DSA Project
Team - 45

BITS Pilani
Pilani Campus

Project Details

Team Members

Priyank Shethia	2021A7PS0657P
Hardik Gupta	2021A7PS2421P
Trayambak Shrivastava	2021A7PS1629P
Aditya Khandelwal	2021A7PS2422P
Aryan Bansal	2021A7PS2776P

Project No. 5

Mound data structure and the implementation of its insertion and extraction functions.

Research Paper Brief

Mounds: Array-Based Concurrent Priority Queues

by Yujie Liu and Michael Spear, *Department of Computer Science and Engineering Lehigh University*

Abstract:

This paper introduces a concurrent data structure called the mound. The mound is a rooted tree of sorted lists that relies on randomization for balance. It supports O(log(log(N))) insert and O(log(N)) extractMin operations, making it suitable for use as a priority queue. We present two mound algorithms: the first achieves lock freedom via the use of a pure-software double-compare- and-swap (DCAS), and the second uses fine grained locks. Mounds perform well in practice, and support novel operations that we expect to be useful in parallel applications, such as extractMany and probabilistic extractMin.

Objectives

1. Implementing the data structure **mound** as described in Section 2 of the assigned research paper.
2. Implementation of insertion function using binary search method.
3. Time complexity of insertion function.
4. Implementation of an extraction method namely ‘extractMin’ using helper function ‘moundify’.
5. Time complexity of ‘extractMin’.

Mound Data Structure

1. Mound is a tree-based data structure of sorted lists which means each node of the tree contains a sorted list. It's operations and implementation is similar to those of a heap implemented simple priority queue.

2. *Mound Invariant -* The value of a node in a mound is the first value stored in the list originating from that node of the tree if the list is non empty and T(explained further) if empty. The mound invariant property states that in a mound data structure, each parent node's value is less than or equal to its child node's value.

3. Each node in the tree is called "MNode" containing a pointer to a list, a "dirty" flag indicating if the mound property is violated, and a counter, which is not used in locking mounds. The "LNode" represents a node in the list and holds a value and a reference to the next node. The mound is initialized with empty lists and therefore maintains the mound property for parent-child pairs. The operationsimplemented include inserting a value and extracting the minimum value in the data structure.

Mound Data Structure

Listing 1 Simple data types and methods used by a mound of elements of type "T"

```
type LNode
   T          value    ▷ value stored in this list node
   LNode*     next     ▷ next element in list

type MNode
   LNode*     list     ▷ sorted list of values stored at this node
   boolean    dirty    ▷ true if mound property does not hold
   integer    c        ▷ counter – incremented on every update

global variables
   tree_{i∈[1,N]} ← ⟨nil, false, 0⟩   : MNode     ▷ array of mound nodes
   depth ← 1                          : integer   ▷ depth of the mound tree
```

NOTE :

1. The member of the MNode data structure, *c (counter)* has not been used in our implementation as it is not used when multiple threads do not operate simultaneously.

2. Generic data types have not been implemented i.e. void pointers have not been used. All values are of the type integer in our implementation.

MNode: Represents nodes in the mound, containing a sorted linked-list

LNode: Represents nodes containing the elements in the sorted linked lists

The implementation has the ability to store arbitrary, non-unique, totally-ordered values of type T, and ⊤ is the maximum value of that type possible. For simplicity and ease of implementation, we have considered “int” as the type of elements in the mound and the value of ⊤ to be equal to the maximum value of “int” ie. 2147483647 (2^31-1). We reserve ⊤ as the return value of an extractMin on an empty mound, to prevent the operation from blocking.

By default a mound is initialized by setting every element in the tree to <nil, false, 0>. This also means that whenever a new node is created it has a value set of <nil, false, 0> by default. To include the same in our implementation we have represented this data set through an LNode which contains the value ⊤. This node is used to represent the end of the linked list, and also acts as a way to view when the linked list is empty.

Code Snippets for Mound Data Structure And Relevant Create Functions

```
typedef struct LNode *LNODE;
typedef struct MNode MNode;

int HEIGHT = 0;

struct LNode
{
    int value;
    LNODE next;
};

struct MNode
{
    LNODE list;
    bool dirty;
    int count;
};
```

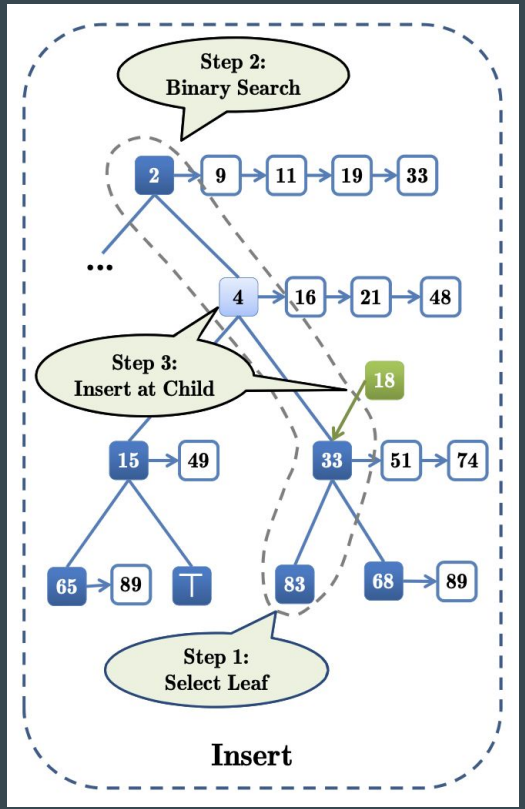
```
MNode *tree;

LNODE createLNode(int value)
{
    LNODE node = (LNODE)calloc(1, sizeof(struct LNode));
    node->value = value;
    node->next = NULL;
    return node;
}

void createTree()
{
    tree = (MNode *)calloc(pow(2, HEIGHT + 1) - 1, sizeof(MNode));
    for (int i = 0; i < pow(2, HEIGHT + 1) - 1; i++)
    {
        (tree + i)->list = createLNode(T);
        (tree + i)->dirty = false;
        (tree + i)->count = 0;
    }
}
```

Insertion

The insertion operation in a mound data structure involves adding a new value while considering a threshold parameter. It selects a random leaf node and compares the value to be inserted with the leaf's value. If the new value is smaller or equal, the insertion can occur at the branch containing that leaf node. For the same, it checks for a suitable ancestor node using a binary search on that particular branch of the tree. The search continues until it finds the ancestor node with the smallest value in that branch which is greater than or equal to the new value. The new value is then inserted at that ancestor node, preserving the mound property.

If the leaf's value is smaller, the algorithm randomly checks another leaf for insertion if the threshold value has not been exceeded. If the algorithm fails to find a suitable leaf after a certain number of attempts, the mound is expanded by one level to accommodate the new value.

The above mentioned number of attempts is what is referred to as the 'Threshold Parameter' in the research paper. In case of failure to insert within the threshold number of attempts (taken to be 8 in our implementation), the array (tree) is resized to introduce another level of the tree and the node is inserted at this level. All nodes in the new level are initialized to empty nodes, in order to maintain the completeness of the tree.



Insertion Code Snippet and Explanation

```
bool insertLNode(int value)
{
    LNODE node = createLNode(value);
    int count = 0;
    int leaf_index = -1;
    for (int i = 1; i <= THRESHOLD; i++)
    {
        int random = rand();
        leaf_index = (int)pow(2, HEIGHT) + random % (int)pow(2, HEIGHT) - 1;
        if (insertion(leaf_index, node) == true)
        {
            COUNT++;
            return true;
        }
    }
    HEIGHT++;
    resizeTree();
    insertAtHead((tree + 2 * (leaf_index + 1) - 1), node);
    COUNT++;
    return true;
}
```

The *insertLNode* function uses two variables

1. *leaf_index* - Stores a randomly generated index corresponding to one of the leaves (MNodes at level = HEIGHT)
2. *count* - Tracks the number of iterations of insertion attempts, so as to compare with the threshold.

At each iteration the possibilities are:

1. **CASE 1** - A successful insertion while *count < threshold* .
2. **CASE 2** - *count >= threshold* In this case, the tree(array) is resized and a new level is introduced (via realloc() function). In this case insertion has to succeed as the new leaves created are initialised to val(⊤).

Insertion Code Snippet and Explanation

```
bool insertion(int leaf_index, LNODE node)
{
    if(node->value>(tree+leaf_index)->list->value)
    return false;
    int insertion_index = binary_search(node->value, leaf_index);

    insertAtHead(tree + insertion_index, node);
    return true;
}
```

The insertion function first checks if the value of the new element is smaller than or equal to the leaf MNode's value. If it is then it attempts to insert an LNode via *binary_search* in the branch which contains the leaf MNode else returns false.

Insertion Code Snippet and Explanation

resizeTree:

The function resizes the mound to include one more level thereby creating 2^height more MNodes. All these new MNodes are initialised to ⊤ i.e. all those point to a linked list which contains only the element ⊤

insertAtHead:

this function inserts an LNode at the beginning of the list originating at the given MNode

```
void resizeTree()
{
    tree = (MNode *)realloc(tree, (pow(2, HEIGHT + 1) - 1) * sizeof(MNode));
    for (int i = pow(2, HEIGHT) - 1; i < pow(2, HEIGHT + 1) - 1; i++)
    {
        (tree + i)->list = createLNode(T);
        (tree + i)->dirty = false;
        (tree + i)->count = 0;
    }
}

void insertAtHead(MNode *head, LNODE node)
{
    node->next = head->list;
    head->list = node;
}
```

Auxiliary Functions Used to Execute InsertLNode

Binary Search(Code Snippet)

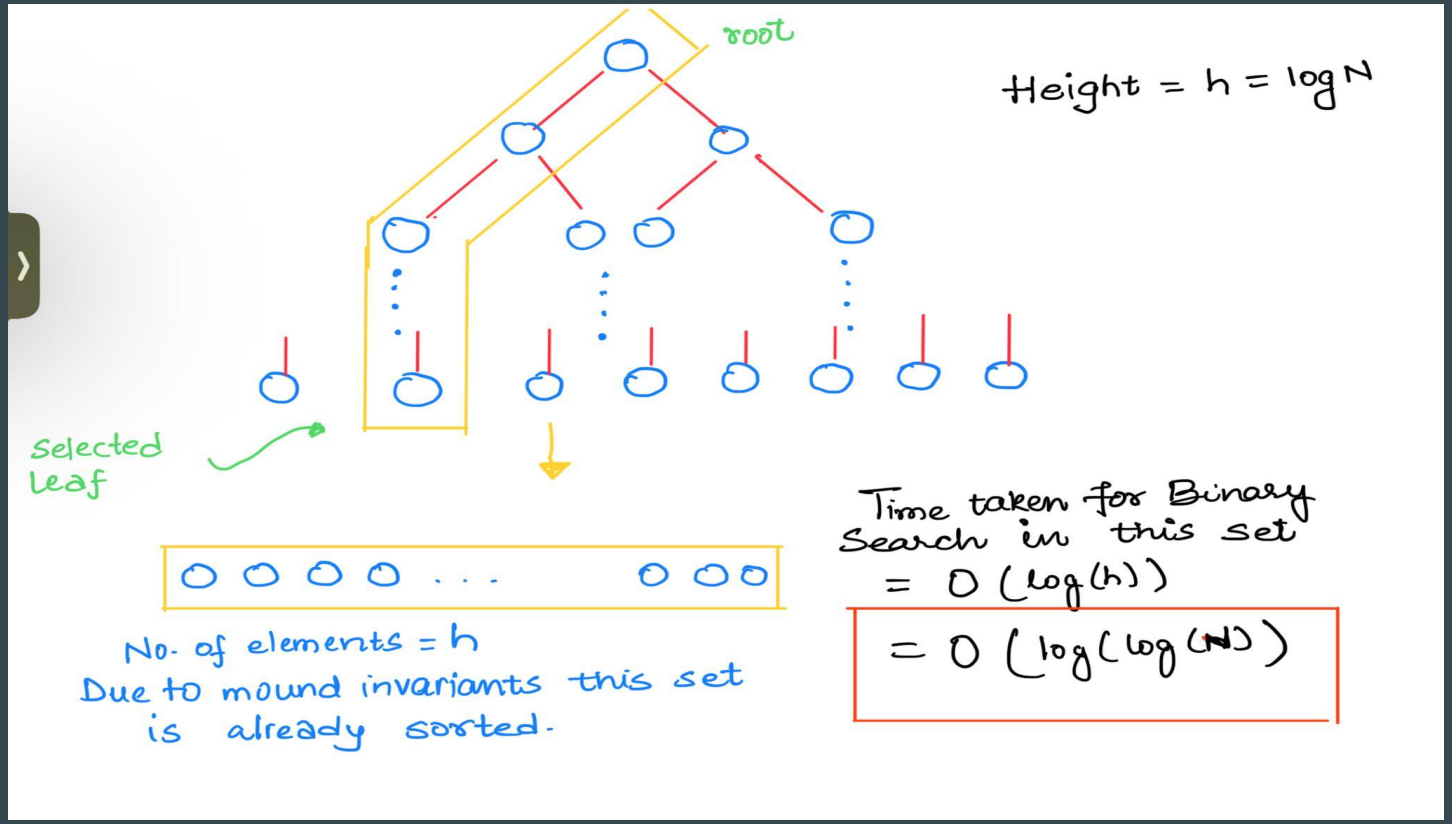
binary_search function: This function applies binary search on the levels of the mound based upon the values of MNodes in those levels within the path from the root to the leaf - MNode. The binary search operates on the values where , initially , the high corresponds to the leaf node, and the low corresponds to the root. The values of high and low variables change after each comparison. It traverses through the path to find the level at which the node at which the insertion should occur, and then utilises the get_ancestor function to find the node which lies on the path at that level.

get_ancestor function: It takes in a parameter 'generation' which tells the code the level at which the node needs to be found and returns the index of the the ancestor MNode at that level in the branch. The value of ancestor increases as we move down the mound.

```
int get_ancestor(int index, int generation)
{
    return index / pow(2, HEIGHT - generation);
}
```

```
int binary_search(int key, int leaf_index)
{
    int ans = -1;
    int low = 0;
    int high = HEIGHT;
    while (low <= high)
    {
        int mid = low + (high - low + 1) / 2;
        int mid1 = get_ancestor(leaf_index + 1, mid) - 1;
        int midVal = (tree + mid1)->list->value;

        if (midVal > key)
        {
            ans = mid1;
            high = mid - 1;
        }
        else if (midVal < key)
        {
            low = mid + 1;
        }
        else if (midVal == key)
        {
            ans = mid1;
            break;
        }
    }
    return ans;
}
```

Insertion Time Complexity(Binary Search)



Overall Insertion Time Complexity

T.C.(InsertLNode) = K x T.C.(Insertion) + C : *where* K = *counter(max.=threshold); denoting the number of iterations*

T.C.(Insertion) = T.C.(Binary_Search) + C : Since all other code in *Insertion* function takes constant time only

T.C.(Binary_Search) = O(log(log(N)) : Derived in previous slide

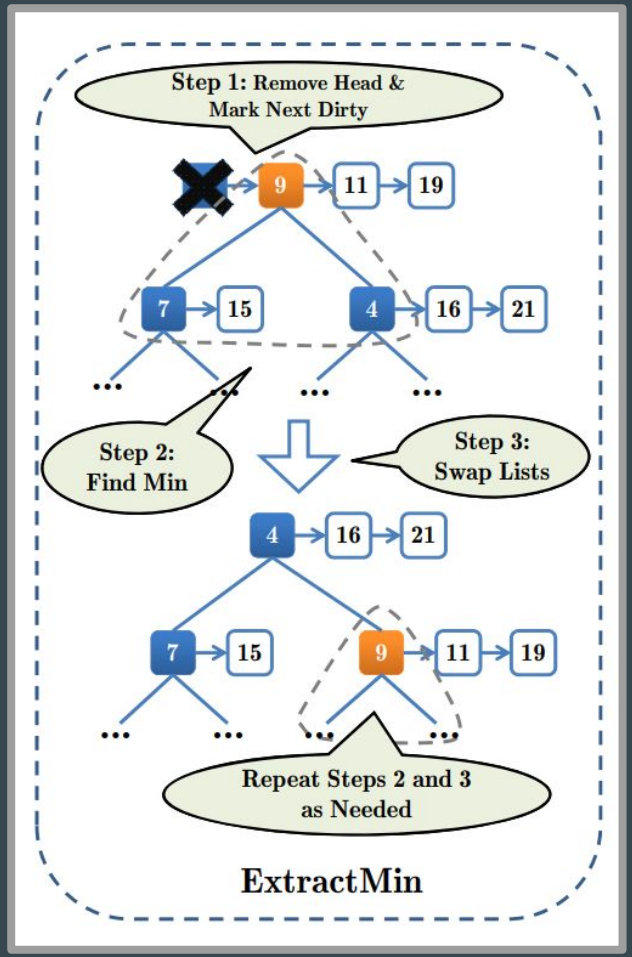
Using the above equations we get:

T.C.(InsertLNode) = O(log(log(N))

ExtractMin

The ExtractMin operation in a mound data structure is used to retrieve and remove the minimum value from the structure. It starts by extracting the current minimum value, typically located at the root node. The root's value is then updated to the next value in its list, or set to a special value (e.g., $\top$) if the list becomes empty.

After extracting the minimum, the mound invariant between the root and its children may be violated, hence “Dirty” is set to be true on the root node. To restore the mound property, a helper function called moundify is used. Moundify examines the triangle formed by the dirty node (the root) and its two children. If either child is dirty, moundify is recursively called on that child first which can happen during parallel access. Once neither child is dirty, moundify compares the values of the dirty node(the root) and its children to determine which is the smallest. (Moundify is explained further)



ExtractMin Code Snippets and Explanation

The invariant of the mound data structure clearly suggests that value stored in the root will be the lowest and hence it will be the extracted value.

Since there is a possibility of mound property having been violated so dirty is set true.

This memory which is occupied by the node containing the value is freed.

After this the moundify function is called at the root MNode to restore the invariant property.

```
int extractMin()
{
    int min = tree->list->value;
    LNODE temp = tree->list;
    tree->list = tree->list->next;
    free(temp);
    tree->dirty = true;
    moundify(0);
    return min;
}
```

Moundify Code Snippet and Explanation

We have created moundify to be a recursively called function with the base case as the node being the leaf node. If it is, then by definition it must hold mound invariant hence we set its dirty to false.

The second and third if conditions are designed specifically for the purpose of dealing with parallel processing systems, where there is a possibility that other threads might hamper the dirty property of some MNodes. These conditions simply call the moundify functions for such MNodes.

```
void moundify(int index)
{
    if (index >= pow(2, HEIGHT) - 1)
    {
        (tree + index)->dirty = false;
        return;
    }
    int parent_val = (tree + index)->list->value;
    int left_child = (index + 1) * 2 - 1;
    int right_child = left_child + 1;
    int left_child_val = (tree + left_child)->list->value;
    int right_child_val = (tree + right_child)->list->value;

    if ((tree + left_child)->dirty)
    {
        moundify(left_child);
    }

    if ((tree + right_child)->dirty)
    {
        moundify(right_child);
    }
```

Moundify Code Snippet and Explanation

After that we select the triangle arising out of the node(*index)* and its children and recursively swaps the node with the smaller of its two children till the node reaches the point where it is smaller than both its children.

If the node(index) happens to be the smallest value node already then we simply make the value of dirty field to be false.

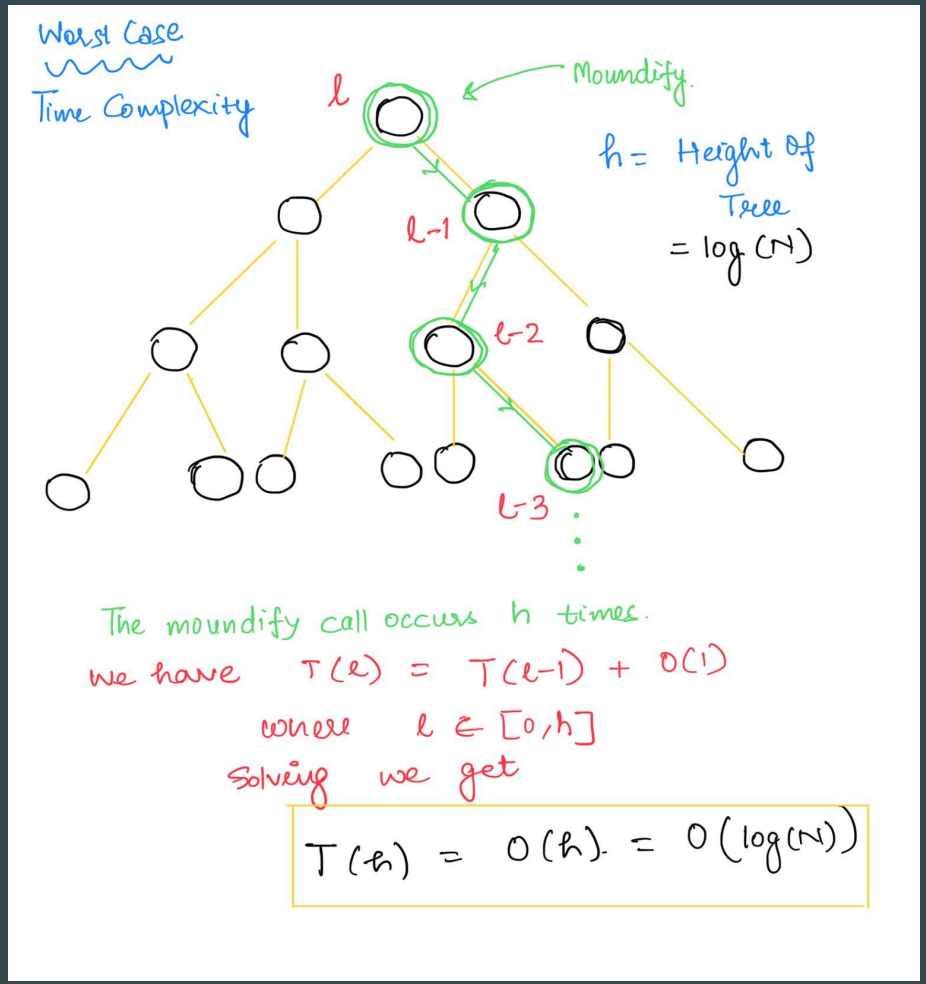
```
    bool isLeftMin = left_child_val <= right_child_val;
    if (isLeftMin && left_child_val < parent_val)
    {
        MNode temp = tree[index];
        tree[index] = tree[left_child];
        tree[left_child] = temp;
        moundify(left_child);
    }
    else if (!isLeftMin && right_child_val < parent_val)
    {
        MNode temp = tree[index];
        tree[index] = tree[right_child];
        tree[right_child] = temp;
        moundify(right_child);
    }
    else
    (tree + index)->dirty = false;
}
```

ExtractMin. Time Complexity

ExtractMin. T.C.= Moundify T.C.

Moundify T.C. (Worst Case shown in adjacent diagram)

Hence, T.C.= O(log(n)).



THANK YOU