

For Friday

- Read Weiss, chapter 6, sections 1-3
- Homework
 - Weiss, chapter 3, exercises 4-5. Use the C++ STL list class (and iterators). I only want to see the required functions. Note that you are returning lists without destroying the old ones. Your answers need to be efficient.

Programming Assignment 1

- Any questions?

Linked List Variations

- Empty head node

Stacks

- What is a stack?
- What would a stack ADT look like?
- How could you implement a stack?
- What are stacks used for?

Stack ADT

- **AbstractDataType** *Stack* {
 instances
 linear list of elements; one end is
 the *top*
 operations
 Create()
 Destroy()
 IsEmpty()
 IsFull() (not always needed)
 Top()
 Push(element)
 Pop()
}

Queues

- What is a queue?
- What would a queue ADT look like?
- How could you implement a queue?
- What are queues used for?

Queue ADT

- **AbstractDataType** *Queue* {
 instances
 linear list of elements; one end is
 the *front*, the other the *rear*
 operations
 Create()
 Destroy()
 IsEmpty()
 IsFull() (not always needed)
 Front()
 Add(elem) (or Put(elem) or Enqueue(elem))
 Delete() (or Get() or Dequeue())
}

A Note on Push and Pop

- In this class, do **not** use push and pop to refer to queue operations.
- They should **only** apply to stack operations.

Queues and Stacks

- Often used in similar contexts to achieve different desired results.
- Often used in conjunction with other data structures.

Trees

- Hierarchical structure
- Single **root**
- Each **node** has zero or more **children**
- Each node except the root has exactly one **parent**

Tree Definition

- A tree t is a finite nonempty set of elements. One of these elements is called the root, and the remaining elements (if any) are partitioned into trees which are called the subtrees of t .

Tree Vocabulary

- Root
- Child
- Parent
- Sibling
- Grandchild
- Grandparent
- Ancestor
- Descendent
- Leaf
- Subtree
- Forest
- Degree
- Level
- Height

Tree properties

- There is exactly one path connecting any two nodes in a tree.
 - Least common ancestor
 - Any node **could** be a root.
- A tree with N nodes has $N-1$ edges.

Binary trees

- Definition:
 - A binary tree t is a finite (possibly empty) collection of elements. When the binary tree is not empty, it has a root element and the remaining elements (if any) are partitioned into two binary trees, which are called the left and right subtrees of t .
- Differences from trees:
 - Each non-empty node has exactly two subtrees
 - Binary tree may be empty
 - The subtrees are ordered

Binary Expression Trees

- We can use binary trees to represent arithmetic expressions.
- Tree determines the order operations are executed in
- No need for parentheses

Binary Tree Properties

- Shares the tree properties.
- A binary tree of height h , $h \geq 0$, has at least h and at most $2^h - 1$ elements in it.
- The height of a binary tree that contains n , $n \geq 0$, elements is at most n and at least the ceiling of $\log_2(n+1)$.

Special Cases

- Full binary tree
 - A binary tree of height h that contains exactly $2^h - 1$ elements
 - In other words, if one element is added to the tree, the height must increase
- Complete binary tree

Linked Binary Trees

- A node is represented as a struct or a class
- Each node has two pointers to other nodes
- One pointer is to the Left child; the other is to the Right child
- An empty subtree is represented by a null pointer
- Sometimes it is convenient to include a pointer to the node's parent

Representation of Binary Trees

- We can represent binary trees in arrays
- We don't use index 0
- The root is at index 1
- Node i 's children are at indexes $2i$ and $2i+1$
- Advantages?
- Disadvantages?

Binary Tree Traversal

- In a binary tree **traversal**, we **visit** each node in the tree exactly once
- There are four different orders in which we often choose to visit the nodes
 - Preorder
 - Inorder
 - Postorder
 - Level order

Preorder Traversal

- ```
void PreOrder(BinTreeNode* tree)
{ // PreOrder traversal of tree
 if (tree) {
 Visit(tree); // visit the root
 PreOrder(tree->left) // do left
subtree
 PreOrder(tree->right) // do right
subtree
 }
```

# Inorder Traversal

- ```
void InOrder(BinTreeNode* tree)
{ // InOrder traversal of tree
  if (tree) {
    InOrder(tree->left) // do left
subtree
    Visit(tree); // visit the root
    InOrder(tree->right) // do right
subtree
  }
```

Postorder Traversal

- ```
void PostOrder(BinTreeNode* tree)
{ // PostOrder traversal of tree
 if (tree) {
 PostOrder(tree->left) // do left subtree
 PostOrder(tree->right) // do right subtree
 Visit(tree); // visit the root
 }
}
```

# Level Order Traversal

```
void LevelOrder(BinTreeNode* tree)
{ // PreOrder traversal of tree
 Queue q;
 while(tree) {
 Visit(tree); // visit tree
 // put children on the queue
 if (tree->left) q.Add(tree->left);
 if (tree->right) q.Add(tree->right);
 // get next node to visit
 if (q.IsEmpty())
 tree = NULL;
 else
 tree = q.Delete();
 }
}
```