Today …

• Homework
  – Homework 5 due today
  – Homework 6

• Sorting Exercises (Mergesort)

• Start Binary Search Trees
  – Ch 10: pp. 499-516, 519-546
Mergesort (based on textbook)

```c
void Mergesort(Entry theArray[], int n, int first, int last) {
    if(first < last) {
        int mid = (first + last) / 2;
        Mergesort(theArray, n, first, mid);
        Mergesort(theArray, n, mid + 1, last);
        Merge(theArray, n, first, mid, last);
    }
}
```

Mergesort (based on textbook)

```c
void Merge(Entry theArray[], int n, int first, int mid, int last) {
    Entry tmpArray[n];
    int first1 = first, first2 = mid + 1, i = first1;
    for(; first1 <= mid && first2 <= last; i++) {
        if(theArray[first1] < theArray[first2])
            tmpArray[i] = theArray[first1++];
        else
            tmpArray[i] = theArray[first2++];
    }
    for(; first1 <= mid; i++, first1++)   // copy remaining from first half
        tmpArray[i] = theArray[first1];
    for(; first2 <= last; i++, first2++)  // copy remaining from sec. half
        tmpArray[i] = theArray[first2];
    for(int j = first; j <= last; j++)   // copy tmpArray to the array
        theArray[j] = tmpArray[j];
}
```
Binary Search Trees

We’ll start with some basic terminology

Next week we’ll dive into more details

Tree Structures

A tree is a “non linear” data structure ...

A list is a “linear” structure

Tree with the same nodes
Trees are used to represent different kinds of relationships

For example, inheritance hierarchies (taxonomies, ontologies, ...)

Algebraic expressions (evaluation order)

Trees are used to represent different kinds of relationships

Nesting (including “has a” relationships)

Sort order
Trees (Terminology)

In general, a tree forms a *hierarchy* consisting of

- Zero or more nodes
- A distinguished “*root*” node (no parents)
- Structures of multiple trees are called “*forests*”

Nodes are arranged in *parent-child* relationships

- The parent of a node *N* is directly above *N*
- The child of a node *N* is directly below *N*
Trees (Terminology)

Each non-root node has exactly one parent

Every node can have zero or more children
  – A node without any children is called a “leaf” node
  – A node with children is called an “internal” node

A “path” is a sequence of nodes …
  – The next node in the sequence after node \( N \) is a child of \( N \)
  – we can obtain a path by starting at a node and following child edges until we reach a leaf
Trees (Terminology)

Parent-child relationships induce *ancestor-descendent* relationships

- An *ancestor* of a node $N$ is either the parent of $N$ or the parent of an ancestor of $N$
- The root is the ancestor of every node in the tree
- A *descendant* of a node $N$ is either a child of $N$ or a child of a descendent of $N$

A descendent of a node lies on a path from the node to a leaf

Trees (Terminology)

Nodes can have zero or more “siblings”

- The siblings of $N$ are the nodes with the same parent as $N$
Trees (Terminology)

A “subtree” is a tree rooted at a descendent of the root

- A subtree includes all children and descendents of the subtree “root” \( N \)
- We often say a subtree is rooted at \( N \)

\[ \text{subtree rooted at B} \]

\[ \text{subtree rooted at G} \]

Trees (Terminology)

In a “binary” tree, every node has at most 2 children

More formally, \( T \) is a binary tree if

- \( T \) has no nodes (is empty), or
- \( T \) has the form

\[ \begin{array}{c}
\text{r} \\
/ \\
T_L \quad T_R
\end{array} \]

- where \( r \) is a node, and \( T_L \) and \( T_R \) are binary trees
- \( T_L \) is the left subtree of \( r \) and \( T_R \) is the right subtree of \( r \)

... note that this definition employs recursion! (this will be helpful as we compute over trees)
Trees (Terminology)

The “height” of a tree is the length of the longest path

– The number of nodes that lie on the longest path from the root to a leaf

Trees (Terminology)

An “empty” binary tree has no nodes
**Trees (Terminology)**

A “**full**” binary tree has a height $h$ with no missing nodes
– i.e., every internal node has **exactly 2** children

![Diagram of a full binary tree]

**Trees (Terminology)**

A “**complete**” binary tree of height $h$ is
– a full binary tree at height $h - 1$, and
– the nodes at height $h$ are filled in from left to right

… sometimes we say “**level**” $h$ to mean the nodes at height $h$

![Diagram of a complete binary tree]
Trees (Terminology)

A “balanced” binary tree has for every node

– left and right subtrees that differ in height by at most 1

Binary Tree versus Binary Search Tree (BST)

• A Binary Search Tree stipulates where nodes are placed in a Binary Tree ...

• In such a way as to maintain the items in sorted order (based on a sort key)
Binary Search Trees

The order constraint:
• For each node \( n \)

  if node \( n_l \) is in the left subtree of \( n \)
  then \( n_l < n \)

  if node \( n_r \) is in the right subtree of \( n \)
  then \( n < n_r \)

*note that a Binary Search Tree is a well-formed Binary Tree*

---

Pointer-Based Binary Search Trees

Nodes consist of values plus left and right child pointers
Searching for nodes in a BST

Given a key (i.e., a word) to search for …

• Start at the root node
  – Is the item at the root node equal to search key?
    If Yes … Found match!
  – Is the item at the root node greater than the search key?
    If Yes … Search the left subtree of the root
  – Is the item at the root node less than the search key?
    If Yes … Search the right subtree of the root

Inserting Nodes

Basic idea

• Find where the node should be (search)
• And then insert the node at that location
• Will always insert into a leaf node

Order of insertions determines the shape of the tree!

![Diagram of a binary search tree](image)
Inserting Nodes

Insert:
1. D
2. B
3. G
4. A
5. C
6. E
7. F
8. H

What happens?

Inserting Nodes

• Insert:
  1. A
  2. B
  3. C
  4. D
  5. E
  6. F
  7. G

What happens?
Inserting Nodes

Insert:
1. G
2. F
3. E
4. D
5. C
6. B
7. A

What happens?

Insertion order can generate entirely different tree “shapes”