Chapter 15: Tree ADTs

1. Tree: tree is either empty or it is a node (called the root) that has an ordered set of zero or more trees.

In **list nodes**, Tree nodes contain both client data and linkage information. Client data is handled as a template parameter as usual.

When the ordered set of trees associated with a root node is not empty, its elements called subtrees or child trees. (subtrees can have their own child subtree.)

Tree nodes with no children are called **terminal (leaf) nodes**.

Tree nodes with at least one no-empty child are called **nonterminal nodes**.

In graph terminology, nodes called **vertices (ding dian),** parent to child links are called **edges (lian xian)**.

**Level:**  each node in a tree must be on a certain level of the tree. The root node is at level 1; the level of a child node is one greater than that of its parent.

**Height:** the height of a tree is zero if the tree is empty; otherwise it is equal to the maximum level if any terminal node in the tree.

An **n-ary Tree** is a tree which each node as exactly n children.

A **Binary Tree** is an n-ary tree with n=2.

* Can refer to two children as “left child” and “right child”
* Any general tree can be transform as a binary tree. “Left” -🡪 “first child”

“Right”🡪 “**next sibling**”

**Sibling:** children with same parent.

Sample Application: Algebraic Expression

* Nonterminal nodes: operators
* Terminal nodes: operands

**Full Binary Tree:**  full binary tree of height **h**, all nodes that are a level less than h has two children each. A full binary tree has no missing nodes.

**Complete Binary Tree:** complete binary tree of height **h** is a binary tree that is full down to (h-1) level, with level ***h*** filled from left to right.

**Balanced Binary Tree:** a binary tree that the left and right subtree of any node have heights that differ by **no more than 1**.

1. **Tree Traversal**:

* Preorder: Visit the data in the **root node**; traverse **left subtree**; traverse **right subtree**.
* Inorder: Traverse **left subtree**; visit the data in the **root node**; traverse **right subree**.
* Postoder: Traverse **left subtree**; traverse **right subtree**; visit the data in the **root node**.

**4. Binary Search Tree:** it is a binary tree which sorted according to the values in its node.

* Node n’s value is greater than all values in its left subtree TL
* Node n’s value is less than all values in its right subtree TR
* Both TL and TR is binary search tree.

5.

**Chapter 16:**

1. Given h header files, be able to write **the traversal code** for Preoder, Inorder, Postorder code:

//public **PreorderTraverse method** calls protected **preoder funcion**

template<typename ItemType>

void BinaryNodeTree<ItemType>::preorderTravrse(void visit(ItemType**&**)) const

{

preorder(visit, rootPtr);

}

//**protected preorder** function !!!!!!!!!!!!!!

template<typename ItemType>

void BinayTreeNode<ItemType>::preorder(void visit(ItemType**&**), BianryNode<ItemType>\* **treePtr**) const

{

if(treePtr !==nullptr)

{ //visit the data in the root node

ItemType theItem = treePtr->getItem();

visit(theItem);

//traverse left subtree

preoder(visit, treePtr->getLeftChildPtr());

//traverse right subtree

preorder(visit, treePtr->getRightChildPtr());

}

}

Inorder, PostOrder traverse codes are similar to the preorder traverse. Just change a little!!!

**Interlude 4:**

1. Public inheritance:

**Chapter 17: Heap**

A **heap** is a **complete binary** tree that either is empty or whose root contains a value greater than or equal to the value in each of its children and has heap as its subtree.

1. Max heap: the root contains the item with the largest value.

Min heap: the root contains the item with the smallest value.

1. **Array-based** Heap operation:

Heap **deletion** (Remove **the root** of heap): remove();

1. With given heap, **copy** item in **last node to root**

---------🡪Semiheap

2 **root trickle down until it reaches the correct position**, that is, the item will come to the position where it would be greater or equal to the item in each of its children.

**Complexity:** O(log n)

Graph: Book P 510

Heap Add: Add(newData);

1. A new item is **inserted at the bottom of the tree**,
2. Then it **trickles up** to its correct position.

**Complexity**: O(log n)

Graph: Book P 511

1. Min-heap and Max-heap with given a series of insertion and deletion.
2. ……….
3. **Heap Sort**: it converts an array into a heap to locate the array’s largest item.
4. Transform the given array **into a heap**.
5. Swap the root with the last node in heap, and sign the root to sorted array, and previous array size -1.
6. Transform the new array into a heap, then swap first with the last item, sign the largest to sorted region.
7. Repeat 1~2.

Complexity of **Heap Sort**: it is similar to merge sort

Best Case: O(NlogN)

Average Case: O(NlogN)

Worst Case:O(NlogN)

**Advantage** than **Merge Sort**: do not need second array. ( Merge sort need a temp-Array)

**Chapter 18:**

1. **Hashing**: when we take any type of data, and assign an integer value to it. We will store the data in an array which we can access based on that “key”, which is integer. It takes constant time **O(1)** search in case of insertion, deletion and find operation.
2. **Hashing function** *h(****x****, N)*: it is something that takes the data as a parameter, and return the integer value that is the “key”.
3. **Collision**: many different items may end up the same “key” (interger).

**Collision resolution strategies**: Collision can be solved by making a linked list (separate chaining) or something like that to store all of the items that have the same key.

1. Collision resolution strategies: 1. Open Addressing

- Linear Probing

-Quadratic probing

2. Separate chaining (Linked list): Load factor ~~1 insertion success.

5. How to **delete** an item from a hash table: find and remove the item; 2. Go to next bucket, if empty then quit, if bucket is full and then delete the item in bucket and re-add it; repeat step2.??????????