2: Data Structures and Libraries

CPCFI

UNAM's School of Engineering

2021

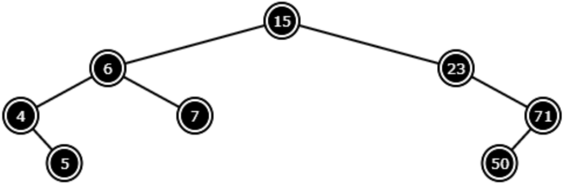
Based on: Halim S., Halim F.*Competitive Programming 3*. Handbook for ACM ICPC and IOI Contestants. 2013

Table of Contents

CPCFI

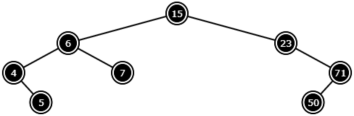
Index

Balanced Binary Search Tree - BST



Balanced Binary Search Tree

- A Binary Search Tree (BST) is a tree where each vertex has at most two children nodes that satisfy the BST Property



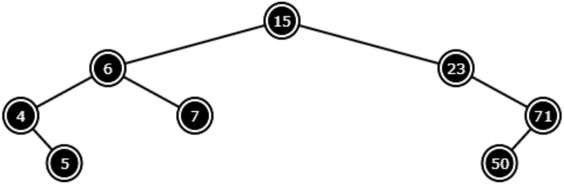
- **BST Property**: Tree with a root at x where each value v:
  - On the left of x holds that $v < x$
  - On the right of x holds that $v \geq x$
- `C++ STL map`: stores *key* $\rightarrow$ *value*
- `C++ STL set`: only stores *key*
- `ch2_05_map_set.cpp`

BST as Table ADT

- A BST is an efficient and dynamic[1] data structure to implement a Table or Map ADT
- Table ADT must support:
  - `search(v)`
  - `insert(v)`
  - `remove(v)`
  - `min()`, `max()`
  - `successor(v)`, `predecessor(v)`
- These operations run in $O(n \log n)$

[1] Data structure that keeps efficiency even in the presence of many update operations

BST - Vertex Attributes



- Each vertex has at least 4 attributes:
  1. `parent`
  2. `left`
  3. `right`
  4. `key`, `value`, `data`

BST - Operations

1. Query operations
   - `Search(v)`
   - `Predecessor(v)`, `Successor(v)`
   - Inorder traversal
2. Update operations
   - `Insert(v)`
   - `Remove(v)`
   - Create BST

CPCFI

BST - Query Operations I

- `search(v)`: $O(h)$ [2]
  - Set the current vertex to the root. Then, check if the current vertex is smaller, equal or larger than `v`, depending on the answer, update the current vertex to the left or right child of the root
  - Similarly, we can find the **minimum** or the **maximum** of a BST by going all the way to the left or right of the BST

[2] h is the height of the tree

BST - Query operations II

- `predecessor(v)`: $O(h)$
  - If `v` has a left subtree, then the predecessor is the maximum element of the left subtree
  - If `v` does not have a left subtree, then we need to iterate over its ancestors (parents) to find the first vertex `w` that is smaller than `v`
  - If `v` is the minimum of the BST, `v` does not have a predecessor
- `successor(v)`: $O(h)$
  - If `v` has a right subtree, then the successor is minimum element of the right subtree
  - If `v` does not have a right subtree, then we need to look for the first vertex `w` that is greater than `v`
  - If `v` is the maximum of the BST, `v` does not have a successor

BST - Query operations III

- Inorder traversal
  - Visit order:
    1. Left subtree
    2. Root
    3. Right subtree
  - Obtains the list of sorted integers inside the BST
  - $O(n)$
- Preorder traversal
  - Visit order:
    1. Root
    2. Left subtree
    3. Right subtree
- Postorder traversal
  - Visit order:
    1. Left subtree
    2. Right subtree
    3. Root

BST - Update operations I

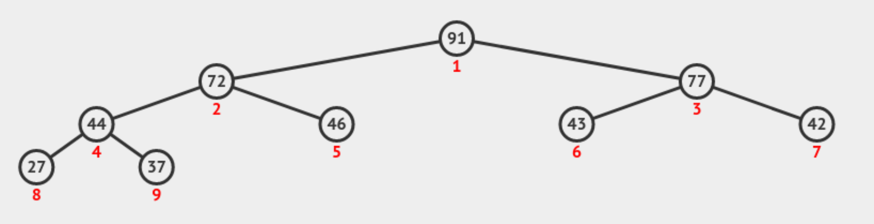
- `insert(v)`: $O(h)$
  - Follow a similar approach to `search(v)` but instead of reporting that the vertex does not exist, create a new vertex in the insertion point

BST - Update operations II

- `remove(v)`: $O(h) +$, three possible cases:
  1. Vertex `v` is a leaf vertex: $O(1)$
     - Simply remove the vertex (parent's left or right child should point to `null`)
  2. Vertex `v` is an internal or root vertex with one child: $O(1)$
     - Connect `v`'s only child with `v`'s parent
  3. Vertex `v` is an internal or root vertex with two children: $O(h)$
     - Replace `v` with its successor
     - Delete its duplicated successor in its right subtree

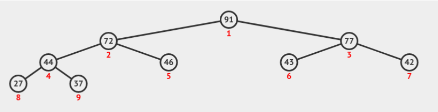
Index

Heap



Heap

- Heap is a **complete** binary tree: every node, except possibly in the last level, must have both left and right children



- **Heap property**: in each subtree rooted at x, items on the left and right subtrees of x are smaller (or equal) than x
  - This property guarantees that the top element of the Heap (or the root) is the maximum element
  - There is no notion of *search* in the Heap
  - Allow for fast deletion of the maximum element and insertion of new items: $O(\log n)$
- The height of a Binary Heap of n elements will have a height no taller than $O(\log n)$

Heap - Priority Queue ADT

- The (Max) Heap is useful for modeling a Priority Queue ADT where the item with the highest priority can be dequeued and a new item v can be enqueued in $O(\log n)$
- C++ STL `priority_queue`
- `ch2_06_priority_queue.cpp`

Heap - 1D representation

- Complete binary search trees (Heaps) can be stored in a compact 1-indexed array of size $n+1$

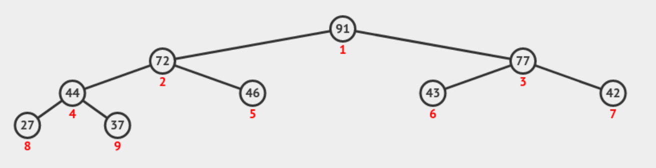


Figure: Heap - Tree Representation

$$A = [N/A, 91, 72, 77, 44, 46, 43, 42, 27, 37]$$

Heap - 1D representation

$$A = [N/A, 91, 72, 77, 44, 46, 43, 42, 27, 37]$$

- Index 0 is ignored
- Operations from index i [bit manipulation]:
  - Parent: $\left\lfloor \frac{i}{2} \right\rfloor$, $i \gg i$
  - Left child: $2i$, $i \ll i$
  - Right child: $2i + 1$, $(i \ll i) + 1$

Heap - Operations

- `insert(v)` - $O(\log n)$
- `extract_max()` - $O(\log n)$
- `create_heap(A)`
  - $O(n)$ version
  - $O(n \log n)$ version
- `heapsort()` - $O(n \log n)$

Heap - Operations: insert(v)

- Insert new element `v` at the index `n+1`
- Move forward in the heap to maintain the heap property
  - If the heap property is violated, swap `v` with its parent: *upward fix*

```
A[A.length] = v;
i = A.length - 1;
while(i > 1 && (A[parent(i)] < i)) {
  swap(A[parent(i)], i);
  i--;
}
```

Heap - Operations: extract_max()

- Read the root of the heap (`A[1]`)
- Replace the root with element at `A[n]`
- Swap until heap property is satisfied: *downward fix*

```
max_element = A[1]; //read root
A[1] = A[A.length - 1]; //replace with last elememt
i = 1;
A.length--;
// Swap until heap property is satisfied again
while (i < A.length){
  L = max(A[i]->children);
  if (A[i] < L){
    swap(A[i], L);
  }
}
```

- Worth noting that `L` is the largest element and not just the left or right children

Heap - Operations: create(A)

- Linear version - $O(n)$
  - Starts with an array `A` and assumes it is a binary max heap and then fixes the binary heap property starting from the last internal vertex (`A[A.length/2]`) back to the root
- Logarithm version - $O(n \log n)$
  - Starts with an empty binary heap and iterates over array `A` by using the `insert(v)` operation

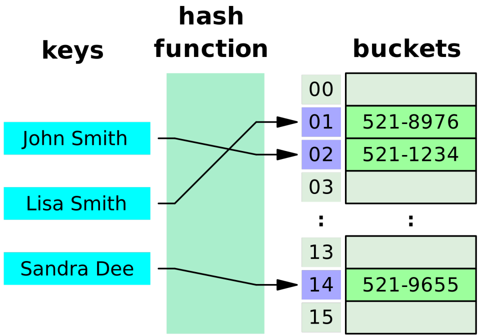
Heap - Operations: heapsort()

- Call `extract_max()` operation n times

```
for(i=0; i < A.length; i++)
  extract_max();
```

Index

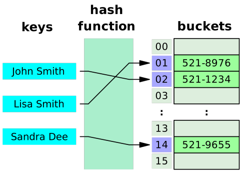
Hash Table



hash function
keys
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
:
13
14
15
521-8976
521-1234
:
521-9655

Hash Table

- Data structure designed to map key to values
- Uses a hash function to map keys into a range of integer indices



- High probability of two items colliding into the same index
- Collision resolution strategies:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Hash Table - Operations

- `search(v)`
- `insert(v)`
- `remove(v)`

Implementation via Direct Addressing Table (DAT):

- Initialize and empty boolean array `A` of size `m`:
  - `search(v)`: check if `A[v]` is `true` or `false`
  - `insert(v)`: set `A[v] = true`
  - `remove(v)`: set `A[v] = false`
- Keys cannot be negative and if possible, the range must be small

Hash Table - Operations

- `search(v)`
- `insert(v)`
- `remove(v)`

Implementation via Integer array:

- `h(v)` is the hash function applied to the key `v`:
  - `search(v)`: check if `A[h(v)] != -1`
  - `insert(v)`: set `A[h(v)] = v`
  - `remove(v)`: set `A[h(v)] = -1`

Hash Table

- Not recommended in programming contests unless absolute necessary
- Designing a well performing hash function is hard
- `C++ STL map` or `C++ STL set` are usually fast for typical programming contest input ($1M$)
- However, Hash Tables are faster: $O(1)$ operations
- `C++ STL unordered_map`

Index

CPCFI

UVa - Competitive Programming 3

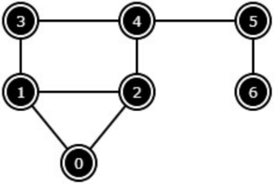
- **CP3 > Data Structures and Libraries > Non Linear Data Structures with Built-in Libraries**
- https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=630

Table of Contents

Index

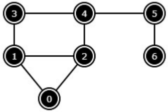
CPCFI

Graph



Graph

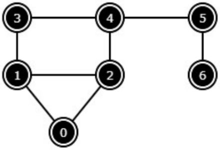
- Set of vertices (V) and edges (E): $G = (V, E)$



- Undirected graph: vertices have no direction
- Directed graph: vertices have direction
- Weighted graph: edges have a numerical values
- Unweighted graph: edges do not have numerical values
- Simple graph: no loops and no multiple edges between two vertices

Graph - Terminology I

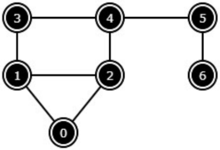
- An undirected edge $e : (u, v)$ is said to be **incident** with its two end-point vertices: u and v
- Two vertices are called **neighbors** if they are incident with a common edge



- Vertices 0 and 2 are neighbors
- The **degree** of vertex v in an undirected graph is the number of edges incident with v

Graph - Terminology II

- A **path** in an undirected graph G is a sequence of vertices such that there is an edge between v_i and v_{i+1} $\forall i \in [0, \ldots, n-1]$



- One possible path: $0, 1, 3, 4, 5$
- A **simple path** is a path where vertices are not repeated

Graph - Terminology III

- An undirected graph G is called **connected** if there is a path between every pair of distinct vertices of G

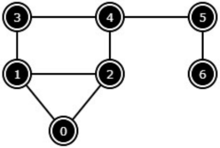


Figure: Connected graph

Graph - Terminology IV

- An undirected graph C is a **connected component** of the undirected graph G if:
  1. C is a subgraph of G
  2. C is connected
  3. No connected subgraph of G has C as a subgraph and contains vertices or edges that are not in C

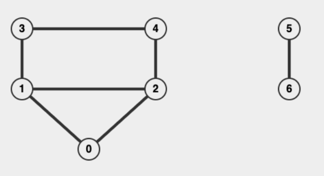
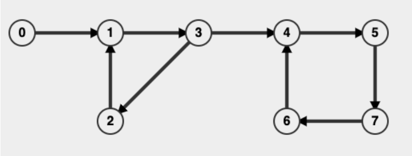


Figure: Graph G with two connected components

Graph - Terminology V

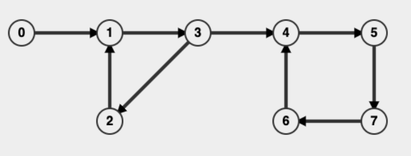
Adjustments for directed graphs:

- If we have a directed edge $e : (u \rightarrow v)$ we say that v is adjacent to u but not necessarily the other way around
- We have to differentiate the **degree** of a vertex into *in-degree* and *out-degree*



Graph - Terminology VI

- A directed graph is **strongly connected** if there is a path in each direction between each pair of vertices, i.e., every vertex is reachable from every other vertex
- A **strongly connected component** is a subgraph of a directed graph that is strongly connected



Graph - Terminology VII

- **Cycle**: path that starts and ends in the same vertex
- **Acyclic graph**: graph that does not contains cycles
- **Directed Acyclic Graph (DAG)**: directed graph that is also acyclic

Graph - Termninology VII

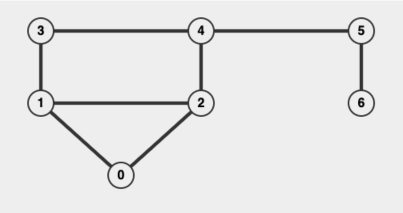
Types of graphs:

1. Undirected - Unweighted
2. Undirected - Weighted
3. Directed - Unweighted
4. Directed - Weighted

CPCFI

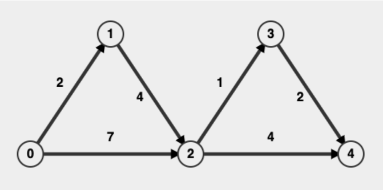
Example 1: social network

- Vertices can represent people and edges the connection between them
- Who is friend with who? Who has the most friends? Is there any isolated people? Is there a common friend between two strangers?



Example 2: transportation network

- Vertices represents stations and edges connection between them (roads with weights)
- What is the path with the least amount of time between station 0 and station 4?



Special Graphs I: Rooted Tree

- **Connected** and **acyclic** graph with V vertices and $E = V - 1$ edges
- One **unique** path between any pair of vertices

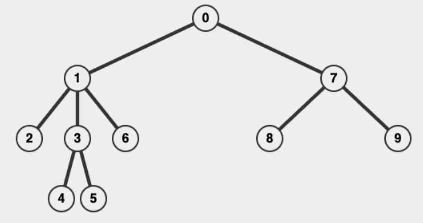


Figure: Rooted tree

Special Graphs I: Non Rooted Tree

- Not all trees are drawn with a root vertex at top and leaf vertices at the bottom

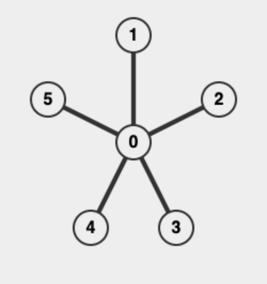


Figure: Non rooted tree

Special Graphs I: Binary Tree

- Rooted tree in which every vertex hast at most two children (`left` and `right`)
- **Full binary tree**: binary tree in which every non-leaf node has exactly two children
- **Complete binary tree**: binary tree in which every level is completely filled

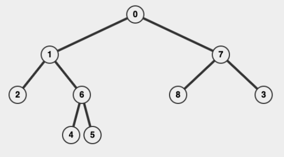
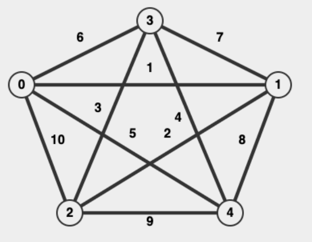


Figure: Full binary tree

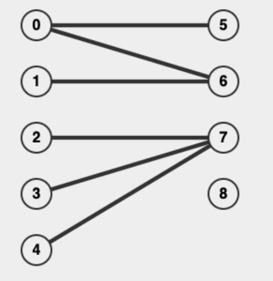
Special Graphs II: Complete graph

- Graph with V vertices and $E = \frac{V(V-1)}{2}$ edges
- There is an edge between any pair of vertices



Special Graphs III: Bipartite

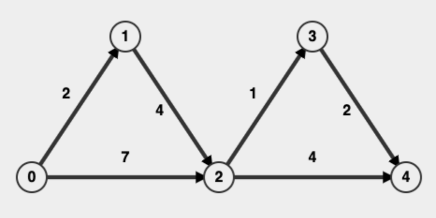
- Undirected graph with V vertices that can be partitioned into two disjoint sets[3] of size m and n where $V = m + n$
- There is no edge between members of the same set
- Bipartite graph can also be complete



[3] Two sets are disjoint if they have no element in common

Special Graphs IV: DAG

- Directed Acyclic Graphs
- Each DAG has at least one topological order



Graph as a Data Structure

There are three main ways to store a graph (nodes, edges) in a data structure, can you think of one?

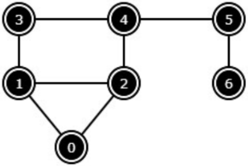
Graph as a Data Structure

Main ways to store a graph in a data structure:

1. Adjacency Matrix
2. Adjacency List
3. Edge List

1. Adjacency Matrix

- An adjacency matrix M is a square matrix where the entry $M[i][j]$ corresponds to the edge's weight from vertex i to vertex j
- For unweighted graphs we can set a unit weight 1 for all edges



Adjacency Matrix

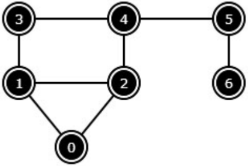
	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

- We simply use a 2×2 `C++ array`
- Space: $O(V^2)$ where V is the number of vertices

CPCFI

2. Adjacency List

- Array L of V lists, on for each vertex
- $L[i]$ stores the list of vertex i's neighbors
- Space: $O(V+E)$. Much more efficient than adjacency matrix M



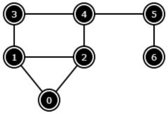
Adjacency List			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

3. Edge List

- E is a collection of edges with both connecting vertices and their weights
- Usually the edges are sorted by increasing weight

Graph implementation

- ch_07_graph.cpp

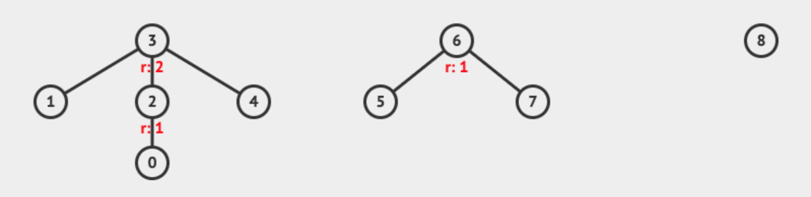


Index

CPCFI

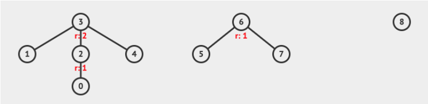
Union-Find Disjoint Sets (UFDS)

Two sets are disjoint if they have no element in common.



Union-Find Disjoint Sets

- Data structure to model a collection of disjoint sets with the ability to efficiently (in $O(1)$):
  1. Determine which set an item belongs to
  2. Unite two disjoint sets into one larger set



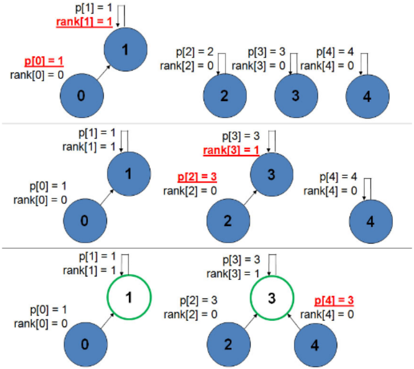
- Applications: finding connected components in an undirected graph
- `ch2_08_unionfind_ds.cpp`

UFDS intuition

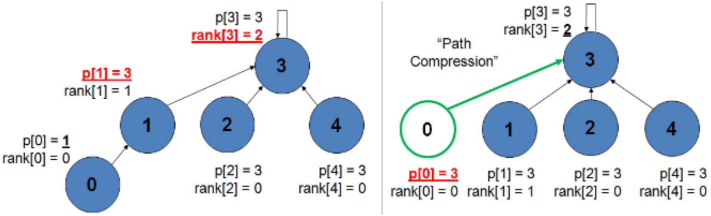
- Disjoint sets are represented as trees
- Each disjoint set has a representative item (root of the tree)
- UFDS creates a tree structure where the disjoint sets form a forest

UFDS - Example

- 5 disjoint sets: $\{0, 1, 2, 3, 4\}$



UFDS - Example (continuation)



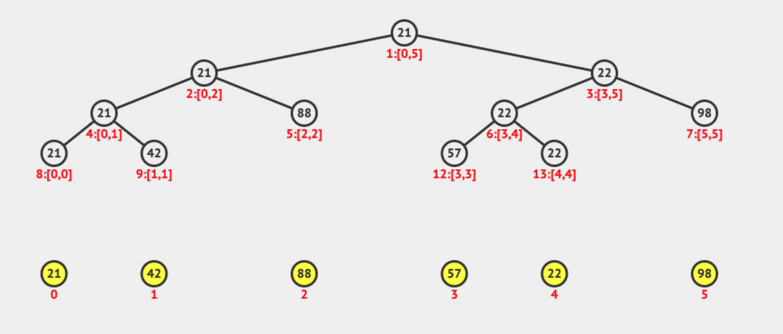
UFDS - Operations

- Each node of the tree contains these two elements:
  1. `p`: index of the representative item
  2. `rank`: height of the disjoint set
- `unionSet(i,j)`: unites to disjoint sets by setting the representative item (root) of one disjoint set to be the new parent of the representative item of the other disjoint set. This will cause that both *i* and *j* have the same representative item
- `findSet(i)`: finds the representative item for node *i*
- `isSameSet(i,j)`: determines if items *i* and *j* belong to the same set. This can be done by calling `findSet(i)` and `findSet(j)` and checking if both are equal

Index

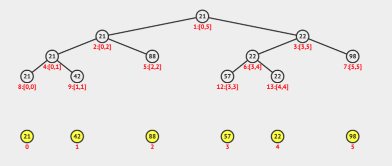
CPCFI

Segment Tree



Segment Tree

- Data structure that can efficiently answer dynamic [4] range queries



- One example of range queries is the problem of finding the index of the minimum element in the array within a range `[i,j]`. This problem is called Range Minimum Query (RMQ)
- `ch2_09_segmenttree_ds.cpp`

[4] Dynamic problems are the ones in which we need to frequently update the data; thus, making pre-processing techniques useless

Segment Tree - Implementation

- `build` routine ($O(2n)$):
  - 1-based compact array `st` where index 1 is the root and the left and right children of index p are indices $2 \cdot p$ and $(2 \cdot p) + 1$ respectively
  - The value of `st[p]` is the RMQ value of the segment associated with index `p`
  - The root (index 1 of `st`) represents segment `[0,n-1]`
  - For each segment `[L,R]` stored in index `p` where `L!=R`, the segment will be split into `[L, (L+R)/2]` and `[(L+R)/2+1, R]`

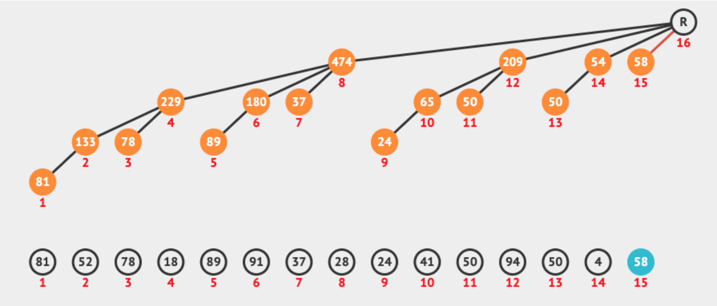
Segment Tree - Implementation II

- Answering an RMQ can be done in $O(\log n)$
- `RMQ(i,i) = i` where `i` is an index of `st`
- `RMQ(i,j)` where `i!=j`:
  - `rmq` routine:
    - Let `p1=rmq(i, (i+j)/2)` and `p2=rmq((i+j)/2+1,j)`
    - `RMQ(i,j)` will be `p1` if `st[p1] <= st[p2]` or `p2` otherwise

CPCFI

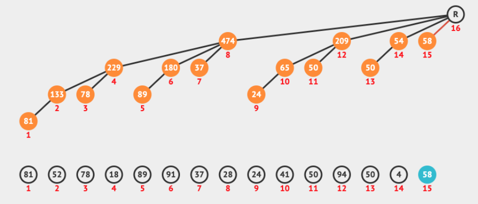
Index

Binary Indexed (Fenwick) Tree



Binary Indexed (Fenwick) Tree

- Invented by Peter Fenwick in 1984
- Useful data structure for implementing dynamic cumulative frequency tables (see example on the next slide)
- Fenwick tree operations are extremely efficient since the use bit manipulation techniques



- `ch2_10_fenwicktree_ds.cpp`

Fenwick Tree - Example

- Suppose we have the test scores of `m=11` students
- `f={2,4,5,5,6,6,6,7,7,8,9}`
- Test scores are integer values in `[1,10]`

Index/ Score	Frequency `f`	Cumulative Frequency `cf`	Short Comment
0	-	-	Index 0 is ignored (as the sentinel value).
1	0	0	`cf[1] = f[1] = 0`.
2	1	1	`cf[2] = f[1] + f[2] = 0 + 1 = 1`.
3	0	1	`cf[3] = f[1] + f[2] + f[3] = 0 + 1 + 0 = 1`.
4	1	2	`cf[4] = cf[3] + f[4] = 1 + 1 = 2`.
5	2	4	`cf[5] = cf[4] + f[5] = 2 + 2 = 4`.
6	3	7	`cf[6] = cf[5] + f[6] = 4 + 3 = 7`.
7	2	9	`cf[7] = cf[6] + f[7] = 7 + 2 = 9`.
8	1	10	`cf[8] = cf[7] + f[8] = 9 + 1 = 10`.
9	1	11	`cf[9] = cf[8] + f[9] = 10 + 1 = 11`.
10	0	11	`cf[10] = cf[9] + f[10] = 11 + 0 = 11`.

- The problem becomes evident if the frequency of previously seen scores suffers an update. In this case, we would have to start all over again from the start in $O(n)$ time

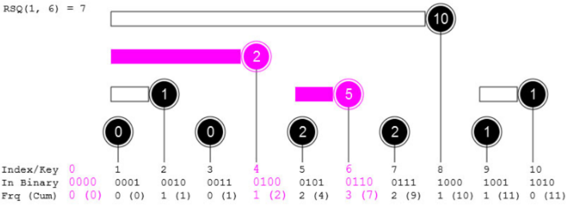
Fenwick Tree - Implementation

- Typically implemented as a dynamic array (vector)
- A Fenwick Tree `ft` is a tree indexed by the bits of its integer keys
  - The keys fall within a fixed range `[1,n]` [5]
- The element at index `i` is responsible for elements in the range `[i-LSOne(i)+1..i]`
- `ft[i]` stores the cumulative frequency of elements `{i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, ..., i}`
- `LSOne(i) = (i & (-i))` produces the Least Significant One-bit in `i`

[5] Note that we skip index 0

Fenwick Tree - Implementation II

- If we want to obtain the cumulative frequency between `[1,...,b]`, we simply add `ft[b]`,`ft[b']`,`ft[b'']`,... until index b^i is 0
- `b' = b - LSOne(b)`
- This process runs in $O(\log n)$ when `b=n`



- To evaluate the cumulative frequency between two indices `[a...b]` where `a != 1` we simply evaluate `rsq(a, b) = rsq(b) - rsq(a-1)`

Index

UVa - Competitive Programming 3

- **CP3 > Data Structures and Libraries > Data Structures with Our-Own Libraries**
- https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=634

CPCFI

References

- Halim S., Halim F., *Competitive Programming 3*, Handbook for ACM ICPC and IOI Contestants. 2013
- Stroustrup B. *The C++ Programming Language*. Fourth ed.
- Skiena S. *The Algorithm Design Manual*. Springer. 2020

CPCFI