Heap Data Structure

A heap is a specialized tree-based data structure that satisfies the heap property. Heaps are commonly implemented as binary trees, but the tree structure is often implicit in an array representation.

Types of Heaps

- **Max Heap**: In a max heap, for any given node `i`, the value of `i` is greater than or equal to the values of its children.
- **Min Heap**: In a min heap, for any given node `i`, the value of `i` is less than or equal to the values of its children.

Operations

Operation	Description	Time Complexity (Worst)	Time Complexity (Average)	Time Complexity (Best)
Insertion	Add a new element to the heap.	O(log n)	O(log n)	O(1)
Deletion	Remove the root element from the heap.	O(log n)	O(log n)	O(1)
Peek	Return the root element of the heap.	O(1)	O(1)	O(1)
Build Heap	Build a heap from an array of elements.	O(n)	O(n)	O(n)
Heapify	Ensure the heap property is maintained after insertion or deletion.	O(log n)	O(log n)	O(1)
Search	Search for an element in the heap.	O(n)	O(n)	O(1)
Remove	Remove an arbitrary element from the heap.	O(log n)	O(log n)	O(1)

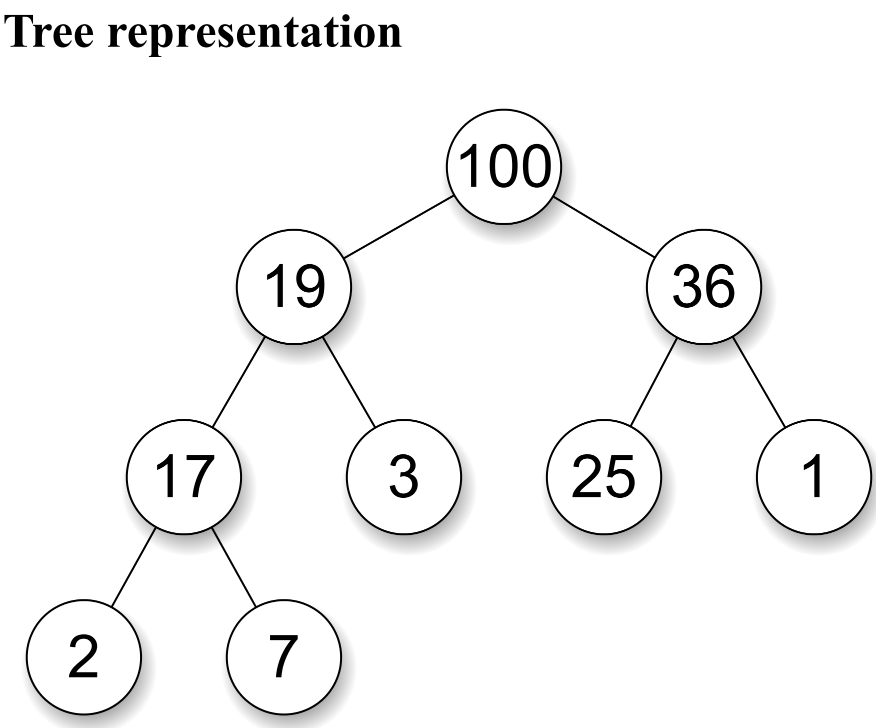
Applications

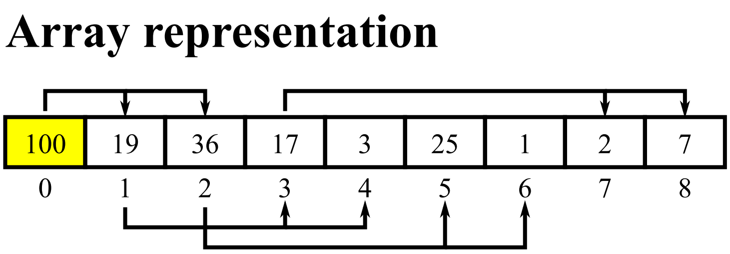
- **Priority Queues**: Heaps are commonly used to implement priority queues where elements with higher priority are served before elements with lower priority.
- **Heap Sort Algorithm**: Heapsort is an efficient sorting algorithm that uses a heap data structure.
- **Graph Algorithms**: Heaps are used in algorithms like Dijkstra's shortest path and Prim's minimum spanning tree.
- **Memory Allocation**: Memory allocation algorithms like malloc in C can use heaps to manage memory.

Implementation

Heaps can be implemented using arrays, where the parent of node `i` is at index `(i-1)/2` and its children are at indexes `2*i+1` and `2*i+2`.

Visualization





In a binary heap represented as an array, the internal nodes and leaves can be identified by their indexes:

1. Internal Nodes: Internal nodes are the nodes that have at least one child. In a binary heap, the internal nodes are all the nodes from index `0` up to `(n/2) - 1`, where `n` is the total number of elements in the heap.
2. Leaves: Leaves are the nodes that have no children. In a binary heap, the leaves are all the nodes from index `(n/2)` to `(n-1)`.

```
   For above example
n=9
n/2=4.5 = 4
therefore internalNodes indexes = [0 to 3]
therefore LeafNodes indexes     = [4 to 8]
```

Example Code

```
#include <iostream>
#include <vector>
using namespace std;


void max_Heapify(vector<int>& vec , int i ,int size){
    //todo-had to add size as a parameter to implement heapsort

    int leftChild  = 2*i+1;
    int rightChild = 2*i+2;
    int largest = i;
    if(leftChild < size && vec[i] < vec[leftChild]){
        largest = leftChild;
    }
    if(rightChild < size && vec[largest] < vec[rightChild]){
        largest = rightChild;
    }
    swap(vec[largest],vec[i]);
    if(largest != i){
        max_Heapify(vec, largest, size);
    }
}

void buildHeap(vector<int>& vec){
    int lastInternalNode = (vec.size()/2)-1;
    for(int i=lastInternalNode;i>=0;i--){
        max_Heapify(vec,i,vec.size());
    }
}



void remove(vector<int>& heap){
//todo-This method removes the root value from the heap and keep the heap
property
    swap(heap[0],heap[heap.size()-1]);
    heap.pop_back();
    max_Heapify(heap,0,heap.size());
}

int peek(vector<int> heap){
//todo-This returns the value of root
    return heap[0];
}

void heapIncreaseKey(vector<int>& heap , int key ,int pos){
//todo-This increase a value of a given index and then maintains heap properties
    if(key<heap[pos])
        cout<<"Key is smaller than current value";
    else{
        heap[pos]=key;
        int parent = (pos-1)/2;
```

```
        while(pos>=1 && heap[pos]> heap[parent] ){
            swap(heap[parent],heap[pos]);
            pos = parent;
            parent = (pos-1)/2;
        }
    }
}

void addElement(vector<int>& vec,int value){
//todo-we use this heapify method when we adding an elemnt to the heap
   vec.push_back(value);
   int index = vec.size()-1;
   int parent = (index-1)/2;
   while(index>=1 && vec[parent]<vec[index]){
       swap(vec[parent],vec[index]);
       index = parent;
       parent = (index-1)/2;
   }
}

void heapSort(vector<int>& vec){
    int size = vec.size();

    // Build max heap
    buildHeap(vec);

    // Extract elements from the heap one by one
    for(int i = size - 1; i > 0; i--){
        swap(vec[0], vec[i]); // Move current root to end
        max_Heapify(vec, 0, i); // Call max heapify on the reduced heap
    }
}




int main() {
    vector<int> heap = {4, 17, 3, 12, 9, 6};

    // Build Heap
    buildHeap(heap); // Time Complexity: O(n), where n is the number of elements
in the heap

    // Output initial heap
    cout << "Initial heap: ";
    for (int i : heap) {
        cout << i << " ";
    }
    cout << endl;

    // Remove root
    remove(heap); // Time Complexity: O(log n), where n is the number of elements
in the heap
    cout << "Heap after removing root: ";
    for (int i : heap) {
        cout << i << " ";
```

```
    }
    cout << endl;

    // Peek at root
    cout << "Peek: " << peek(heap) << endl; // Time Complexity: O(1)

    // Increase key at index 1
    heapIncreaseKey(heap, 22, 1); // Time Complexity: O(log n), where n is the
number of elements in the heap
    cout << "Heap after increasing key at index 1 to 22: ";
    for (int i : heap) {
        cout << i << " ";
    }
    cout << endl;

    // Add element 30
    addElement(heap, 30); // Time Complexity: O(log n), where n is the number of
elements in the heap
    cout << "Heap after adding element 30: ";
    for (int i : heap) {
        cout << i << " ";
    }
    cout << endl;

    // Sort the heap
    heapSort(heap); // Time Complexity: O(n log n), where n is the number of
elements in the heap
    cout << "Sorted heap: ";
    for (int i : heap) {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

Questions

Which of the following is the recurrence relation for Heapify Operation?

a. $T(n) \leq T(2n/3) + \Theta(n)$

b. $T(n) \leq T(n/2) + \Theta(1)$

c. $T(n) \leq T(2n/3) + \Theta(1)$ ✓

d. $T(n) \leq T(3n/2) + \Theta(1)$

The correct answer is c. $T(n) \leq T(2n/3) + \Theta(1)$.

This recurrence relation describes the time complexity of the heapify operation in a binary heap. The heapify operation ensures that a binary tree maintains the heap property (i.e., each parent node is greater than or equal to its child nodes in a max-heap or less than or equal to its child nodes in a min-heap). The process involves potentially moving a node down the tree to restore the heap property, and in the worst case, this movement involves traversing down the longest path, which is approximately $2n/3$ nodes in a complete binary tree. The additional $\Theta(1)$ term represents the constant time spent on comparison and potential swapping of nodes at each level.