CSE 12
Introduction to Tree Structures

- The PriorityQueue ADT
- PriorityQueue implementations
- Trees and tree terminology
- Binary trees
- Heaps and heap implementations
Priority Queue: a HighestPriority-In-First-Out (HPIFO) data structure

- the highest priority item in the structure will be the first item to be taken out of it

• Values: a collection of data items of type T, which are comparable to each other for ‘priority’

• Operations:
  - add(T)
  - T removeHighest()
  - T peekHighest()
  - int size()

**Introduction** these are the essential ones!
Examples of Priority Queue applications

• Operating systems schedulers: processes can be placed in a priority queue awaiting their turn to use the processor; the highest priority (most time-critical) process will be selected to run next.

• Many high-performance algorithms (Huffman coding, Dijkstra shortest path, Kruskal spanning tree, etc.) use a priority queue to decide what step to take next.

• Simulations of a priority-queue-like behavior (e.g., emergency room waiting area)
Priority Queue Properties and Attributes

Properties
• A PriorityQueue is a HPIFO collection.

  All elements in the collection must be comparable to each other with respect to a quantity called their ‘priority’.

  A remove operation on the collection removes the item with the highest priority of all items in the collection.

Attributes

  size : The number of elements in the priority queue
A picture of a priority queue

Highest priority item
The Priority Queue ADT

Operations

PriorityQueue

pre-condition: none
responsibilities: constructor—create an empty priority queue.
post-condition: size is 0

add( Type element )
pre-condition: element is comparable to other elements in the queue
responsibilities: insert element into the priority queue
post-condition: size is increased by 1

Type remove()
pre-condition: size is greater than 0
responsibilities: remove & return the highest priority element
post-condition: size is decreased by 1, highest priority element is removed
return: the highest priority element

Type peek() ... return the highest priority element
Implementing Priority Queue

- There are always many ways to implement a given ADT
- However, some ways are better than others!
- From its name, you might think a priority queue is a kind of queue, and so you should use the Inheritance pattern and implement a PriorityQueue class that extends Queue
- Or, you might think you should use the Adapter pattern, and define a PriorityQueue class that delegates to Queue
- Actually neither of these is a very good approach, and we won’t consider them
- (In fact, it would be better to go the other way, and define Stack or Queue by adapting PriorityQueue! )
Implementing Priority Queue

• We will first consider using a linked list to implement the Priority Queue ADT

• There are at least two ways to do that:
  1. add() is implemented so that the linked list is always sorted in order of priority; remove() and peek() then operate on the first element of the list
  2. add() is implemented to always add at the front of the list; remove() and peek() then need to traverse the list to find the highest priority element

• What are the time costs of each approach?
Linked-list Priority Queue time costs

• Assume a linked-list Priority Queue with N elements.

• Approach 1 (always sorted list):
  – add() has worst-case time cost ______
  – remove(),peek() have worst-case time cost ______

• Approach 2 (always add at front):
  – add() has worst-case time cost ______
  – remove(),peek() have worst-case time cost ______
Implementing Priority Queue

• A linked list *could* be used to implement the Priority Queue ADT... but it is much more common to use a *heap*

• A heap data structure is a kind of *binary tree*

• So let’s look at trees, binary trees, and heaps...
Tree structures

• A tree is a *hierarchical* structure
  – This is a generalization of a linear structure such as a list

• A tree is a set of elements called nodes, structured by a "parent" relation:
  – If the tree is nonempty, exactly one node in the set is the *root* of the tree: this is at the top of the hierarchy
    • The root of a tree is the only node that has no parent
  – Every node in the tree except the root has exactly one other node that is its parent
    • the nodes that have the root as parent are at the level one from the top in the hierarchy; etc.
Tree terminology

- **Children** of a node $P$: the nodes that have $P$ as parent

- **Root** of a tree: the unique node in the tree with no parent

- **Leaves** of a tree: the nodes with no children

- **Internal nodes** of a tree: the nodes that are not leaves
• In computer scienceland, trees are upside-down, with the root at the top and leaves at the bottom.

• In this tree...
  what is the root?
  what are the leaves?
  what are the internal nodes?
More Tree terminology

- **Descendant** of a node P:
  - If a node C is a child of P, then C is a descendant of P
  - If a node C is a child of a descendant of P, then C is a descendant of P

- **Ancestor** of a node C: if C is a descendant of P, then P is an ancestor of C

- **Path**: a sequence of unique nodes, going from parent-to-child

- **Path length**: the number of nodes contained in a path
Still more Tree terminology

• *Level* or *depth* of a node:
  – The level of the root is 1
  – The level of any non-root node is 1 + the level of its parent
  – (So the level of a node is equal to the length of the path from the root to the node)

• *Height* of a node: the length of the longest path from the node to a leaf

• Height of a *tree*: the height of the root of the tree
A note about height…

• (Note: the definitions of level, depth, and height given here use “1-based” counting. Sometimes you will see definitions that use “0-based” counting, where the root of the tree is defined to be at level 0, etc. For qualitative “big-O” analysis, this difference doesn’t matter; but for other purposes, it might!)
Examples of some tree concepts

<table>
<thead>
<tr>
<th>path</th>
<th>e₁ – e₃</th>
<th>length: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>e₁ – e₄ – e₈ – e₁₀</td>
<td>length: 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>depth</th>
<th>e₁</th>
<th>depth: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>e₂, e₃, e₄</td>
<td>depth: 2</td>
</tr>
<tr>
<td></td>
<td>e₅, e₆, e₇, e₈</td>
<td>depth: 3</td>
</tr>
<tr>
<td></td>
<td>e₉, e₁₀</td>
<td>depth: 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>height</th>
<th>e₁</th>
<th>height: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>e₃</td>
<td>height: 2</td>
</tr>
<tr>
<td></td>
<td>e₁₀</td>
<td>height: 1</td>
</tr>
</tbody>
</table>

| size                  | 10                      |

<table>
<thead>
<tr>
<th>ancestors of e₁₀:</th>
<th>e₈, e₄, e₁</th>
</tr>
</thead>
</table>

| descendants of e₄:    | e₈, e₉, e₁₀             |
Binary Trees

- A *binary tree* is a tree in which no node has more than 2 children: a node in a binary tree may have 0, 1, or 2 children only.
- Some examples of binary trees:
Binary trees: some special cases

(a) A **full** binary tree of height 3.

(b) **Complete** binary trees of height 3 and 4.

**A full binary tree** of height $h$ is one in which all nodes from level 1 through level $h - 1$ have two children.

**A complete binary tree** of height $h$ is one in which all nodes from level 1 through $h - 2$ have two children and all the children of nodes at level $h - 1$ are contiguous and to the left of the tree.
Binary Tree properties

We will derive these properties later, and just state them for now:

• A full binary tree of height \( h \) has \( 2^h - 1 \) nodes

• The height of a full binary tree with \( n \) nodes is \( \log_2(n+1) \)

• A complete binary tree of height \( h \) has between \( 2^{h-1} \) and \( 2^h - 1 \) nodes

• A height of a complete binary tree with \( n \) nodes is at most \( \log_2 n + 1 \)

• The height of a binary tree with \( n \) nodes is at most \( n \)
  • this “worst case” occurs when no node has more than 1 child: the tree is, essentially, just a linked list
To enable typical binary tree operations, a binary tree node should have these attributes:

- the data element to be stored in the node
- left child (possibly null)
- right child (possibly null)
- parent (possibly null)
Binary Tree Node implementations

- A binary tree node can be implemented as a dynamically created instance of a binary tree node class.

- To do that: define a class with instance variables corresponding to the attributes, and create instances of that class.

- However, it is also possible to use an array to implement a binary tree.

- In that case, the elements of the array hold data, and the child/parent attributes are implicit...
A full binary tree can be implemented using an array as shown: if the tree has height $h$, the array must have length $2^h - 1$.

Any binary tree of height $h$ could be implemented this way (we need some way to indicate which nodes are “missing”)

But this can be very space inefficient: it always uses $2^h - 1$ space, no matter the size of the tree... even if the size is only $h$
Array Implementation of Binary Tree

Navigating around the tree implemented using an array:

For a node at index $i$, its:
- left child is at index $2 \times i + 1$
- right child is at index $2 \times i + 2$
- parent is at index $(i - 1) / 2$

For a full tree with $n$ nodes, the interior nodes of the tree are all found at indexes $0$ through $n/2 - 1$, and the leaves at indexes $n/2$ through $n - 1$.

Need a way to indicate a “missing” node
A heap is a special kind of binary tree that has the following invariants:

- “Structural property”: A heap is a complete binary tree.
- “Ordering property”: A heap is either a \textit{minheap} or a \textit{maxheap}:
  
  \textit{minheap} – The data element in each node is \textit{less than} or equal to the elements in its descendants.
  
  \textit{maxheap} – The data element in each node is \textit{greater than} or equal to the elements in its descendants.

A total order must apply to data elements in a heap...they must be comparable to each other!
Valid and Invalid Minheaps

(a) A valid minheap.

(b) An invalid minheap (violates the structural property: 21’s left child is missing).

(c) An invalid minheap (violates the minheap ordering property: P is greater than B).
Adding an Element to a MinHeap

(a) A minheap prior to adding an element. The circle is where the new element will be put initially.

(b) Add the element, 6, as the new rightmost leaf. This maintains a complete binary tree, but may violate the minheap ordering property.

(c) “Bubble up” the new element. Starting with the new element, if the child is less than the parent, swap them. This moves the new element up the tree.

(d) Repeat the step described in (c) until the parent of the new element is less than or equal to the new element. The minheap invariants have been restored.
Adding an Element to a MinHeap

**Pseudocode: add( element)**
1. insert the new element as the new rightmost leaf on the bottom level
2. increment size by 1
3. bubbleUp(rightmost leaf)

**Pseudocode: bubbleUp(node) // recursive version**
1. if node is root of heap, return
2. if node's element is greater than or equal to node's parent's element, return
3. swap node and parent's elements
4. bubbleUp(parent)

The time cost of the add() operation is bounded by the distance the new element can travel.

Since the new element starts as a leaf and the highest it can climb is to the root, this bound is the height of the tree. In a complete binary tree this is \( O( \log n ) \), so the time cost of add() is \( O( \log n ) \) worst case.
Removing an Element from a MinHeap

(a) Moving the rightmost leaf to the top of the heap to fill the gap created when the top element (5) was removed. This is a complete binary tree, but the minheap ordering property has been violated.

(b) “Trickle down” the element. Swapping top with the smaller of its two children leaves top’s right subtree a valid heap. The subtree rooted at 18 still needs fixing.

(c) Last swap. The heap is fixed when 18 is less than or equal to both of its children. The minheap invariants have been restored.
Removing an Element from a MinHeap

Pseudocode: top() // remove and return the top element of the heap
1. extract the top element from the heap and store it in oldTop
2. move the rightmost leaf element into top
3. decrement size by 1
4. trickleDown( top ) // fix the heap starting at the top
5. return oldTop

Pseudocode: trickleDown( node ) // fix the heap rooted at node
1. let cursor reference node
2. while the cursor has a child and the cursor’s element is greater than the element in either child
3. // invariant: the cursor’s children are roots of valid heaps
4. swap the cursor’s element with the element of the smaller of cursor’s children
5. // invariant: the cursor and its children are a now valid heap
6. move cursor to the swapped child
7. // loop post-condition: the binary tree rooted at node is a valid heap

As with add(), the cost is bounded by the height of the tree: O(log n)
Implementing a Heap using an Array

• A heap can be implemented using dynamically created objects, with pointers linking the nodes...

• ...however, in practice, a heap is almost always implemented using an array

• For heaps, this is space efficient!

• Since a heap with $n$ elements is a complete binary tree, it can be stored in an array of length $n$ without wasting any space
Implementing a Heap using an Array

- Using an array implementation, all the heap operations are easy to implement with simple integer index computations:
  - The root is always at index 0
  - The rightmost leaf is always at index size-1
  - The parent of a node at index $i$ is at index $(i-1)/2$
  - The left child of a node at index $i$ is at index $2*i+1$
  - The right child of a node at index $i$ is at index $2*i+2$
Implementing Priority Queue using a heap

- A heap is a natural way to implement the PriorityQueue ADT
  - use a minheap if “small” elements have higher priority
  - use a maxheap if “large” elements have higher priority

- add(), remove() and peek() are implemented in terms of the corresponding heap operations

- What are the time costs of the PriorityQueue operations when implemented in this way?
Heap-based Priority Queue time costs

Assume a heap-based Priority Queue with N elements.

With this approach:
- add() has worst-case time cost _____
- remove() has worst-case time cost ______
- peek() has worst-case time cost ______

Compare to either linked-list-based approach.
Which is the best? Why?
Priority queues and sorting

• If you have a priority queue implementation, you could use it to sort an array of N items:

• Create an empty Priority Queue

• Iterate through the array, adding each of its N items to the Priority Queue

• Perform N remove operations on the Priority Queue, storing returned items in sequential positions in the array
Priority queue sorting time costs

• Assume a linked-list Priority Queue implementation.

• With a sorted list implementation ("Approach 1") of Priority Queue:
  – adding N items has worst-case time cost ______
  – removing N items has worst-case time cost ______
  – so, sorting N items has worst-case time cost ______

• With an unsorted list implementation ("Approach 2") of Priority Queue:
  – adding N items has worst-case time cost ______
  – removing N items has worst-case time cost ______
  – so, sorting N items has worst-case time cost ______
Priority queue sorting time costs

- Assume a heap-based Priority Queue implementation.

- With a heap implementation of Priority Queue:
  - adding $N$ items has worst-case time cost $\underline{\quad}$
  - removing $N$ items has worst-case time cost $\underline{\quad}$
  - so, sorting $N$ items has worst-case time cost $\underline{\quad}$

- Using a heap to implement a Priority Queue is the way to go!
Priority queue sorting space costs

- Using a priority queue to sort an array of length $N$:
  - The array takes space $O(N)$, and the priority queue takes space $O(N)$; so the total space cost is $O(N)$

- Q: But are two arrays (one for the array to be sorted, one for the heap implementing the priority queue) really needed?
- A: No; it is possible to perform the sort using only one array of length $N$

- This is the idea of *heapsort*
Heapsort

Consider an array to be sorted in nondecreasing order, and a maxheap implemented using an array.

We will insert the array elements, left to right, in the maxheap; then remove the maxheap elements, one at a time, placing them in the array right to left.
Heapsort: building the heap

1. 8
   12 2 10 6 4

2. 12 8
   2 10 6 4

3. 12 8 2
   10 6 4

4. 12 10 2 8
   6 4

5. 12 10 2 8 6
   4

6. 12 10 4 8 6 2
   2
Heapsort: removing from the heap

1. 10 8 4 2 6 12
   10 8 4 2 6 12
   8 6 4 2 12
   8 6 4 2 10 12
   6 2 4 10 12
   6 2 4 8 10 12

2. 4 2
   4 2
   6 8 10 12
   4 2
   6 8 10 12
   4 2

3. 2
   2
   4 6 8 10 12
   2
   4 6 8 10 12
   2

4. 6 8 10 12
   6 8 10 12
   6 8 10 12
   6 8 10 12
   6 8 10 12
   6 8 10 12

5. 2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12

6. 2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12
   2 4 6 8 10 12
Heapsort

- Note that at each step, the parts of the arrays “used” by the heap and “used” by the input/output array do not overlap.

- So, all the operations of building the heap from the input array, and removing from the heap to create the output array, can be done *in place* in the original array!
Heapsort: building the heap in place

1. 8 12 2 10 6 4

2. 12 8 2 10 6 4

3. 12 8 2 10 6 4

4. 12 10 2 8 6 4

5. 12 10 2 8 6 4

6. 12 10 4 8 6 2
Heapsort: removing from the heap in place

1. \(\begin{array}{cccccc}
10 & 8 & 4 & 2 & 6 & 12 \\
\end{array}\)

2. \(\begin{array}{cccccc}
8 & 6 & 4 & 2 & 10 & 12 \\
\end{array}\)

3. \(\begin{array}{cccccc}
6 & 2 & 4 & 8 & 10 & 12 \\
\end{array}\)

4. \(\begin{array}{cccccc}
4 & 2 & 6 & 8 & 10 & 12 \\
\end{array}\)

5. \(\begin{array}{cccccc}
2 & 4 & 6 & 8 & 10 & 12 \\
\end{array}\)

6. \(\begin{array}{cccccc}
2 & 4 & 6 & 8 & 10 & 12 \\
\end{array}\)
Resizeable arrays

- In Java, when an array is created, its length (number of elements) is specified, and cannot be changed.

- However, it is often useful to present an interface that makes it appear to the client that the capacity of an array is automatically increasing as needed.

- Let's look at the source code in `java/util/ArrayList.java` to see an example of how this can be done.
package java.util;

/**
 * Resizable-array implementation of the <tt>List</tt> interface.
 * Implements all optional list operations,
 * and permits all elements, including <tt>null</tt>.
 * Each <tt>ArrayList</tt> instance has a <i>capacity</i>. The capacity is
 * the size of the array used to store the elements in the list.
 * It is always at least as large as the list size.
 * As elements are added to an ArrayList, its capacity grows automatically.
 *
 * @param <E> Type parameter for the elements in the list.
 */

public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{

java.util.ArrayList

- Instance variables

```java
/**
 * The array buffer into which the elements of the ArrayList are stored.
 * The capacity of the ArrayList is the length of this array buffer.
 */
private Object[] elementData;

/**
 * The size of the ArrayList (the number of elements it contains).
 */
private int size;
```
java.util.ArrayList

• Constructors

```java
/**
 * Constructs an empty list with the specified initial capacity.
 *
 * @param   initialCapacity   the initial capacity of the list
 * @exception IllegalArgumentException if the specified initial capacity
 *            is negative
 */
public ArrayList(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                              initialCapacity);
    this.elementData = new Object[initialCapacity];
}

/**
 * Constructs an empty list with an initial capacity of ten.
 */
public ArrayList() {
    this(10);
}
```
java.util.ArrayList

• add() method

/**
 * Appends the specified element to the end of this list.
 * 
 * @param e element to be appended to this list
 * @return <tt>true</tt> (as specified by {@link Collection#add})
 */

public boolean add(E e) {
    ensureCapacity(size + 1);  // Can increase capacity
    elementData[size++] = e;
    return true;
}
ensureCapacity() method

```java
/**
 * Increases the capacity of this <tt>ArrayList</tt> instance, if
 * necessary, to ensure that it can hold at least the number of elements
 * specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
public void ensureCapacity(int minCapacity) {
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        elementData = new Object[newCapacity];
        for(int i=0; i<size; i++)
            elementData[i] = oldData[i];
    }
}
```
Next time

- Binary Tree Traversals
- Binary Search Tree ADT
- A Linked Implementation of Binary Search Tree
- Binary Search Tree Operation Time Costs

Reading: Gray, Ch 10