Data Structures

Topic #11

Today's Agenda

- **Complete our discussion of trees**

- Learn about heaps

- walk through the deletion algorithms for:
  - AVL
  - 2-3, 2-3-4

- **Learn about graphs**

Heaps

- A heap is a data structure similar to a binary search tree.

- However, heaps are not sorted as binary search trees are.

- And, heaps are always complete binary trees.

- Therefore, we always know what the maximum size of a heap is.

Heaps

- Unlike a binary search tree, the value of each node in a heap is greater than or equal to the value in each of its children.

- In addition, there is no relationship between the values of the children; you don't know which child contains the larger value.

- Heaps are used to implement priority queues.

Heaps

- A priority queue is an Abstract Data Type!

- Which, can be implemented using heaps. Think of a To-Do lists; each item has a priority value which reflects the urgency with which each item needed to be addressed.

Heaps

- By preparing a priority queue, we can determine which item is the next highest priority.

- A priority queue maintains items sorted in descending order of their priority value -- so that the item with the highest priority value is always at the beginning of the list.

Heaps

- Priority Queue ADT operations can be implemented using heaps...

    - which is a weaker binary tree but sufficient for the efficient performance of priority queue operations.

    - Let's look at heap:
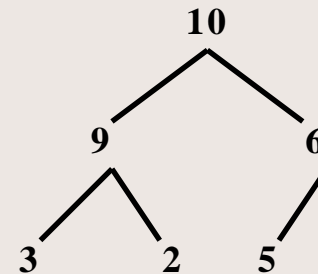
Heaps

- To remove an item from a heap, we remove the largest item (or the item with the highest priority).

- Because the value of every node is greater than or equal to that of either of its children, the largest value must be the root of the tree.

- A remove operation is simply to remove the item at the root and return it to the calling routine.

Heaps

- Once you have removed the largest value, you are left with two disjoint heaps:
  - Therefore, you need to transform the nodes
  - Move the last node and place it in the root
  - Then take that value and trickle it down the tree until it reaches a node in which it will not be out of place.
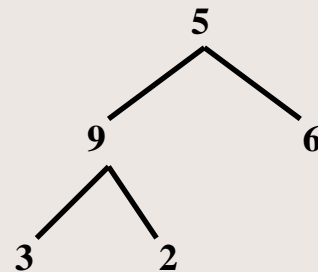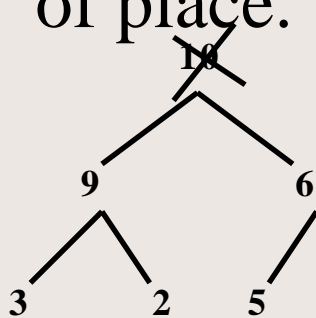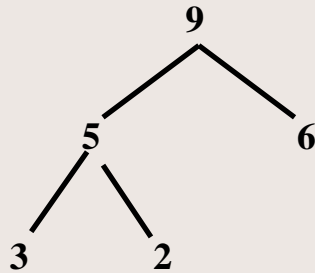

Step 1
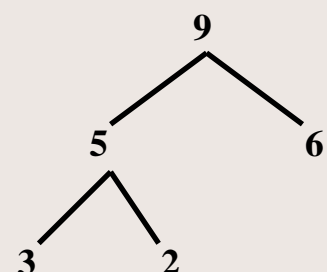
Step 2

Heaps

- To insert an item, we use just the opposite strategy.

  - We insert at the bottom of the tree and trickle the number upward to be in the proper place.

  - With insert, the number of swaps cannot exceed the height of the tree -- so that is the worst case!

  - Which, since we are dealing with a binary tree, this is always approximately $\log_2(N)$ which is very efficient.
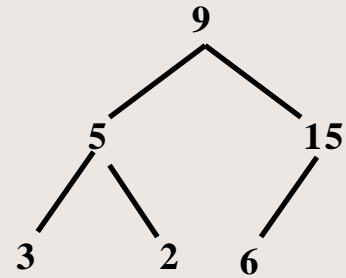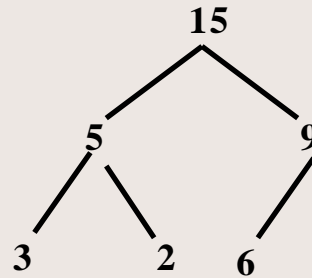
Heaps

9
5   6
3   2

Step 1:insert 15

9
5   6
3   2   15

Step 2

9
5   15
3   2   6

Step 3

15
5   9
3   2   6
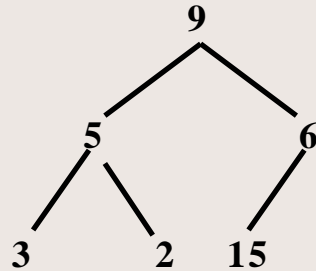
Step 5

Heaps

- The real advantage of a heap is that it is always balanced.
  - It makes a heap more efficient for implementing a priority queue than a binary search tree because the operations that keep a binary search tree balanced are far more complex than the heap operations.
  - However, heaps are not useful if you want to try to traverse a heap in sorted order -- or retrieve a particular item.
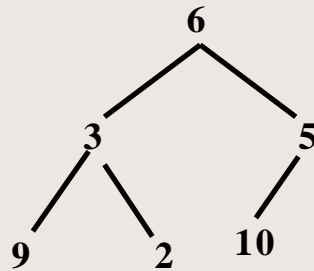
Heapsort

- A heapsort uses a heap to sort a list of items that are not in any particular order.

- The first step of the algorithm transforms the array into a heap.

- We do this by inserting the numbers into the heap and having them trickle up...one number at a time.
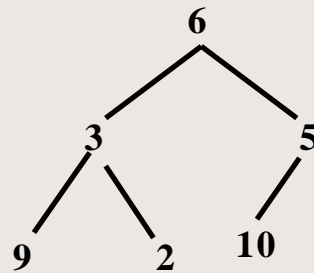
Heapsort

- A better approach is to put all of the numbers in a binary tree -- in the order you received them.

- Then, perform the algorithm we used to adjust the heap after deleting an item.

- This will cause the smaller numbers to trickle down to the bottom.

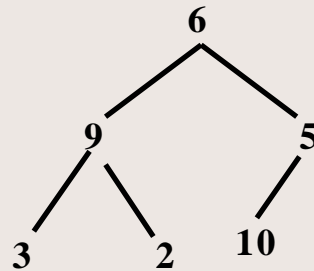# Heapsort

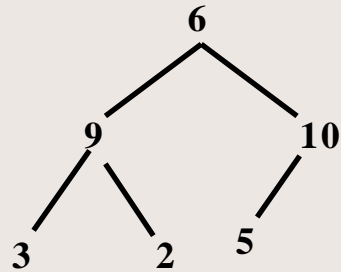6
3   5
9   2   10

Step 1

6
3   5
9   2   10

Step 2: compare 6 with 3 & 5; no swaps

6
9   5
3   2   10

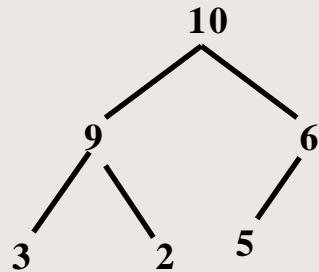**Step 3: compare 3 with 9 and 2; swap 3 and 9; this means that 3 & 2 are now in their correct sorted locations**

Heapsort

```
              6
            /   \
           9     10
          / \     /
         3   2   5
```

**Step 4: compare 5 with 10; swap;
this means that 5 is now in its correct
sorted location; Level 3 is sorted.**

```
             10
            /   \
           9     6
          / \     /
         3   2   5
```

**Pass 2: start again. Compare 6 with 3 and 10; swap
6 and 10. 3 and 6 are now in the correct sorted locations;
Level 2 is sorted.**

Deletion Algorithms

- In class, take 5 minutes and practice creating the following trees:
    - insert: 30, 50, 70, 10, 20, 60, 15, 25, 85
    - create: BST, AVL, 2-3, 2-3-4, red-black
    - create: a heap
    - now, delete a leaf in each tree
    - now, delete an internal node

Moving from Trees to Graphs

- Trees are useful when there is an inherent hierarchical relationship between our data
  - sorting data by a key can build such a relationship
- But, not all problems are so simple
- Maybe there are complex relationships between nodes ... not just hierarchical!
 - this is where graphs become important

Graphs

- Graphs can be used as data structures and as the last ADT we will learn about;
  - to represent how data can be related to one another...setting up an inherent structure.
- Graphs consist of vertices (nodes) and edges which connect the vertices.
- A path is a sequence of edges that can be taken to get from one vertex to another.
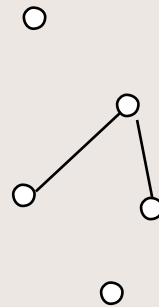
Graphs

- Paths may pass through the same vertex more than once but simple paths may not.

- A cycle is a path that starts and ends at the same vertex but does not pass through other vertices more than once.

- Graphs are considered to be connected if there is a path between every pair of vertices.
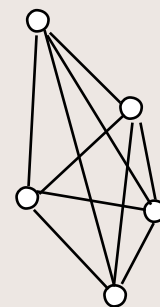
Graphs

- A graph is complete if there is an edge between every pair of vertices.

- Edges can have labels - or numeric values - which create a weighted graph.

- Weights can label the distances between cities.
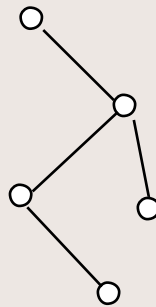
connected
graph

disconnected
graph

complete
graph

Graphs

- ADT Graph operations might include:
  - create an empty graph
  - determine if the graph is empty
  - insert a vertex
  - insert an edge between two vertices
  - delete a vertex
  - delete an edge between two vertices
  - retrieve the value of the data at a vertex
  - replace the data at a particular vertex
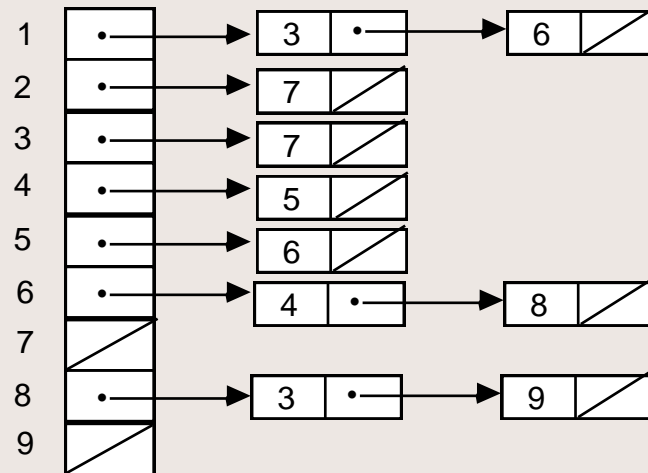 - determine if an edge is between vertices

Graphs

- One common way of implementing a graph is to use an adjacency list.

- Using this approach, imagine that the vertices are numbered 1,2, thru N.

- This type of implementation consists of N linked lists.

- There is a node in the ith linked list for the vertex j if and only if there is an edge from vertex i to vertex j.

Graphs
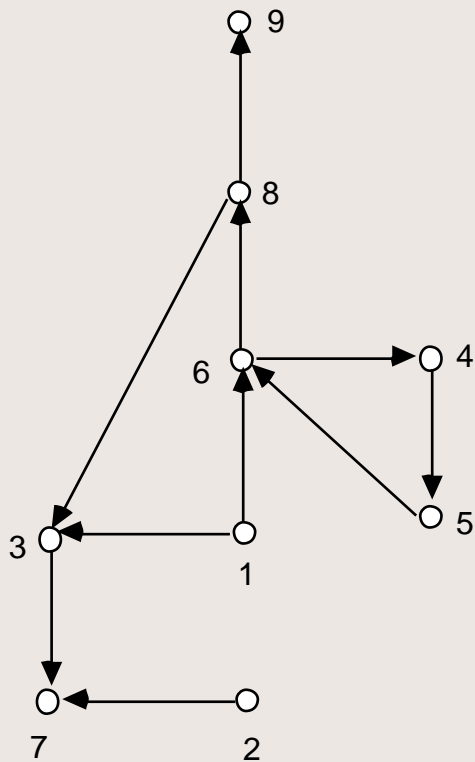
A directed graph

Graphs

- Two common operations are to find an edge between two vertices and to find all vertices adjacent to a given vertex.

- Using an adjacency list to determine whether there is an edge from one vertex to another, we must traverse the linked list associated with one of the vertices to determine whether the other vertex is present.

Graphs

- To determine all vertices adjacent to a given vertex, we only need to traverse the linked list associated with the specified vertex.

- What about traversing a graph?
  - Graph-traversal starts at a specified vertex and does not stop until it has visited all of the vertices that it can reach.
  - It visits all vertices that have a path between the specified vertex and the next.

Graphs

- Notice how different this is than tree traversal.
  - with tree traversal, we always visit all of the nodes in the tree
  - with graph traversal we do not necessarily visit all of the vertices in the graph unless the graph is connected.
  - If a graph is not connected, then a graph traversal that begins at a vertex will only visit a subset of the graphs vertices.

Graphs

- There are two basic graph-traversal algorithms that we will discuss.
- These algorithms visit the vertices in different orders,
  - but if they both start at the same vertex, they will visit the same set of vertices.
- Depth first search
- Breadth first search

Graphs

- Depth-First Search
  - This algorithm starts at a specified vertex and goes as deep into the graph as it can before backtracking.
  - This means that after visiting the starting vertex, it goes to an unvisited vertex that is adjacent to the one just visited.
  - The algorithm continues from this way until it has gotten as far as possible before returning to visit the next unvisited vertex adjacent to the starting place.

Graphs

- Depth-First Search
  - This traversal algorithm does not completely specify the order in which you should visit the vertices adjacent to a particular vertex.
  - One possibility is to visit the vertices in sorted order.
  - This only works - of course - when our nodes in each linked list of an adjacency list are in sorted order.

Graphs

- Depth-First Search
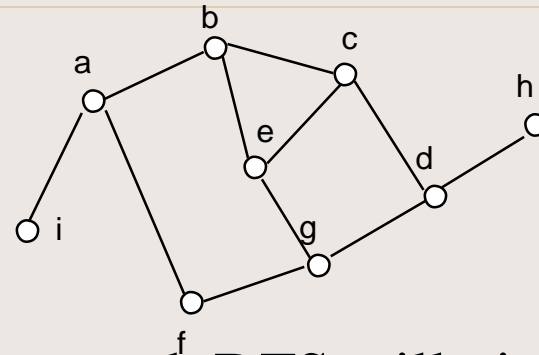
  – Because the graph is connected, DFS will visit every vertex. In fact, the traversal could visit the vertices in this order:

  a b c d g e f h i

  – Notice a stack of vertices can be used to implement this approach -- where the last one visited is

Graphs

- Breadth-First Search
  - This starts at a specified vertex and visits every vertex adjacent to that vertex before embarking from any of those vertices to the next set.
  - It does not embark from any of these vertices adjacent to the starting vertex until it has visited all possible vertices adjacent.
  - Notice that this is a first visited -- first explored strategy. Therefore, it should be obvious that a queue of vertices can be used

Graphs

- Breadth-First Search
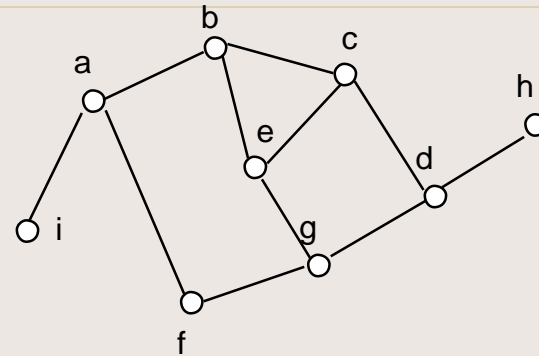
  – The the above graph, starting at vertex a we could traverse the vertices in the following order:

    a b f i c e g d h