Parts of a binary tree

- A binary tree is composed of zero or more nodes.
- Each node contains:
  - A value (some sort of data item)
  - A reference or pointer to a left child (may be null), and
  - A reference or pointer to a right child (may be null)
- A binary tree may be empty (contain no nodes).
- If not empty, a binary tree has a root node:
  - Every node in the binary tree is reachable from the root node by a unique path.
- A node with neither a left child nor a right child is called a leaf.
  - In some binary trees, only the leaves contain a value.
Types of Binary Tree

- A Binary Tree is **Full Binary Tree** if every node has 0 or 2 children.
- Following are examples of **full binary tree**.
- We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.
- In a Full Binary, number of leaf nodes is number of internal nodes plus 1 ($L = I + 1$) where $L =$ Number of leaf nodes and $I =$ Number of internal nodes.
Types of Binary Tree

- A Binary Tree is **Complete Binary Tree** if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.

- A complete binary tree is one where all the levels are full with exception to the last level and it is filled from left to right.
Types of Binary Tree

- A Binary tree is **Perfect Binary Tree** in which all internal nodes have two children and all leaves are at same level.
- Following are examples of Perfect Binary Trees.
A balanced tree is a tree in which difference between heights of sub-trees of any node in the tree is not greater than one.
Binary Search Tree

• Stored keys must satisfy the *binary search tree* property.
  
  – ∀ y in left subtree of x, then \( \text{key}[y] \leq \text{key}[x] \).
  
  – ∀ y in right subtree of x, then \( \text{key}[y] \geq \text{key}[x] \).
Priority Queue in C++ STL

- Priority queues are a type of container adaptors, specifically designed such that the first element of the queue is the greatest of all elements in the queue.

- The functions associated with priority queue are:
  - `empty()` – Returns whether the queue is empty
  - `size()` – Returns the size of the queue
  - `top()` – Returns a reference to the top most element of the queue
  - `push(g)` – Adds the element ‘g’ at the end of the queue
  - `pop()` – Deletes the first element of the queue
```cpp
#include <iostream>
#include <queue>

using namespace std;

void showpq(priority_queue<int> gq)
{
    priority_queue<int> g = gq;
    while (!g.empty())
    {
        cout << 't' << g.top();
        g.pop();
    }
    cout << endl;
}

int main()
{
    priority_queue<int> gquiz;
    gquiz.push(10);
    gquiz.push(30);
    gquiz.push(20);
    gquiz.push(5);
    gquiz.push(1);

    cout << "The priority queue gquiz is: ";
    showpq(gquiz);

    cout << "gquiz.size(): " << gquiz.size();
    cout << "gquiz.top(): " << gquiz.top();

    cout << "gquiz.pop(): ";
    gquiz.pop();
    showpq(gquiz);

    return 0;
}
```

The priority queue gquiz is: 30 20 10 5 1

gquiz.size(): 5
gquiz.top(): 30
gquiz.pop(): 30 20 10 5 1
Priority Queue ADT

• A priority queue stores a collection of items

• An item is a pair (key, element)

• Main methods of the Priority Queue ADT
  • `insertItem(k, o)` inserts an item with key k and element o
  • `removeMin()` removes the item with the smallest key

• Additional methods
  • `minKey()` returns, but does not remove, the smallest key of an item
  • `minElement()` returns, but does not remove, the element of an item with smallest key
  • `size()`, `isEmpty()`

• Applications:
  • Standby flyers
  • Auctions
Comparator ADT

- A *comparator* encapsulates the action of comparing two objects according to a given total order relation.
- A generic priority queue uses a comparator as a template argument, to define the comparison function ($<$, $=$, $>$).
- The comparator is external to the keys being compared. Thus, the same objects can be sorted in different ways by using different comparators.
- When the priority queue needs to compare two keys, it uses its comparator.
Using Comparators in C++

• A comparator class overloads the “()” operator with a comparison function.

• Example: Compare two points in the plane lexicographically.

```cpp
class LexCompare {
public:
    int operator()(Point a, Point b) {
        if (a.x < b.x) return -1;
        else if (a.x > b.x) return +1;
        else if (a.y < b.y) return -1;
        else if (a.y > b.y) return +1;
        else return 0;
    }
};
```
**Example 8.4**: The following table shows a series of operations and their effects on an initially empty priority queue $P$. Each element consists of an integer, which we assume to be sorted according to the natural ordering of the integers. Note that each call to `min()` returns a reference to an entry in the queue, not the actual value. Although the “Priority Queue” column shows the items in sorted order, the priority queue need not store elements in this order.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Output</th>
<th>Priority Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(5)</code></td>
<td>–</td>
<td>{5}</td>
</tr>
<tr>
<td><code>insert(9)</code></td>
<td>–</td>
<td>{5, 9}</td>
</tr>
<tr>
<td><code>insert(2)</code></td>
<td>–</td>
<td>{2, 5, 9}</td>
</tr>
<tr>
<td><code>insert(7)</code></td>
<td>–</td>
<td>{2, 5, 7, 9}</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>[2]</td>
<td>{2, 5, 7, 9}</td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>–</td>
<td>{5, 7, 9}</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>3</td>
<td>{5, 7, 9}</td>
</tr>
<tr>
<td><code>min()</code></td>
<td>[5]</td>
<td>{5, 7, 9}</td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>–</td>
<td>{7, 9}</td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>–</td>
<td>{9}</td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>–</td>
<td>{}</td>
</tr>
<tr>
<td><code>empty()</code></td>
<td><code>true</code></td>
<td>{}</td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>“error”</td>
<td>{}</td>
</tr>
</tbody>
</table>
List-based Priority Queue

• **Unsorted list implementation**
  - Store the items of the priority queue in a list-based sequence, in arbitrary order
  
  4 — 5 — 2 — 3 — 1

• **Performance:**
  - `insertItem` takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - `removeMin`, `minKey` and `minElement` take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

• **sorted list implementation**
  - Store the items of the priority queue in a sequence, sorted by key
  
  1 — 2 — 3 — 4 — 5

• **Performance:**
  - `insertItem` takes $O(n)$ time since we have to find the place where to insert the item
  - `removeMin`, `minKey` and `minElement` take $O(1)$ time since the smallest key is at the beginning of the sequence
C++ Priority Queue Interface

```cpp
template <typename E, typename C>
class PriorityQueue {
public:
    int size() const; // number of elements
    bool isEmpty() const; // is the queue empty?
    void insert(const E& e); // insert element
    const E& min() const throw(QueueEmpty); // minimum element
    void removeMin() throw(QueueEmpty); // remove minimum
};
```

**Code Fragment 8.4:** An informal PriorityQueue interface (not a complete class).
Another important application of a priority queue is sorting, where we are given a collection $L$ of $n$ elements that can be compared according to a total order relation, and we want to rearrange them in increasing order (or at least in nondecreasing order if there are ties).

The algorithm for sorting $L$ with a priority queue $Q$, called `PriorityQueueSort`, is quite simple and consists of the following two phases:

- In the first phase, we put the elements of $L$ into an initially empty priority queue $P$ through a series of $n$ insert operations, one for each element.
- In the second phase, we extract the elements from $P$ in nondecreasing order by means of a series of $n$ combinations of `min` and `removeMin` operations, putting them back into $L$ in order.
Implementing a Priority Queue with a List

- Next Lecture