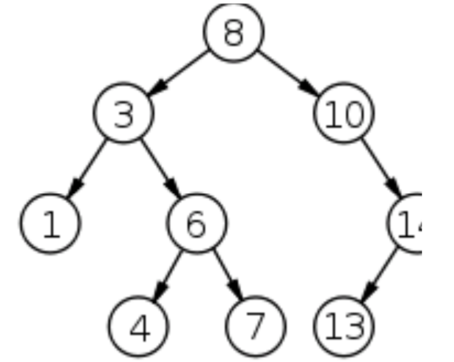
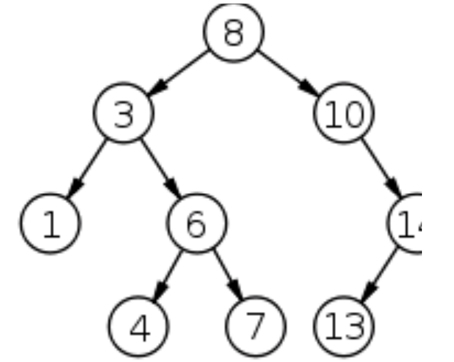
Binary Search Tree (1)



A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

- ADT Table (key → data)
- Binary Search Tree (BST)
  - Advertised O(log n) for insert, search, and delete
  - Requirement: the BST must be **balanced**!
    - AVL tree, Red-Black Tree, etc… *argh*
- Fret not, just use: C++ STL **map** (Java **TreeMap**)
  - UVa 10226 (Hardwood Species)*

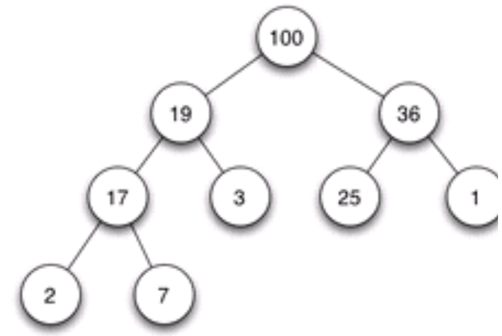
Binary Search Tree (2)



A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

- ADT Table (key exists or not)
- Set (Single Set)
  - C++ STL **set**, similar to C++ STL **map**
    - map stores a **(key, data)** pair
    - set stores just the **key**
  - In Java: **TreeSet**
- Example:
  - UVa 11849 – CD

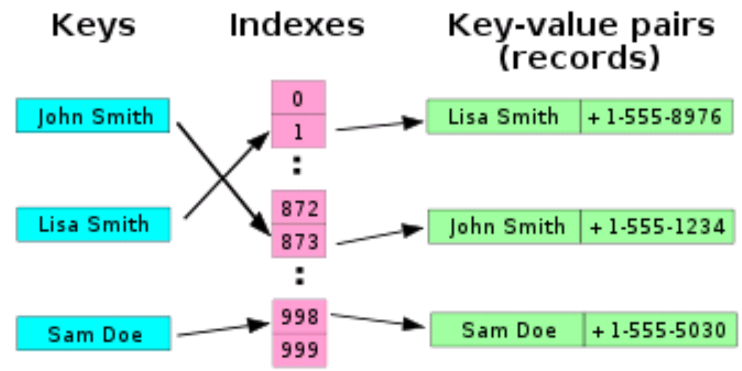
Heap



Example of a full binary max heap

- Heap
  - C++ STL **algorithm** has some heap algorithms
    - partial_sort uses heapsort
  - C++ STL **priority_queue** (Java **PriorityQueue**) is heap
    - Prim's and Dijkstra's algorithms use priority queue
- But, we rarely see pure heap problems in ICPC

Hash Table



A small phone book as a hash table.

- Hash Table
  - Advertised O(1) for insert, search, and delete, but:
    - The hash function must be good!
    - There is no Hash Table in C++ STL (∃ in Java API)
  - Nevertheless, O(log n) using **map** is usually ok
- Direct Addressing Table (DAT)
  - Rather than hashing, we more frequently use DAT
  - UVa 11340 (Newspaper)

Quick Check

1. I can cope with this pace…
2. I am lost with so many new information in the past few slides

0 of 120

5 Minutes Break

- One data structures *without* built-in libraries will be discussed in the last part…
  - Binary Indexed (Fenwick) Tree
  - Graph, Union-Find Disjoint Sets, and Segment Tree are not discussed in this year's CS3233 Week02
    - Graph DS is covered in details in CS2010/CS2020
    - UFDS is covered briefly in CS2010/CS2020
    - Please study Segment Tree on your own
      - We *try* not to set any contest problem involving Segment Tree

Time Check
8.30pm

Graph (not discussed today, revisited in Week06/07/08)

Union-Find Disjoint Sets (not discussed today, read Ch2 on your own)

Segment Tree (not discussed today, read Ch2 on your own)

Fenwick Tree (discussed today)

DATA STRUCTURES WITHOUT BUILT-IN LIBRARIES

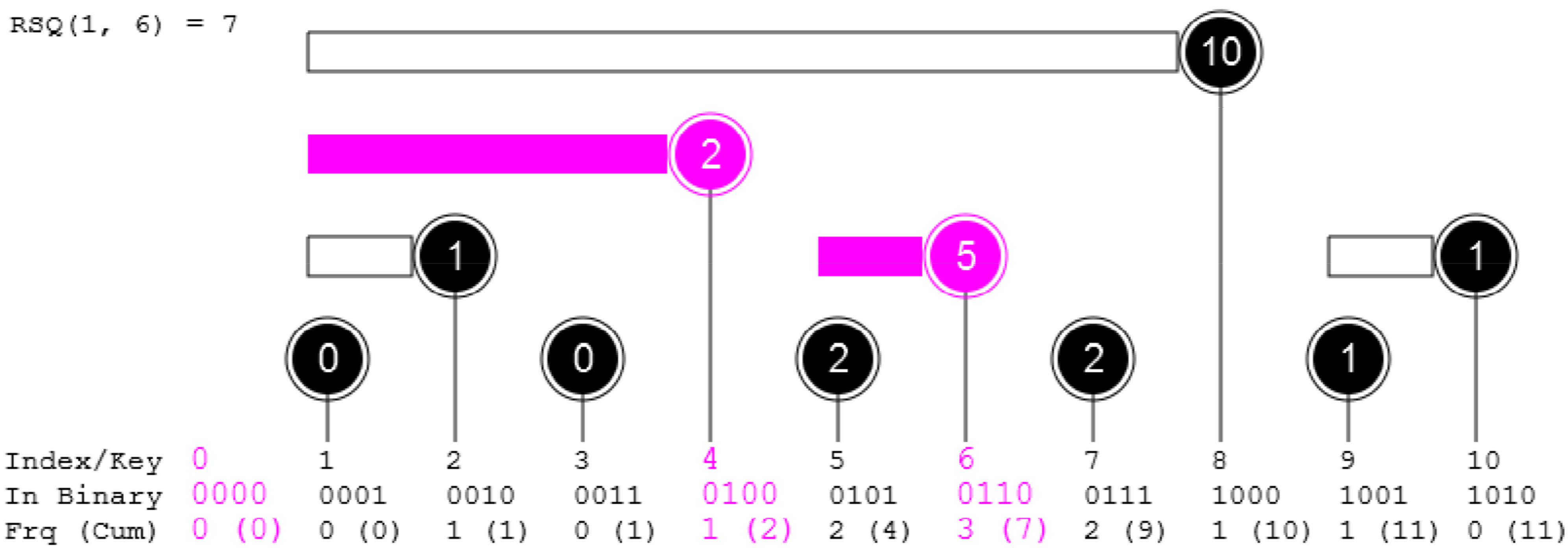
Fenwick Tree – Basics (1)

- Cumulative Frequency Table
  - Example, s = {2,4,5,5,6,6,6,7,7,8,9} (already sorted)

Index/Score/Symbol	Frequency	Cumulative Frequency
0	-	- (index 0 is ignored)
1	0	0
2	1	1
3	0	1
4	1	2
5	2	
6	3	
7	2	
8	1	
9	1	
10	0	

Fenwick Tree – Basics (2)

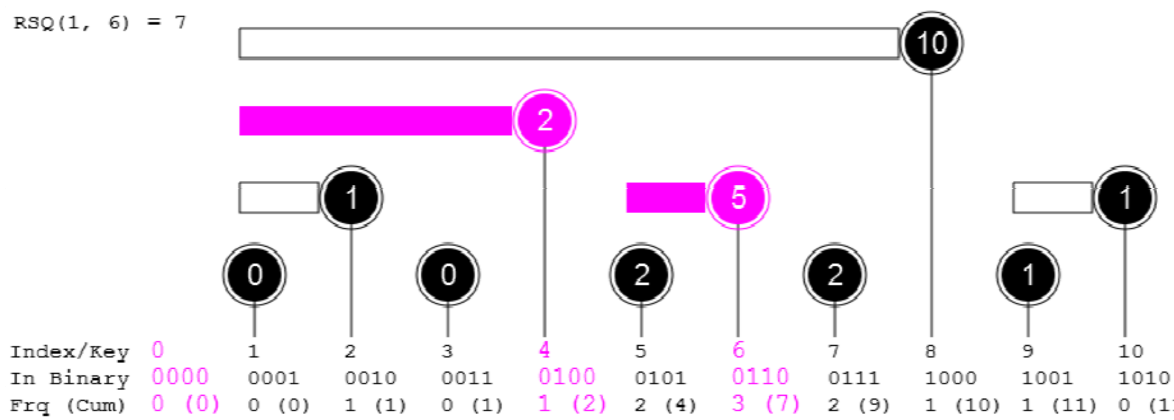
- Fenwick Tree (inventor = Peter M. Fenwick)
  - Also known as “**Binary Indexed** Tree”, very *aptly* named
  - Implemented as an **array**, let call the array name as **ft**
  - We will frequently use this bit manipulation, remember!
    - **LSOne(i)** = Least Significant One of **i** computed via **i & (-i)**



Fenwick Tree – Basics (3)

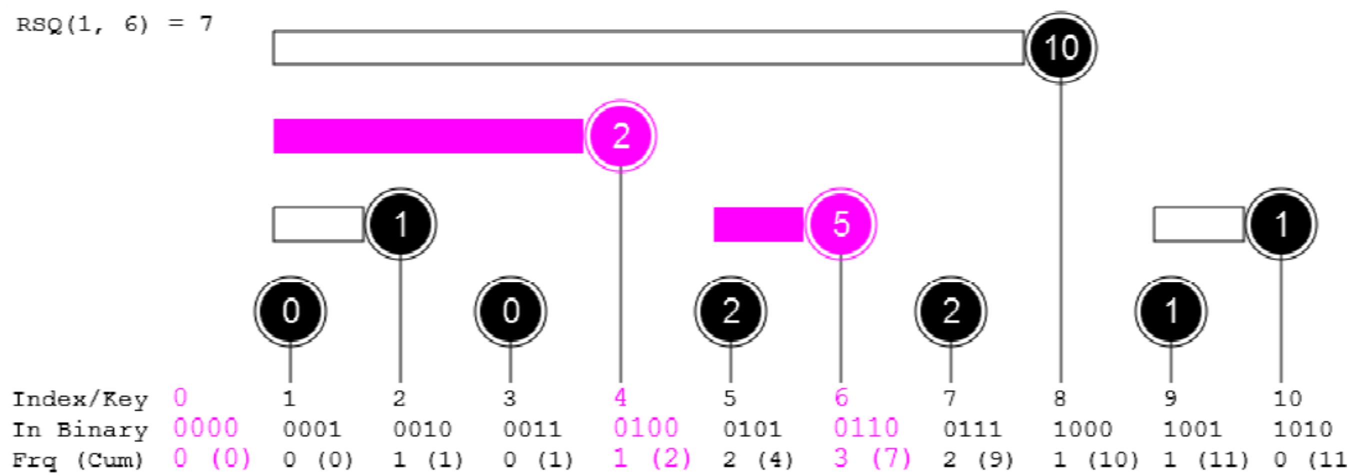
- Each index **i** of **ft** is responsible for certain **range: [i-LSOne(i)+1 .. i]**
- **ft[i]** stores the cumulative frequency of elements: **{i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, ..., i}**

Key/Idx	Binary	Range	F	CF	FT
0	**0000**	**N/A**	**N/A**	**N/A**	**N/A**
1	0001	[1..1]	0	0	0
2	0010	[1..2]	1	1	1
3	0011	[3..3]	0	1	0
4	0100	[1..4]	1	2	2
5	0101	[5..5]	2	4	2
6	0110	[5..6]	3	7	5
7	0111	[7..7]	2	9	2
8	1000	[1..8]	1	10	10
9	1001	[9..9]	1	11	1
10	1010	[9..10]	0	11	1



Fenwick Tree – RSQ (1)

- To get the cumulative frequency from index **1** to `b`, use `rsq(b)`
  - The answer is the sum of sub-frequencies stored in array `ft` with indices related to `b` via this formula `b' = b - LSOne(b)`
  - Apply this formula iteratively until `b` is `0`
  - Example: `rsq(6)`
    - b = 6 = 0110, b’ = b - LSOne(b) = 0110 - 0010, b' = 4 = 0100
    - b' = 4 = 0100, b’’ = b’ - LSOne(b’) = 0100 - 0100, b'' = 0 → **stop**
    - Sum **ft[6]+ft[4] = 5+2 = 7 (the pink area covers range [1..4]+[5..6] = [1..6])**

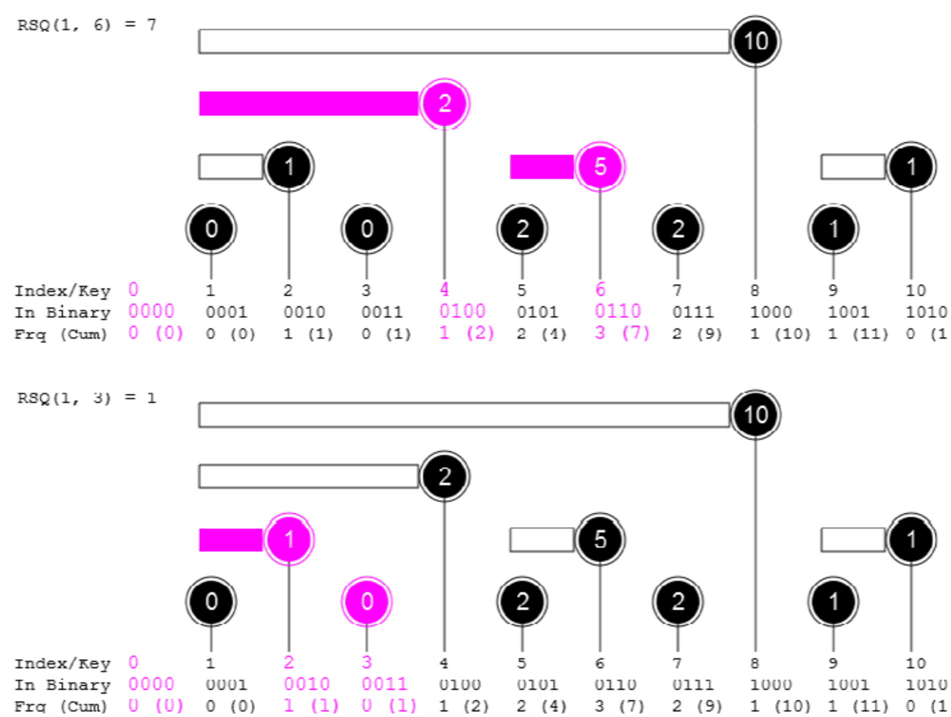


Analysis: This is O(log n)

Why?

Fenwick Tree – RSQ (2)

- To get the cumulative frequency from index **a** to `b`, use `rsq(a, b)`
  - If `a` is greater than one, we use: `rsq(b) – rsq(a-1)`
  - Example: `rsq(4, 6)`
    - `rsq(4, 6) = rsq(6) – rsq(4-1) = rsq(6) – rsq(3) = (5+2) - (0+1) = 7 - 1 = 6`



Analysis:
This is
O(2 log n) =
O(log n)

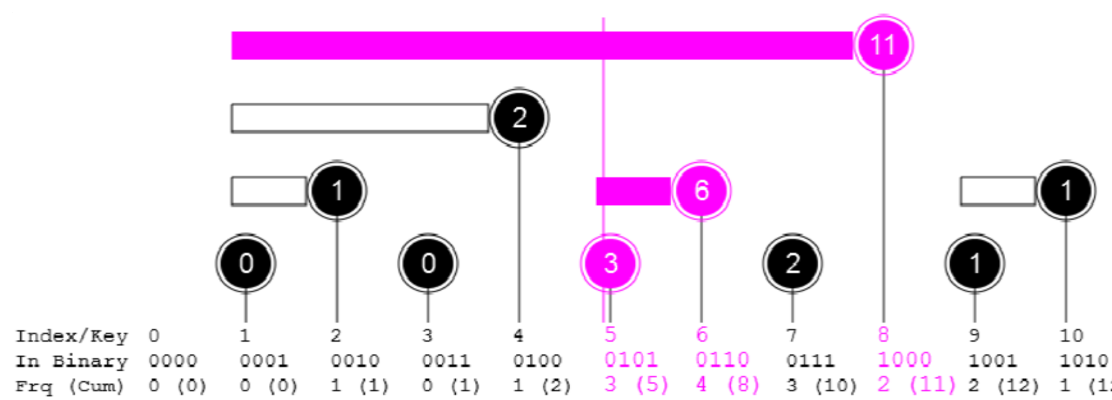
Why?

Fenwick Tree – Update

- To update the frequency of an key/index `k` by `v` (`v` is either positive or negative), use `adjust(k, v)`
  - Indices that are related to `k` via `k' = k + LSOne(k)` will be updated by `v` when `k < ft.size()`
  - Example: `adjust(5, 1)`
    - k = 5 = 0101, k' = k + LSOne(k) = 0101 + 0001, k' = 6 = 0110
    - k' = 6 = 0110, k'' = k' + LSOne(k') = 0110 + 0010, k'' = 8 = 1000
    - k'' = 8 = 1000, k''' = k'' + LSOne(k'') = 1000 + 1000, k''' = 16 = 10000 → stop
  - Observe that the **pink line** in the figure below **stabs through** the ranges that are under the responsibility of indices 5, 6, and 8
    - ft[5], 2 updated to 3
    - ft[6], 5 updated to 6
    - ft[8], 10 updated to 11

Analysis:
This is also
O(log n)

Why?



Fenwick Tree – Library

```
class FenwickTree {
private: vi ft;                   // recall that vi is: typedef vector<int> vi;
public: FenwickTree(int n) { ft.assign(n + 1, 0); }    // init n + 1 zeroes
  int rsq(int b) {                                     // returns RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }                              // note: LSOne(S) (S & (-S))
  int rsq(int a, int b) {                              // returns RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }
  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) {                    // note: n = ft.size() - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};
```

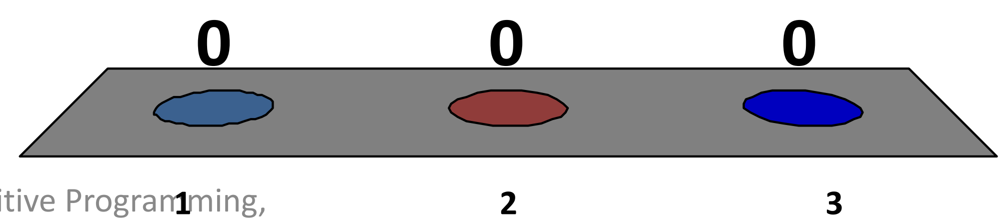

FT/BIT is in IOI syllabus!

Fenwick Tree – Sample Application

- Fenwick Tree is very suitable for *dynamic* RSQs (cumulative frequency table) where update occurs on a certain index only
- Now, think of potential real-life applications!
  - http://uhunt.felix-halim.net/id/32900
  - Consider code runtime of [0.000 - 9.999]s for a particular UVa problem
    - There are up to 10+ million submissions/codes
      - About thousands submissions per problem
    - If your code runs in 0.342 secs, what is your rank?
- How to use Fenwick Tree to deal with this problem?

Quick Check

1. I am lost with Fenwick Tree
2. I understand the basics of Fenwick Tree, but since this is new for me, I may/may not be able to recognize problems solvable with FT
3. I have solved several FT-related problems before



0 of 120

Summary

- There are a lot of *great* Data Structures out there
  - We need the most efficient one for our problem
    - Different DS suits different problem!
- Many of them have **built-in libraries**
  - For some others, we have to build **our own (focus on FT)**
    - Study these libraries! Do not rebuild them during contests!
- From Week03 onwards and future ICPCs/IOIs, use C++ STL and/or Java API and our built-in libraries!
  - Now, your team should be in rank 30-45 (from 60) (still solving ~1-2 problems out of 10, but faster)

References

- **Competitive Programming 2.9**, Chapter 2
  - Steven, Felix ☺
- **A new data structure for cumulative frequency table**
  - Peter M Fenwick
  - http://www.uop.edu.jo/download/pdfcourses/ds/19492.pdf
- **Fenwick Tree @ TopCoder**
  - By boba5551
  - http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees

Study These Visualizations

- http://www.comp.nus.edu.sg/~stevenha/visualization/bitmask.html
- http://www.comp.nus.edu.sg/~stevenha/visualization/bit.html

- You can use your smart phones/tablet PCs to access them ☺
- Google searches (as of last year), there is no other visualizations on bitmask/BIT like these
- PS: Report bugs to Steven, if any