A Study In Min Heap

What is a Min Heap ?

Min heap is data structure that satisfies two properties :

Shape property

It states that min heap is a complete binary tree, which is a binary tree that is filled at all levels, except perhaps the last level, which is filled from left to right.

We can infer a couple of things from the above statement. Firstly, the leaf nodes of the tree will be in last level or the level above it. Secondly, in all levels except perhaps the last level, every parent node should exactly have two children. And finally, no right sibilng can exist without its left sibling.

Heap property

It states that the value of the parent node is always less than or equal to that of its children.

It means that the minimum value in the heap shall occur at the root node, and the maximum value in the heap shall occur at any of the leaf nodes. Do note that heap property doesn't mention any relationship between the left and right sibling of a node. The left sibling may have a value lower than the right sibling, or it may be the other way around. As long as every parent has a value lower than its siblings, heap property is satisifed.

If any of the above two property is violated, its not a min heap.

Various Heap Operations

Various operations like insertion and deletion of a node can be done efficiently using a min heap structure. But before we get into that, lets first start with how to create a min heap structure. Given below is the C code for the same.

```
#define LCHILD(x) 2 * x + 1
#define RCHILD(x) 2 * x + 2
#define PARENT(x) x / 2

typedef struct node {
    int data ;
```

```
} node ;

typedef struct minHeap {
    int size ;
    node *elem ;
} minHeap ;
```

In the above code, we are creating two structures, one to represent a node and other to represent the entire min heap. The node structure has only a single member to hold the data. The minHeap structure has two members - one to hold the total size of the min heap at any time, and the other is a pointer to the heap.

Though the heap can be implemented using various child and parent pointers, its easier to use a dynamic array. Since in C array indexing begins at index 0, given a node at index i, its left child shall be at (2 * i + 1) and right child shall be at (2 * i + 2). Also, given a node at index i, its parent node shall be at index (i / 2).

Since a min heap can be growing dynamically, we shall be using a dynamic array which shall grow and shrink as required.

We also need an initializor function that shall initialize the data members of the min heap with default values.

```
minHeap initMinHeap(int size) {
    minHeap hp ;
    hp.size = 0 ;
    return hp ;
}
```

We are creating a variale of type minHeap and setting its size variable to 0, and returning the minHeap variable back. So in the main function, you can have something like,

```
minHeap hp = initMinHeap() ;
```

Insertion

At best case, it takes only O(1) time to insert a node into min heap. But at worst case, the new node may be lower than all nodes in the heap, that it needs to be compared with log(n) elements at most to be put into the root node position. So insertion operation is normally considered as an O(logn) operation. Given below is the C code for the same.

```
void insertNode(minHeap *hp, int data) {
    // allocating space
    if(hp->size) {
        hp->elem = realloc(hp->elem, (size + 1) * sizeof(node)) ;
    } else {
        hp->elem = malloc(sizeof(node)) ;
    }
```

```
    // initializing the node with value
    node nd ;
    nd.data = data ;

    // Positioning the node at the right position in the min heap
    int i = (hp->size)++ ;
    while(i && nd.data < hp->elem[PARENT(i)].data) {
        hp->elem[i] = hp->elem[PARENT(i)] ;
        i = PARENT(i) ;
    }
    hp->elem[i] = nd ;
}
```

In first part of insertNode() function, we allocate space for one node. On its first iteration, size variable is zero, and hence call to malloc() function is performed. Henceforth, all calls shall go to realloc() function which shall increase the memory to allocate one more node in the min heap.

In second part of insertNode() function, we simply create a node variable and initialize it with data that we need to store in the node.

In last part of insertNode() function, we shall find the correct position for the new node in min heap structure. We consider the new memory space created. We shall compare value of last node with the new node value. If its more, we shall move it to the new memory location. We shall now check its parent node, and compare it with new node. If its more, we shall move it to old position of last node. It will continue until we find a node which has value lower than the new node, and place the new node as a child node of that node.

Deletion

Deletion operation is quite different from insertion operation. In each deletion operation, we shall delete the minimum element from min heap, ie, we shall always delete the root node in each deletion operation, and place the last node in root node position. Since we are placing a leaf node in root node, its guaranteed that heap property shall be violated. We shall then call a special function called heapify() function recursively to make sure that heap property is satisfied.

```
void swap(node *n1, node *n2) {
    node temp = *n1 ;
    *n1 = *n2 ;
    *n2 = temp ;
}
```

```
void heapify(minHeap *hp, int i) {
    int smallest = (LCHILD(i) < hp->size && hp->elem[LCHILD(i)].data < hp->elem[i].data) ? LCH
    if(RCHILD(i) < hp->size && hp->elem[RCHILD(i)].data < hp->elem[largest].data) {
        smallest = RCHILD(i) ;
    }
    if(smallest != i) {
        swap(&(hp->elem[i]), &(hp->elem[smallest])) ;
        heapify(hp, smallest) ;
```

```
    }
}
```

```
void deleteNode(minHeap *hp) {
    if(hp->size) {
        printf("Deleting node %d\n\n", hp->elem[0].data) ;
        hp->elem[0] = hp->elem[--(hp->size)] ;
        hp->elem = realloc(hp->elem, hp->size * sizeof(node)) ;
        heapify(hp, 0) ;
    } else {
        printf("\nMin Heap is empty!\n") ;
        free(hp->elem) ;
    }
}
```

In deleteNode() function, last node is placed at root node position, heap size decremented by 1, and memory of min heap reduced by one node. Heapify() function shall then be called.

In heapify() function, given a node at index i, we shall compare all the three nodes (parent, left and right child), and find the smallest node among the three. If its not the parent node, then heap property is violated. Swap parent node with smallest node, and call heapify() function until heap property is satisfied.

BuildMinHeap

Given an array of n numbers, we can call the insertNode() function n times to create the min heap. Since each call to insertNode() can take upto O(logn) time, we need a maximum of O(nlogn) time to create the entire min heap. But instead of following such an approach, we can follow a relatively better buildMinHeap() function that requires only O(n) time.

```
void buildMinHeap(minHeap *hp, int *arr, int size) {
    int i ;

    // Insertion into the heap without violating the shape property
    for(i = 0; i < size; i++) {
        if(hp->size) {
            hp->elem = realloc(hp->elem, (hp->size + 1) * sizeof(node)) ;
        } else {
            hp->elem = malloc(sizeof(node)) ;
        }
        node nd ;
        nd.data = arr[i] ;
        hp->elem[(hp->size)++] = nd ;
    }

    // Making sure that heap property is also satisfied
    for(i = (hp->size - 1) / 2; i >= 0; i--) {
        heapify(hp, i) ;
    }
```

```
}
```

In first step, we insert all elements from the array into the min heap without bothering about heap property. Since the heap is built using array structure, shape property is never violated.

In last step, we shall determine the last parent in the heap and call heapify() function on that parent node, until heap property is satisfied for that node. We shall then work back by calling heapify() function on each of those parent nodes, until we reach root node, by which time, heap property shall be satisfied.

Traversal

There are mainly two types of traversals possible on a min heap - depth first traversal and breadth first (level order) traversal.

In depth first traversal, we visit the nodes depth-wise, meaning that we go deeper and deeper into childrens of a left sibling before covering the right sibling. There are mainly three depth first traversals possible - inorder, preorder and postorder traversal.

Inorder Traversal

```
void inorderTraversal(minHeap *hp, int i) {
    if(LCHILD(i) < hp->size) {
        inorderTraversal(hp, LCHILD(i)) ;
    }
    printf("%d ", hp->elem[i].data) ;
    if(RCHILD(i) < hp->size) {
        inorderTraversal(hp, RCHILD(i)) ;
    }
}
```

In the above function, we check whether left child of a node at index i exists in the heap. If yes, we call the inorderTraveral() function on the left child. We now display value of the node at index i. Then we check whether right child of a node at index i exists in the heap. If yes, we call inorderTraversal() on the right child. This shall continue until we traverse the whole min heap.

Preorder Traversal

```
void preorderTraversal(minHeap *hp, int i) {
    if(LCHILD(i) < hp->size) {
        preorderTraversal(hp, LCHILD(i)) ;
    }
    if(RCHILD(i) < hp->size) {
        preorderTraversal(hp, RCHILD(i)) ;
    }
    printf("%d ", hp->elem[i].data) ;
}
```

In the above function, we check whether left child of the node at index i exists. If yes, we call preorderTraversal() on the left

child. We then check whether right of the node exists. If yes, we call preorderTraversal() on the right child. we then display the value of the node at index i. We can see that code for preorderTraversal() and inorderTraversal() are quite similar and shall be similar to postorderTraversal() function. The only difference is the order in which the recursive functions are called.

Postorder Traversal

```
void postorderTraversal(minHeap *hp, int i) {
    printf("%d ", hp->elem[i].data) ;
    if(LCHILD(i) < hp->size) {
        postOrderTraversal(hp, LCHILD(i)) ;
    }
    if(RCHILD(i) < hp->size) {
        postorderTraversal(hp, RCHILD(i)) ;
    }
}
```

In postOrderTraversal() function, we display the value of the node at index 1, and recursively call the function, first on the left child and then on the right child, until we traverse the whole min heap.

Levelorder Traversal

```
void levelOrderTraversal(minHeap *hp) {
    int i ;
    for(i = 0; i < hp->size; i++) {
        printf("%d ", hp->elem[i].data) ;
    }
}
```

Implementing level order traversal is quite simple, as we are using the array implementation. All we need to do is to traverse through the whole array from index 0 to (hp->size - 1) to get the level order traversal.

Applications of Min Heap

As you are already aware by now, when we delete an element from the min heap, we always get the minimum valued node from the min heap, which means that we can access the minimum valued node in O(1) time. So if you need a quick access to the smallest value element, you can go for min heap implementation.

Min heaps can be used to implement priority queues. In a priority queue, rather than using the value of a node, we are using priority of the node to position it in the min heap. Priority queues are used heavily in job schedulers.