CSC211 · Algorithms and Data Structures

17. Binary Trees and Representation

Level-order insertion, the array representation, and the index math that connects them

COMSATS University Islamabad, Lahore Campus

Agenda

1. The Binary Tree ADT, formalized as an interface
2. Level-order insertion, and why it needs a queue
3. A complete, working BinaryTree class
4. The array representation: level-order insertion as plain push_back
5. Index math — 2i+1, 2i+2, (i-1)/2 — and its one pitfall
6. Why array representation wastes space on anything but a complete tree

The Binary Tree ADT

Operation	Meaning
insert(value)	Add a new node holding value at the next available position
isEmpty()	Report whether the tree has any nodes
height()	Report the tree's height
Traversal operations	Visit every node in some defined order (Lecture 19)

i **“Next available position”** means scanning the tree level by level, left to right — a breadth-first walk.

Level-Order Insertion Needs a Queue

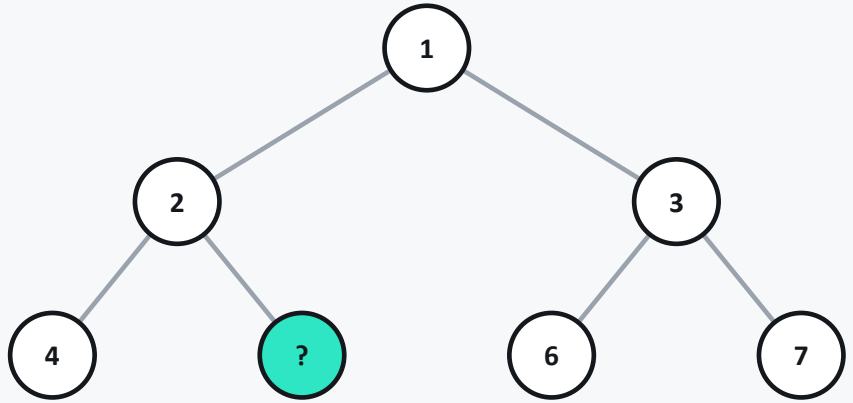
Queue:
[root]

Dequeue a node.
Left child free? → insert there.
Right child free? → insert there.
Otherwise enqueue both children
and keep going.

Insert new node here

Re-enqueue and
continue scanning

Breadth-first, not depth-first: you must finish checking every node at the current level before moving deeper — exactly what a queue's FIFO order gives you, and a stack cannot.



teal = next open slot

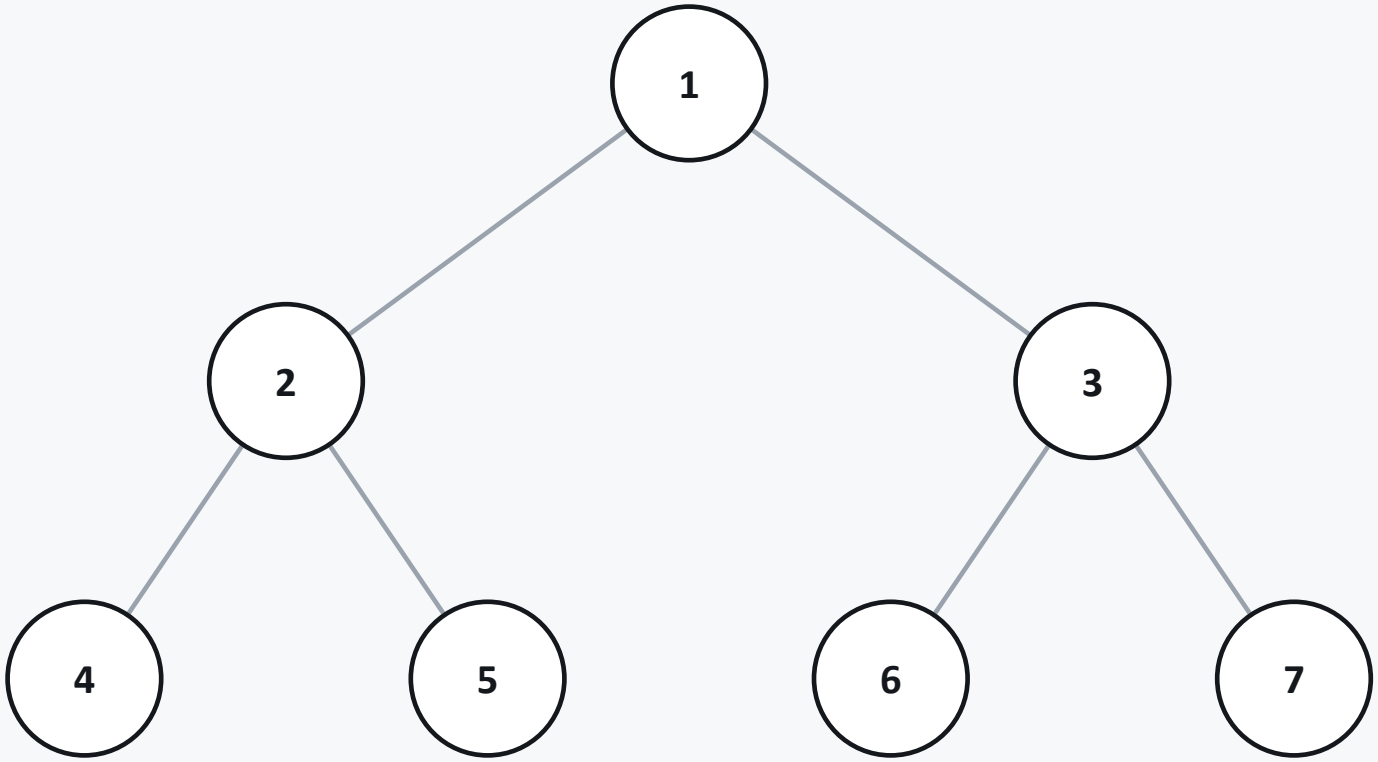
insert(): Level-Order via a Queue

binary_tree.cpp — BinaryTree::insert

```
void insert(int value) {
    TreeNode* newNode = new TreeNode(value);
    if (root == nullptr) { root = newNode; return; }
    queue<TreeNode*> toVisit;
    toVisit.push(root);
    while (!toVisit.empty()) {
        TreeNode* current = toVisit.front(); toVisit.pop();
        if (current->left == nullptr) { current->left = newNode; return; }
        else toVisit.push(current->left);
        if (current->right == nullptr) { current->right = newNode; return; }
        else toVisit.push(current->right);
    }
}
```

Highlighted lines: a full child gets re-enqueued so its own children are checked next.

The Result: A Complete Tree



2

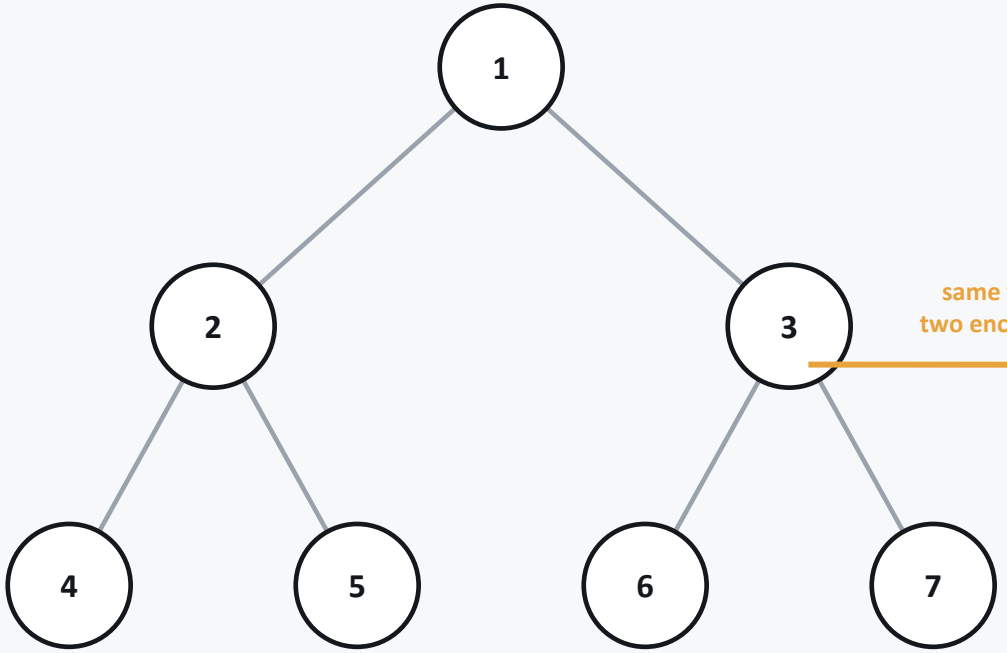
height() — 3 levels, root at height 0

0 gaps

level 2 is fully packed: no missing slots

Inserting 1..7 in order, level by level, left to right

Array Representation — Same Tree, No Queue



Linked representation

same tree,
two encodings

[0]	1
[1]	2
[2]	3
[3]	4
[4]	5
[5]	6
[6]	7

Array representation (level order)

insert() on the Array: Just push_back

array_binary_tree.cpp

```
vector<int> nodes;  // level-order storage

void insert(int value) {
    nodes.push_back(value);
}
```

The vector already IS the tree, stored level by level. The next open slot in level order is always the next index — no searching required.

O(n)

linked insert — rediscovers the slot by walking the tree

O(1)

array insert — the slot is always nodes.size()

Same operation, same result — the array never has to search for the slot at all.

The Index Math

left child

```
2 · i + 1
```

right child

```
2 · i + 2
```

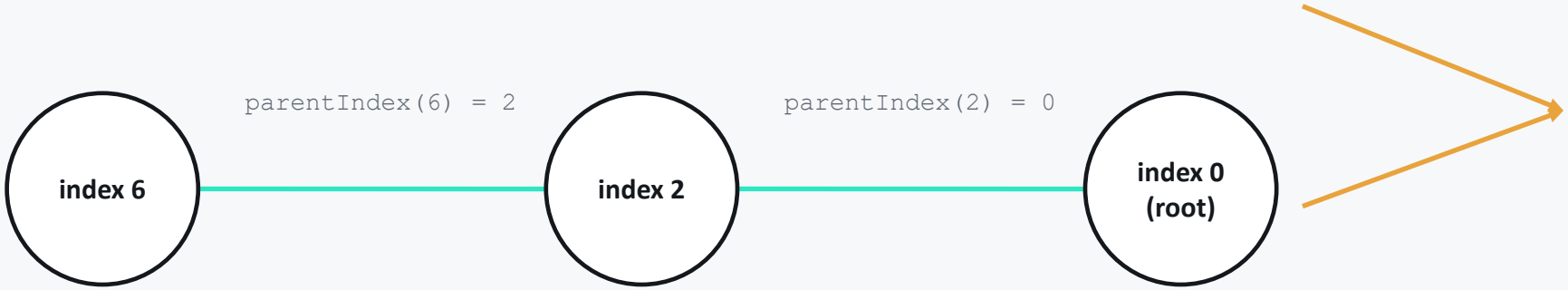
parent

```
(i - 1) / 2
```

The first two are always safe — check index < nodes.size() before using the result. The parent formula has a sharp edge (next slide).

i	0	1	2	3	4	5	6
value	1	2	3	4	5	6	7

The Parent-Index Pitfall

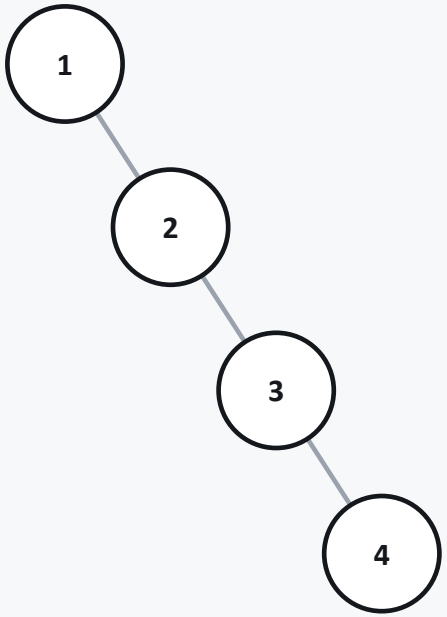


index 0 -> (i-1)/2 (C++ truncates toward zero)

```
int parent = (i - 1) / 2;    // (0-1)/2 = -1/2 truncates to 0, not -1
// Fix: guard with an explicit "is this the root?" check before trusting it.
```

i	0	1	2 / 3
(i−1)/2	0 ← wrong!	0	0 / 1

Space Efficiency: Skewed Trees Waste Exponentially



Skewed tree, height 3: 4 real nodes

height	real nodes	array slots	wasted
2	3	7	4
4	5	31	26
6	7	127	120
10	11	2,047	2,036

$2^{h+1} - 1$ slots reserved for only h+1 real values — the array representation only stays efficient for *complete* trees, which is exactly why heaps (Lecture 23) are the structure built to exploit it.

Key Takeaways

- The Binary Tree ADT's core operation is insertion at the next available position — level-order (breadth-first) insertion.
- Level-order insertion needs a queue, not a stack: it must fully process one level before moving to the next.
- In the array representation, level-order insertion is just push_back — storage order already IS level order.
- 2i+1, 2i+2, and (i-1)/2 connect a node to its children and parent in O(1) — but (i-1)/2 silently returns 0 at the root.
- Array representation's compactness bets on the tree staying complete — a skewed tree pays with array space that grows exponentially in height.