Array and Simple Queries

Jake Fallin, Dylan DiGeronimo, Kexian Wu

I pledge my honor that I have abided by the Stevens Honor System.

Problem Statement

- [The problem]
- We take in
  - N (number of elements in array)
  - M (number of queries)
  - A (array of integers)
- 2 Types of queries
  - 1: (1 i j)
    - Removes from i to j and adds to the front of the array
  - 2: (2 i j)
    - Removes from i to j and adds to the back of the array
- After execution print Abs(A[1] - A[N]) and the resulting array

Sample Input

```
8 4
1 2 3 4 5 6 7 8
1 2 4
2 3 5
1 4 7
2 1 4
```

Sample Output

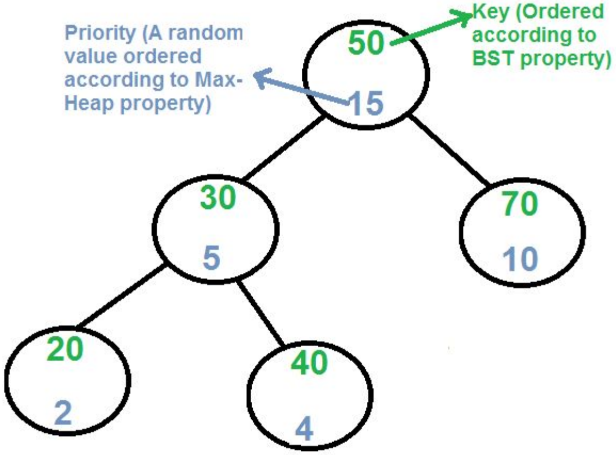
```
1
2 3 6 5 7 8 4 1
```

Our First Approach

- Attempted a simple solution
- Loop
  - Read query type
  - Move elements around using a temp array
  - Edit the main array as needed
  - Repeat on next query
- Took too long
  - Inefficiency = **BAD**
  - Needs to work on very large arrays
- We needed something more efficient
  - Decided to use a treap

What's a Treap?

- Variation on a balanced binary search tree
  - Uses randomization and heap priority to maintain balance
  - Each node has a value and a priority
- Would allow us to shift data more efficiently than manipulating an array
  - Search, insert, and delete are O(log n)



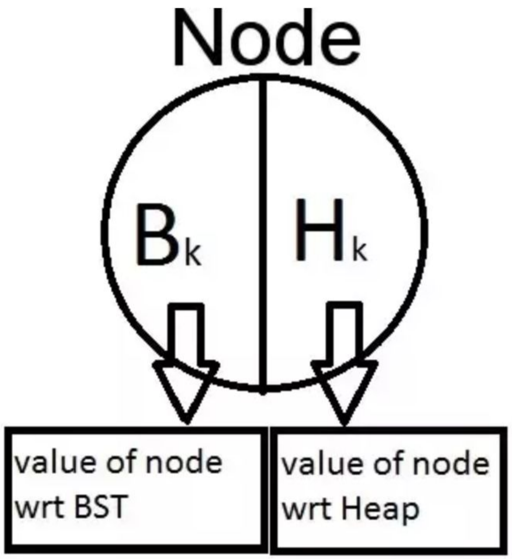
Algorithm Summary

1. Read in user input
2. Create the initial treap
3. Read in query and extract subtree from i to j
    a. If type 1, merge subtree at “front” of treap
    b. If type 2, merge subtree at “back” of treap
4. Repeat for all queries
5. Store final tree in array using inorder traversal

Code Overview

Method by Method

The Node




```
// 1<=N<=10^5
const int MAX_N = 100000;

// insert the (key, priority,size)
struct Node
{
  int value;     // data/key field
  int priority; //rand() -- int type
  int size;      // size -- tree size
  Node *left;    // left child
  Node *right;   // right child
} node[MAX_N];

int values[MAX_N];
int inc = 0;
```

Creating the node, as well as initializing the size, storage array, and a counter value

Merge

```
// get the size of a tree
int getSize(Node *n)
{
  if (n == NULL)
  {
    return 0;
  }
  return n->size;
}
```

Helper function to return tree size

```
// join two trees -- with no rotation
// the heap order is to maintain a min heap
// the merge operation merges two given treaps L and R into a single treap T
// only care about immediate parent

// all Nodes in the left subtree are visited before the root is visited
// and all Nodes in the right subtree are visited after the root is visited
Node *merge(Node *n1, Node *n2)
{
  //if null then the tree is the other side
  if (n1 == NULL)
  {
    return n2;
  }
  if (n2 == NULL)
  {
    return n1;
  }
  //if y priority is higher
  if (n1->priority < n2->priority)
  {
    //merge right subtree recursively
    n1->right = merge(n1->right, n2);
    //increase size
    n1->size = getSize(n1->left) + getSize(n1->right) + 1;
    return n1;
  }
  //if x priority is higher
  else
  {
    //merge left subtree recursively
    n2->left = merge(n1, n2->left);
    //increase size
    n2->size = getSize(n2->left) + getSize(n2->right) + 1;
    return n2;
  }
}
```

Function that merges two treaps, preserving min heap order

Extract

```
//Helper function for extracting subtrees that recursively splits
void splitNode(Node *n, Node *&left, Node *&right, int value)
{
  if (!n)
  {
    //set to null and ignore
    left = NULL;
    right = NULL;
  }
  else
  {
    //use left subtree
    int maxSize = getSize(n->left) + 1;
    //if left subtree is greater than value
    if (value < maxSize)
    {
      // if in the bounds split right
      right = n;
      splitNode(n->left, left, n->left, value);
    }
    else
    {
      //if in the bounds split right
      left = n;
      splitNode(n->right, n->right, right, value - maxSize);
    }
    //increase tree size
    n->size = getSize(n->left) + getSize(n->right) + 1;
  }
}
```

Helper that splits trees

```
//in order to get the subtree must be able to spit into subtrees
Node *extract(Node *&n, int from, int to)
{
  Node *left, *right, *middle;
  //split from right
  splitNode(n, middle, right, to);
  //split from left
  splitNode(middle, left, middle, from);
  //merge into Node
  n = merge(left, right);
  return middle;
}
```

Function that extracts a node

Inorder Traversal

```
// Inorder tree traversal
// 1) visit node
// 2) traverse left subtree
// 3) traverse right subtree
// Performs recursive Inorder traversal of a given binary tree.
void Inorder(Node *n)
{
  if (n != NULL)
  {
    Inorder(n->left);
    values[inc] = n->value;
    inc++;
    Inorder(n->right);
  }
}
```

Function that recursively passes over the tree, storing the values in the array we initialized earlier, using the counter variable we created to track the index

Main Pt. 1: Input & Initialization

```
int main()
{
  // Dr. B's io speed trick
  ios::sync_with_stdio(false);
  cin.tie(NULL);
```

Speeds up input time

```
/*
 * Take in user input
 * Line 1: Size n of array, number m of queries
 * Line 2: int array A[]
 * Remaining lines: Queries
 * Query format: 1 i j (remove from i to j and move to front)
 * or 2 i j (remove from i to j and move to back)
 */
int n, m;
cin >> n;
cin >> m;
Node *tree = NULL;
```

Get input for array size and query numbers and initialize tree

```
for (int i = 0; i < n; i++)
{
  // initialize values in each Node

  cin >> node[i].value;
  node[i].priority = rand();
  node[i].size = 1;
  // points to the root of the tree
  tree = merge(tree, node + i);
}
```

Reads in the user-inputted array, putting each value in a node, and then filling the tree

Main Pt. 2: Evaluating Queries

```
while (m > 0)
{
  m--;
  int type;
  int i, j;
  cin >> type;
  cin >> i;
  cin >> j;

  Node *subtree = extract(tree, i - 1, j);
  // Modify the given array by removing elements from i to j and adding them to the front
  if (type == 1)
  {
    // points to the root of the tree
    tree = merge(subtree, tree);
  }
  // Modify the given array by removing elements from i to j and adding them to the back
  else if (type == 2)
  {
    // points to the root of the tree
    tree = merge(tree, subtree);
  }
}
```

Using a while loop, evaluates each query and the indices it will act on

Extracts new subtree from index i to index j

If query type 1, attach the subtree to the front

If query type 2, attach the subtree to the end

Main Pt. 3: Storage & Printing

```
  // Store values of tree using in-order traversal
  Inorder(tree);

  //print tree
  cout << abs(values[0] - values[n - 1]) << endl;
  for (int i = 0; i < n; ++i)
  {
    cout << values[i] << " ";
  }
  //newline
  cout << endl;

  return 0;
}
```

Store the final order in the array using an inorder traversal of the tree

Calculate the absolute value and print it

Using a loop, print the array in order, with a space after each item

Woo hoo, it's done!

Test Cases

- Proven to work in extremes
  - Negative integers
  - Extremely large input (99000 elements and 99000 queries!) ⤵
  - Extremely small input (Empty array, 0 queries)
  - Don’t believe us? Watch it happen!

Input (stdin) Download

```
99000 99000
27321 11336 3867 12684 25897 18039
1871 19184 4161 26528 9152 16404 29
```

Expected Output Download

```
9570
26783 23986 18840 9941 4689 28985 2
4421 10000 836 1453 22129 31818 236
```

Compiler Message

```
Success
```

References Used

- https://threads-iiith.quora.com/Treaps-One-Tree-to-Rule-em-all-Part-1
- https://threads-iiith.quora.com/Treaps-One-Tree-to-Rule-em-all-Part-2

Thank You Very Much!