - Swap contents of **x** and the lower valued child and reassign **x** to point that corresponding child
    - Repeat from Step 3

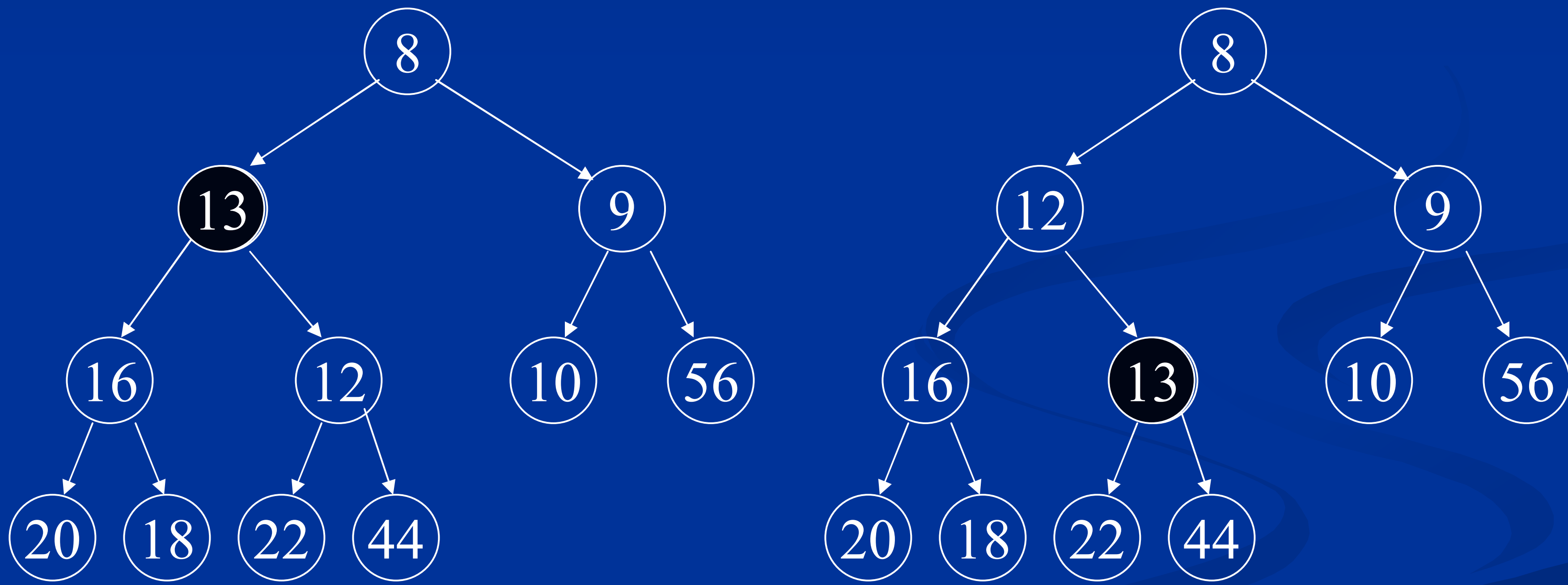
    else
    - Return minimum_value. Algorithm TERMINATES

Heaps

- DeleteMin on given tree



Heaps

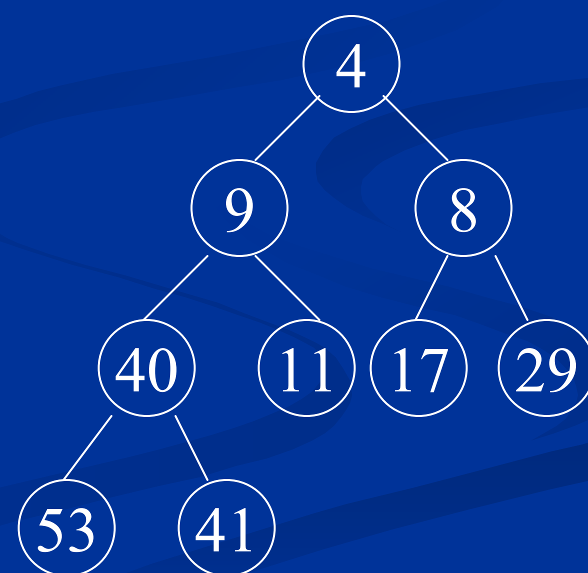


Agenda

- Priority Queues/Heaps
  - Operations on Heaps
  - **Implementations of Heaps**
  - More about heaps (BuildHeap, HeapSort)
- Disjoint Sets
  - Operations on Disjoint Sets
  - Up Trees
    - Union
    - Find
  - Array Implementation

Heaps

- Heap Implementation
  - Heaps being implicitly represented are implemented using arrays
  - Equations
    - LeftChild(i) = 2 i
    - RightChild(i) = 2 i + 1
    - Parent(i) = ⌊ i/2 ⌋



4	9	8	40	11	17	29	53	41

1 2 3 4 … 9

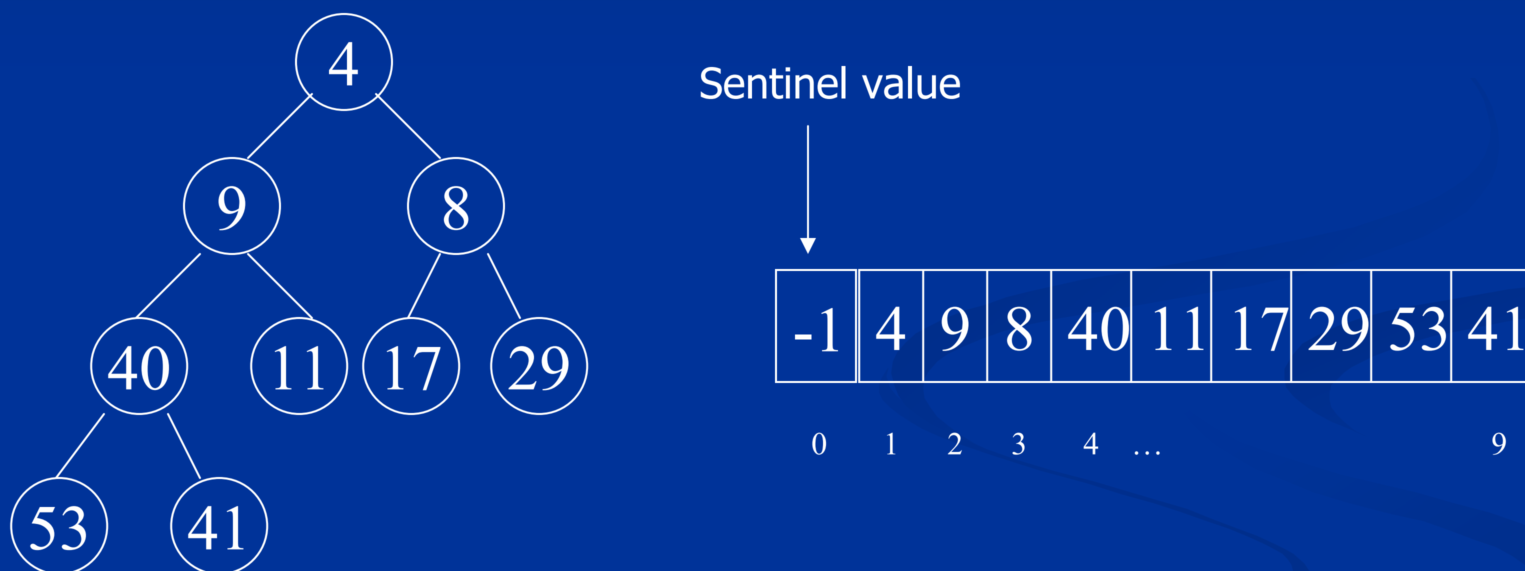
Heaps

- In practice, there is no “exchanging” of values as the proper position of an item is located by searching up the tree (during insertion) or down the tree (during deletion). Instead a “hole” is moved up or down the tree, by shifting the items along a path down or up single edges of the tree. The item is moved only once, at the last step

Heaps

- Another optimization is to use a **sentinel** value.
  - heap[0] = sentinel, where heap is an array used to represent the heap
  - the sentinel value is a key(priority) value less than **any** value that could be present in the heap
  - used so that swapping with parent at the root node is also handled without any special consideration

Heaps


4
9
8
40
11
17
29
53
41
Sentinel value
-1
4
9
8
40
11
17
29
53
41
0
1
2
3
4 …
9

Sample Code for Heaps

- Sample code available at
  ~cs225/src/library/07-pqds
- The Heap Applet can be seen at
  http://www.cs.pitt.edu/~kirk/cs1501/animations/PQueue.html
- The code for the following operations will be discussed
  - Insert
  - DeleteMin
  - FindMin

Class declaration

heap.h

```
template <class Etype>
class Binary_Heap
{
 private:
   unsigned int Max_Size;   // defines maximum size of the array
   unsigned int Size;       // defines current number of elements
   Etype *Elements;         // array that holds data
public:
   Binary_Heap(unsigned int Initial_Size = 10);
   ~Binary_Heap( ) {delete [ ] Elements; }
   void Make_Empty()   { Size=0;}
   int Is_Empty( ) const {return Size==0;}
   int Is_Full( ) const  {return Size==Max_Size; }
   void Insert(const Etype& X);
   Etype Delete_Min( );
   Etype Find_Min( ) const;
};
```

Heap Constructor

heap.cpp

```
// default constructor
template <class Etype>
Binary_Heap<Etype>::Binary_Heap(unsigned int Initial_Size)
{
   Size=0;
   Max_Size = Initial_Size;
   Elements = new Etype[Max_Size+1];
   Assert(Elements!=NULL, "Out of space in heap constructor");
   Elements[0] = -1;      // sentinel value
}
```

Insert

heap.cpp

```
// inserts the value passed as parameter into the heap
template <class Etype>
void Binary_Heap<Etype>::Insert(const Etype & X)   {
   Assert(!Is_Full(), "Priority queue is full");

   unsigned int i = ++Size;       // may have to resize array….
   while (i != 1 && Elements[i/2]>X)
   {
      Elements[i]=Elements[i/2]; // swap bubble with number above
      i /= 2;
   }
   Elements[i]=X;   // bubble, which has now come to rest, is given
                    //  the new element
}
```

Delete_Min

heap.cpp

```
template <class Etype>
Etype Binary_Heap<Etype>::Delete_Min( )  {
  unsigned int Child;
  Assert(!Is_Empty(), "Priority Queue is Empty");
  Etype Min_Element = Elements[1];
  Etype Last_Element = Elements[Size--];
  for (int i=1; i*2 <= Size; i=Child)  {
    // Find smaller child
    Child = i*2;       // child is left child
    if (Child !=Size && Elements[Child+1] < Elements[Child])
      Child++;
    // Percolate one level
    if (Last_Element > Elements[Child])
      Elements[i] = Elements[Child];
    else
      break;    }
  Elements[i] = Last_Element;
  return Min_Element;  }
```

Find_Min

heap.cpp

```
// returns minimum element of the heap without deleting it
template <class Etype>
Etype Binary_Heap<Etype>::Find_Min( ) const
{
   Assert(!Is_Empty(), "Priority Queue is Empty");
   return Elements[1];
}
```

Agenda

- Priority Queues/Heaps
  - Operations on Heaps
  - Implementations of Heaps
  - **More about heaps (BuildHeap, HeapSort)**
- Disjoint Sets
  - Operations on Disjoint Sets
  - Up Trees
    - Union
    - Find
  - Array Implementation

More on Heaps

- We can have a method called *BuildHeap* which will initialize a Heap in O(**n**) time rather than simply doing **n** Insertions into the heap which will take O(**n** log **n**) time
- It converts an array into a heap
- for(i = **n**/2 to 1)
  - PercolateDown(i)
- **n**/2 represents the first element from the right end of the array that has children

BuildHeap

```
void BuildHeap(Array<int>& theArray)  //indexed 1 to n
{   for (int j = theArray.Size( )/2; j >= 1; j--)
                Percolate(j, theArray);
}
void Percolate(int i, Array<int>& theArray) {
      int child, temp;
      while(1) {
                child = i*2;       // child is left child
               if(child > theArray.Size( ))
                                break;
                if(child != theArray.Size( ) && theArray[child+1] < theArray[child])
                                child++;
               if (theArray[i] > theArray[child]) {
                                temp = theArray[i];
                                theArray[i] = theArray[child];
                                theArray[child] = temp;
                                i = child;
               }
              else
                                break;
     } //end of while
 }
```

More on Heaps (Contd.)

- *HeapSort*
  - We can do sorting using a Heap
  - Simply build a heap using *BuildHeap* from the given array and then empty it
  - This means of sorting will have a run-time complexity of O(**n** log **n**)
- Heaps
  - MaxHeaps
  - MinHeaps

Agenda

- Priority Queues/Heaps
  - Operations on Heaps
  - Implementations of Heaps
  - More about heaps (BuildHeap, HeapSort)
- **Disjoint Sets**
  - Operations on Disjoint Sets
  - Up Trees
    - Union
    - Find
  - Array Implementation

Disjoint Sets

- Disjoint Sets
  - We have a fixed set **U** of Elements $\mathbf{X}_i$
  - **U** is divided into a number of disjoint subsets S_1, S_2, S_3, … S_k
  - $S_i \cap S_j$ is empty $\forall\ i \neq j$
  - $S_1 \cup S_2 \cup S_3 \cup \ldots S_k = \mathbf{U}$

Disjoint Sets

- Operations on Disjoint Sets
  - **MakeSet(X):** Return a new set consisting of the single item **X**
  - **Union(S,T):** Return the set **S** ∪ **T**, which replaces **S** and **T** in the database
  - **Find(X):** Return that set **S** such that **X** ∈ **S**

Agenda

- Priority Queues/Heaps
  - Operations on Heaps
  - Implementations of Heaps
  - More about heaps (BuildHeap, HeapSort)
- Disjoint Sets
  - Operations on Disjoint Sets
  - **Up Trees**
    - Union
    - Find
  - Array Implementation

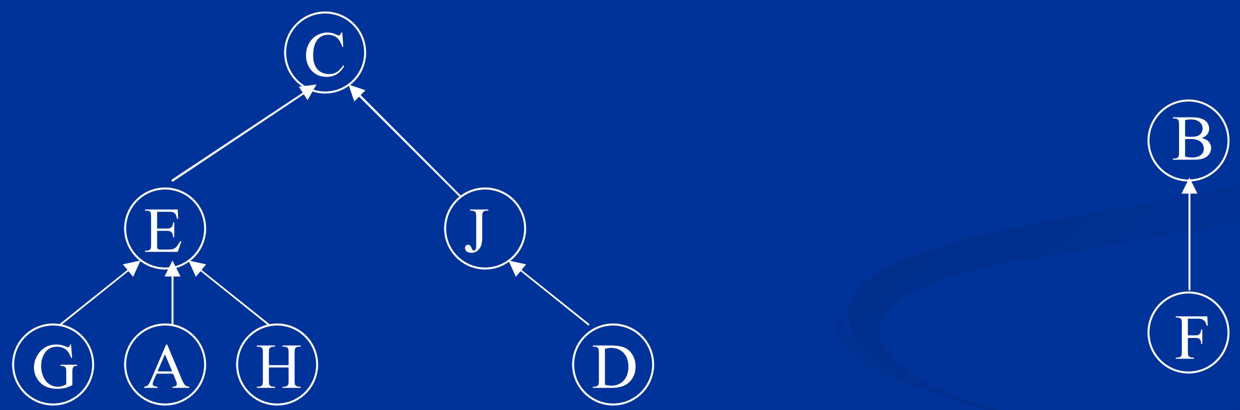
Up-Trees

- If each element can belong to only one set (definition of disjoint), a tree structure known as an **up-tree** can be used to maintain disjoint sets
- In an up-tree we have pointers up the tree from children to parents

Up-Trees

- Up-Tree Properties
  - Each node has a single pointer field to point to its parent; at the root this field is empty
  - A node can have any number of children
  - The sets are identified by their root nodes

Up-Trees



Disjoint Sets: {A,C,D,E,G,H,J} and {B,F}

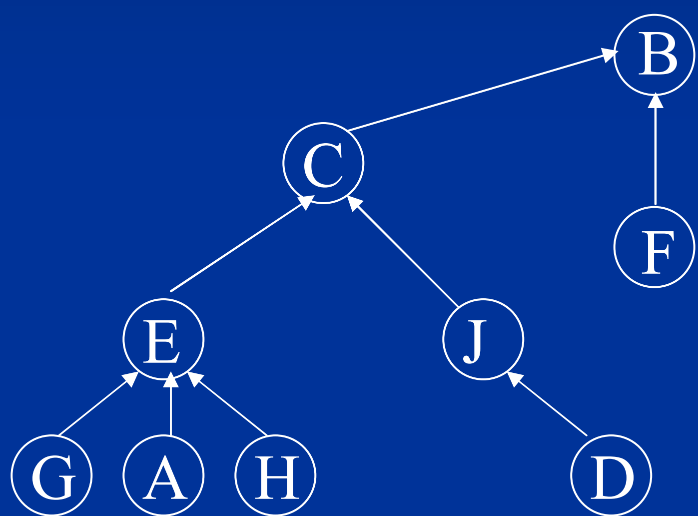
Up-Trees

- **Union:**
  - To form the Union of sets **S** and **T**, just make the root of one tree point to the tree of the other. If we make root of **S** point to the root of **T**, we say we are merging **S** into **T**
  - To prevent the linear growth of the height of the tree, we ensure that we always merge the smaller tree into the larger one (merging the tree with fewer nodes into the one with more nodes)

Up-Trees

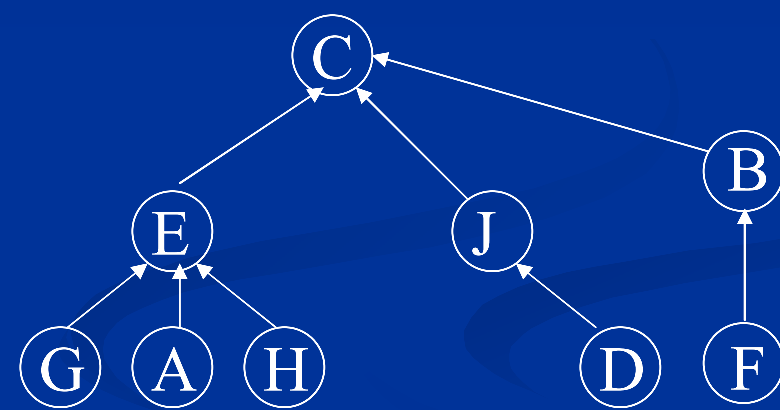
- This is called the **balanced merging strategy**
- We take care to prevent the linear growth of the tree's height because the **Find** operation takes time proportional to the height of the tree in the worst case
- Each node has an additional **Count** field that is used, if the node is the root, to hold the count of the number of nodes in the tree

Up-Trees



(a)

Incorrect Way



(b)

Correct Way

Up-Trees

- **Find**
  - To find which set an element belongs to, follow the pointers up the tree until we reach the root
  - If we assume that when doing a **Find(X)** we know the location of the node **X**, then **Find** can be implemented in O(log n) time; i.e. the operation of **LookUp** taken constant time

Up-Trees

- If we cannot directly access the node **X** then the operation of **LookUp** can take logarithmic time
- So **Find** would first take logarithmic time for **LookUp** and then logarithmic time again to search the up-tree, but the total time would still be logarithmic, i.e. **Find** can still be implemented in O(log n) time

Up-Trees

- **Path Compression:**
  - A **Find** would take less time in a shallow, bushy tree than it would in a tall, skinny tree
  - By using our balanced merging strategy to prevent the growth in the tree's height, we have ensured that the height can at worst be logarithmic in size
  - However, since any number of nodes can have the same parent, we can **restructure** our up-tree to make it bushier

Up-Trees

- This restructuring is called **Path Compression**. After doing a **Find**, make any node along the path to the root point directly to the root
- Any subsequent **Find** on any one of these nodes, or their descendents, will take less time since the node is now closer to the root. **Find** now takes almost constant time; O(log* **n**) ***amortized worst-case time*** to be precise

Agenda

- Priority Queues/Heaps
  - Operations on Heaps
  - Implementations of Heaps
  - More about heaps (BuildHeap, HeapSort)
- Disjoint Sets
  - Operations on Disjoint Sets
  - Up Trees
    - Union
    - Find
  - **Array Implementation**

Up-Trees

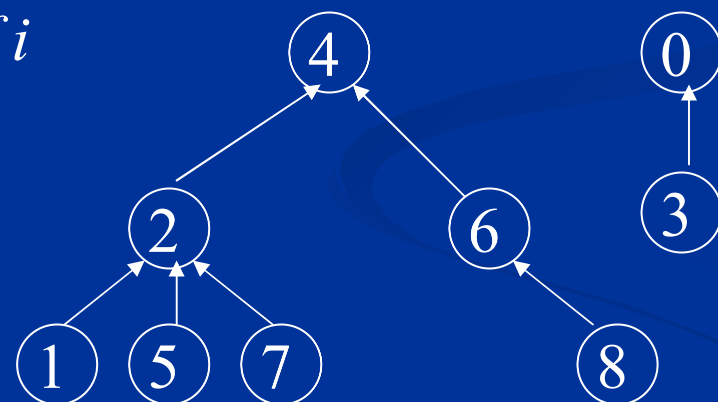
- **Array Implementation**
  - If we assume all elements of the universe to be integers from 0 to N, then we can represent the Up-Trees as one Array of size N

Up-Trees

a	-1	2	4	0	-1	2	4	2	6
	0	1	2	3	4	5	6	7	8

a[i] = parent of i

a[root] = -1



Sample Code

- All code discussed in class available at
  ~cs225/src/library/07-pqds/disjoint
- Operations to be discussed
  - Find(X)
  - SetUnion(S,T)
- We will first see sample code for a simple disjoint set implementation without the balanced merging strategy or path compression and then an efficient implementation that uses both

Simple Implementation

Class declaration

arraydisjoint.h

```
class DisjointSets
{
public:
  DisjointSets(unsigned int numElems = 10);
  DisjointSets(DisjointSets const & origDS);
  ~DisjointSets( ) { delete SetArray; }
  const DisjointSets& operator=(DisjointSets const & origDS);
  virtual void SetUnion(unsigned int elem1, unsigned int elem2);
  virtual unsigned int Find(unsigned int elem);

protected:
  Array<int> *SetArray;
  int SetSize;
};
```

Find

arraydisjoint.cpp

```
// Find returns the "name" of the set containing elem.
unsigned int DisjointSets::Find(unsigned int elem)
{
  if ( (*SetArray)[elem] <= 0 )
     return elem;
  else
    return Find((*SetArray)[elem]);
}
```

SetUnion

arraydisjoint.cpp

```
void DisjointSets::SetUnion(unsigned int elem1, unsigned int elem2)
{
 unsigned int root1, root2;
 // check if elem1 is root, if not, find the root
 if ( (*SetArray)[elem1] > 0 )
  root1 = Find(elem1);
 else
  root1 = elem1;
 // same for elem2
 if ( (*SetArray)[elem2] > 0 ) {
  root2 = Find(elem2);
 else
  root2 = elem2;
 // set root2 to be child of root1
 (*SetArray)[root2] = root1;
}
```

Efficient Implementation

Find

smartunion.cpp

```
unsigned int DSetsBySize::Find(unsigned int elem)
{
 if ( (*SetArray)[elem] <= 0 ) {
  return elem;
 }else {
  // Recursively set array to whatever Find returns.  Find will
  //  return the root, thus each node from this one up is set to root.
  return ((*SetArray)[elem] = Find( (*SetArray)[elem] ));
 }
}
```

SetUnion

smartunion.cpp

```
void DSetsBySize::SetUnion(unsigned int elem1, unsigned int elem2)
{unsigned int root1, root2;
 // check if elem1 is root, if not, find the root
 if ( (*SetArray)[elem1] > 0 )
        root1 = Find(elem1);
 else
        root1 = elem1;
  // same for elem2
 if ( (*SetArray)[elem2] > 0 )
        root2 = Find(elem2);
 else
        root2 = elem2;
```

SetUnion(Contd.)

smartunion.cpp

```
if ( (*SetArray)[root2] < (*SetArray)[root1] ) {
  // root2 has greater size, since size is given as
  //  the negation of actual size
  // find the size of the union, and make root1 the child of root2
  (*SetArray)[root2] += (*SetArray)[root1];
  (*SetArray)[root1] = root2;
 }else {  // root1 has greater height or they have equal heights
  // find the size of the union, and make root2 the child of root1
  (*SetArray)[root1] += (*SetArray)[root2];
  (*SetArray)[root2] = root1;
 }
}//end of SetUnion
```

Summary

- Priority Queues/Heaps
  - Operations on Heaps
  - Implementations of Heaps
  - More about heaps (BuildHeap, HeapSort)
- Disjoint Sets
  - Operations on Disjoint Sets
  - Up Trees
    - Union
    - Find
  - Array Implementation