Recursive Functions

Class 15

Functions and Procedures

- section 3.2
- your author makes a big distinction between functions and procedures
- this comes from the Pascal and Ada line of languages
- not as big a distinction in the C and C++ family

- function: arguments are not changed; a value is returned
  examples: `sqrt(x)` `floor(x)` `gcd(m, n)`
- procedure: may or may not be a return value; if so it is via changed arguments
  examples: `print(s)` `swap(a, b)`

Recursively Defined Functions and Procedures

- following directly from the concept of inductively defined sets of 3.1
- defined in terms of an inductively defined set
- a recursive function or procedure either generates the members of an inductively defined set or processes the members of an inductively defined set

Natural Numbers

- the natural numbers are an inductively defined set
  - Basis: $0 \in \mathrm{N}$
  - Induction: if $n \in \mathrm{N}$ then $n + 1 \in \mathrm{N}$
- suppose we want the sum of the first n natural numbers
- we can write a recursive function to process the first n elements of the inductively defined set

```
unsigned sum(unsigned n)
{
  if (n == 0)
  {
    return 0;
  }
  return n + sum(n-1);
}
```

- notice the inductive definition moves forward $(n + 1)$
- the recursive definition moves backwards $(n - 1)$

String Complement

- the section moves away from explicitly defining an inductive set
- focuses on recursive functions

- we have a string over $\{a, b\}$ for example *ababbba*
- we want its complement, in this case *babaaab*

- this could be defined iteratively
- but we wish to explore it recursively

String Complement

- Basis: the complement of the empty string is the empty string
- Recursion: a string of the form as: $\text{comp}(as) = b\,\text{comp}(s)$
  a string of the form bs: $\text{comp}(bs) = a\,\text{comp}(s)$

```
string comp(s)
{
  if (s.length() == 0)
  {
    return "";
  }
  if (s.at(0) == 'a')
  {
    return "b" + comp(s.substr(1));
  }
  return "a" + comp(s.substr(1));
}
```

String Prefix

- given two strings, a common problem is to find their longest common prefix
- the longest common prefix of "monkey" and "money" is "mon"
  the longest common prefix of "super" and "superb" is "super"
- for strings s and t there are four cases
  1. $s = \Lambda$: the prefix is Λ (base case)
  2. $t = \Lambda$: the prefix is Λ (base case)
  3. $s[0] \neq t[0]$: the prefix is Λ (base case)
  4. $s[0] = t[0]$: the prefix is $s[0] + \text{prefix}(s[1,], t[1,])$

see code

Sorting a List

- insertion sort is the name of a general class of sorting algorithm
- it is a very natural, intuitive way to sort a list of things
- it depends on inserting one item into a list that is already sorted
- the one new item is inserted into the correct spot, resulting in a list that is one longer, also sorted
- the key operation is insert, not sorting per se

Inserting Into a Sorted List

- arguments: an item to insert, and a list in which to insert it
- precondition: the list is sorted
- postcondition: the list contains the element, and is sorted

- Basis: if the list is empty, the item is added to the front of the list
- Basis: if the item is smaller than or equal to the head element, the item is added to the front of the list
- Recursion: the original head is prepended to the result of inserting the item into the tail of the list

see code

Tree Terminology

- tree
- node
- root
- edge
- child
- parent
- leaf
- sibling
- path
- path length
- depth
- height
- ancestor
- descendant
- proper ancestor
- proper descendant
- traversal

Tree

a tree is a connected graph

$$0 \text{ edges and } 0 \text{ nodes}$$

or

$$\text{any two} \begin{cases} \text{acyclic} \\ n \text{ nodes} \\ n-1 \text{ edges} \end{cases}$$

- most of our trees will be rooted
- a distinguished node root (possibly nullptr)
- directed, with a unique path from the root to every other node

Binary Trees

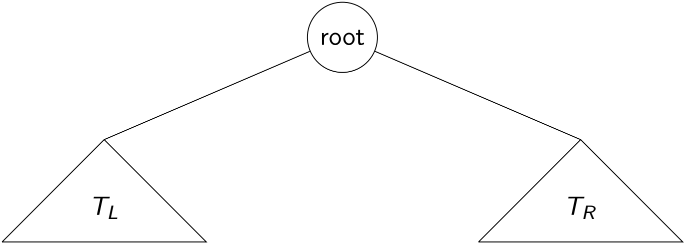
- by far the most important tree in CS is the binary tree
- every node has exactly two children (either of which may be null)

Implementation

```
class tree_node
{
  Object data;
  tree_node* left_child;
  tree_node* right_child;
};
```

Visualizing

- a binary tree consists of
  - a root, empty or an object containing data
  - a left child which is a binary tree
  - a right child which is a binary tree



Traversals

- traversal: "visiting" every node in the tree exactly once
- always starting at the root
- three important tree traversal types
  - preorder
  - inorder
  - postorder

Traversals

preorder

1. visit the root
2. traverse children left & right

inorder

1. traverse left child
2. visit the root
3. traverse right child

postorder

1. traverse children left & right
2. visit the root

Preorder Traversal

```
void preorder(tree_node* node)
{
  if (node != nullptr)
  {
    visit(node);
    preorder(node->left_child);
    preorder(node->right_child);
  }
}
```

Postorder Traversal

```
void postorder(tree_node* node)
{
  if (node != nullptr)
  {
    postorder(node->left_child);
    postorder(node->right_child);
    visit(node);
  }
}
```

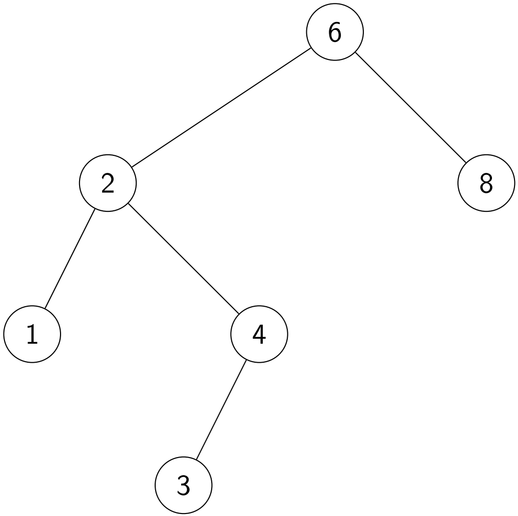
Inorder Traversal

```
void inorder(tree_node* node)
{
  if (node != nullptr)
  {
    inorder(node->left_child);
    visit(node);
    inorder(node->right_child);
  }
}
```

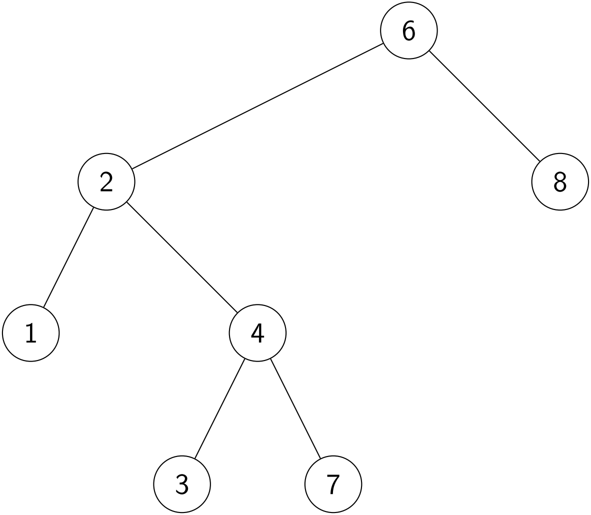
BST

- if we impose a couple of additional conditions, we get the binary search tree
  1. the data is of a Comparable type
  2. the data in a node is greater than any value in its left child subtree
  3. the data in a node is less than any value in its right child subtree
- simplifying assumption: there are no duplicate values in the tree

Example BST



Not a BST



BST Inorder

- note that an inorder traversal of a BST always produces a sequence of visits in strict order