# Data structures and libraries

Bjarki Ágúst Guðmundsson
Tómas Ken Magnússon

School of Computer Science
Reykjavík University

Árangursrík forritun og lausn verkefna

# Today we're going to cover

- Basic data types
- Big integers
- Why we need data structures
- Data structures you already know
- Sorting and searching
- Using bitmasks to represent sets
- Common applications of the data structures
- Augmenting binary search trees
- Representing graphs

# Basic data types

- You should all be familiar with the basic data types:
  - `bool`: a boolean (`true`/`false`)
  - `char`: an 8-bit signed integer (often used to represent characters with ASCII)
  - `short`: a 16-bit signed integer
  - `int`: a 32-bit signed integer
  - `long long`: a 64-bit signed integer
  - `float`: a 32-bit floating-point number
  - `double`: a 64-bit floating-point number
  - `long double`: a 128-bit floating-point number
  - `string`: a string of characters

# Basic data types

| Type | Bytes | Min value | Max value |
|---|---|---|---|
| bool | 1 | | |
| char | 1 | -128 | 127 |
| short | 2 | -32768 | 32767 |
| int | 4 | -2148364748 | 2147483647 |
| long long | 8 | -9223372036854775808 | 9223372036854775807 |
| | *n* | $-2^{8n-1}$ | $2^{8n-1} - 1$ |

| Type | Bytes | Min value | Max value |
|---|---|---|---|
| unsigned char | 1 | 0 | 255 |
| unsigned short | 2 | 0 | 65535 |
| unsigned int | 4 | 0 | 4294967295 |
| unsigned long long | 8 | 0 | 18446744073709551615 |
| | *n* | $0$ | $2^{8n} - 1$ |

| Type | Bytes | Min value | Max value | Precision |
|---|---|---|---|---|
| float | 4 | $\approx -3.4 \times 10^{-38}$ | $\approx 3.4 \times 10^{-38}$ | $\approx 7$ digits |
| double | 8 | $\approx -1.7 \times 10^{-308}$ | $\approx 1.7 \times 10^{-308}$ | $\approx 14$ digits |

# Big integers

- What if we need to represent and do computations with very large integers, i.e. something that doesn't fit in a `long long`
- Simple idea: Store the integer as a string
- But how do we perform arithmetic on a pair of strings?
- We can use the same algorithms as we learned in elementary school
  - Addition: Add digit-by-digit, and maintain the carry
  - Subtraction: Similar to addition
  - Multiplication: Long multiplication
  - Division: Long division
  - Modulo: Long division

# Example problem: Integer Inquiry

- http://uva.onlinejudge.org/external/4/424.html

# Why do we need data structures?

- Sometimes our data needs to be organized in a way that allows one or more of
  - Efficient querying
  - Efficient inserting
  - Efficient deleting
  - Efficient updating
- Sometimes we need a better way to represent our data
  - How do we represent large integers?
  - How do we represent graphs?
- Data structures help us achieve those things

# Data structures you've seen before

- Static arrays
- Dynamic arrays
- Linked lists
- Stacks
- Queues
- Priority Queues
- Sets
- Maps

# Data structures you've seen before

- Static arrays - int arr[10]
- Dynamic arrays - vector<int>
- Linked lists - list<int>
- Stacks - stack<int>
- Queues - queue<int>
- Priority Queues - priority_queue<int>
- Sets - set<int>
- Maps - map<int, int>

# Data structures you've seen before

- Static arrays - int arr[10]
- Dynamic arrays - vector<int>
- Linked lists - list<int>
- Stacks - stack<int>
- Queues - queue<int>
- Priority Queues - priority_queue<int>
- Sets - set<int>
- Maps - map<int, int>

- Usually it's best to use the standard library implementations
  - Almost surely bug-free and fast
  - We don't need to write any code
- Sometimes we need our own implementation
  - When we want more flexibility
  - When we want to customize the data structure

# Sorting and searching

- Very common operations:
  - Sorting an array
  - Searching an unsorted array
  - Searching a sorted array
- Again, usually in the standard library
- We'll need different versions of binary search later which need custom code, but lower_bound is enough for now

# Sorting and searching

- Very common operations:
  - Sorting an array - sort(arr.begin(), arr.end())
  - Searching an unsorted array - find(arr.begin(), arr.end(), x)
  - Searching a sorted array - lower_bound(arr.begin(), arr.end(), x)
- Again, usually in the standard library
- We'll need different versions of binary search later which need custom code, but lower_bound is enough for now

# Representing sets

- We have a small ($n \leq 30$) number of items
- We label them with integers in the range $0, 1, \ldots, n-1$
- We can represent sets of these items as a 32-bit integer
- The $i$th item is in the set represented by the integer $x$ if the $i$th bit in $x$ is $1$
- Example:
  - We have the set $\{0, 3, 4\}$
  - `int x = (1<<0) | (1<<3) | (1<<4);`

# Representing sets

- Empty set:

```
0
```

- Single element set:

```
1<<i
```

- The universe set (i.e. all elements):

```
(1<<n)-1
```

- Union of sets:

```
x|y
```

- Intersection of sets:

```
x&y
```

- Complement of a set:

```
~x & ((1<<n)-1)
```

# Representing sets

- Check if an element is in the set:

```
if (x & (1<<i)) {
    // yes
} else {
    // no
}
```

# Representing sets

- Why do this instead of using set<int>?
- Very lightweight representation
- All subsets of the *n* elements can be represented by integers in the range $0 \ldots 2^n - 1$
- Allows for easily iterating through all subsets (we'll see this later)
- Allows for easily using a set as an index of an array (we'll see this later)

# Applications of Arrays and Linked Lists

- Too many to list
- Most problems require storing data, usually in an array

# Example problem: Broken Keyboard

- http://uva.onlinejudge.org/external/119/11988.html

# Applications of Stacks

- Processing events in a first-in first-out order
- Simulating recursion
- Depth-first search in a graph
- Reverse a sequence
- Matching brackets
- And a lot more

# Applications of Queues

- Processing events in a first-in first-out order
- Breadth-first search in a graph
- And a lot more

# Applications of Priority Queues

- Processing events in order of priority
- Finding a shortest path in a graph
- Some greedy algorithms
- And a lot more

# Applications of Sets

- Keep track of distinct items
- Have we seen an item before?
- If implemented as a binary search tree:
  - Find the successor of an element (the smallest element that is greater than the given element)
  - Count how many elements are less than a given element
  - Count how many elements are between two given elements
  - Find the $k$th largest element
- And a lot more

# Applications of Maps

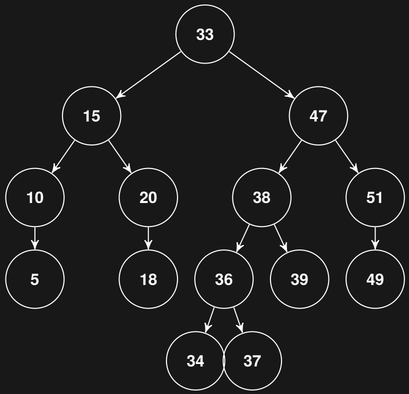
- Associating a value with a key
- As a frequency table
- As a memory when we're doing Dynamic Programming (later)
- And a lot more

# Augmenting Data Structures

- Sometimes we can store extra information in our data structures to gain more functionality
- Usually we can't do this to data structures in the standard library
- Need our own implementation that we can customize
- Example: Augmenting binary search trees

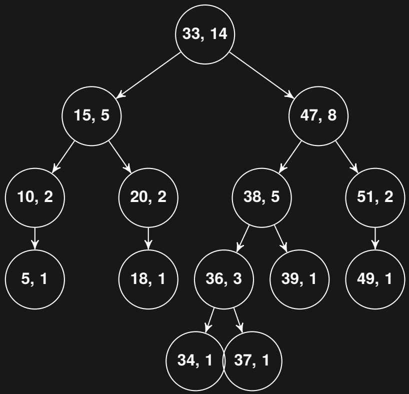
# Augmenting Binary Search Trees

- We have a binary search tree and want to efficiently:
  - Count number of elements $< x$
  - Find the $k$th smallest element
- Naive method is to go through all vertices, but that is slow: $O(n)$



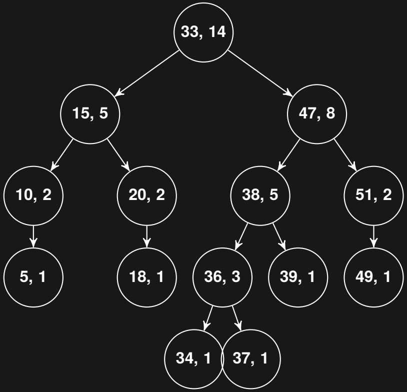
# Augmenting Binary Search Trees

- Idea: In each vertex store the size of the subtree
- This information can be maintained when we insert/delete elements without adding time complexity



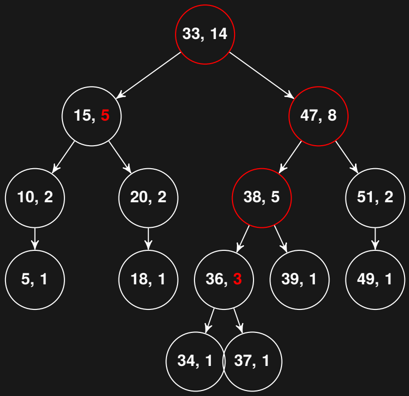
# Augmenting Binary Search Trees

- Count number of elements $< 38$
  - Search for $38$ in the tree
  - Count the vertices that we pass by that are less than *x*
  - When we are at a vertex where we should go right, get the size of the left subtree and add it to our count



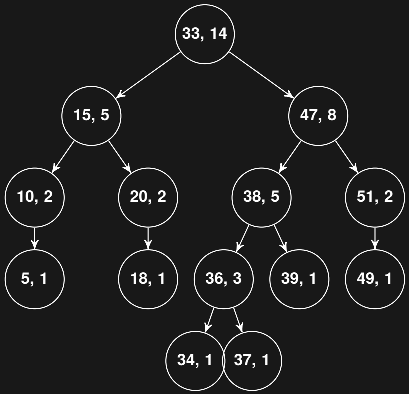
# Augmenting Binary Search Trees

- Count number of elements $< 38$
  - Search for $38$ in the tree
  - Count the vertices that we pass by that are less than $x$
  - When we are at a vertex where we should go right, get the size of the left subtree and add it to our count
- Time complexity $O(\log n)$



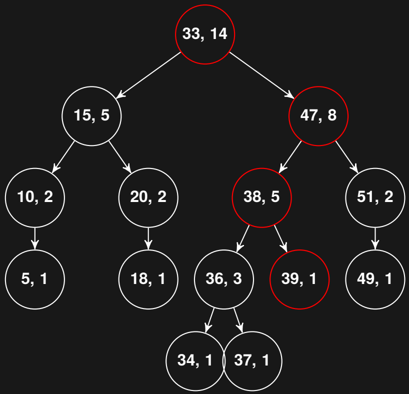
# Augmenting Binary Search Trees

- Find *k*th smallest element
  - We're on a vertex whose left subtree is of size *m*
  - If $k = m + 1$, we found it
  - If $k \leq m$, look for the *k*th smallest element in the left subtree
  - If $k > m + 1$, look for the $k - m - 1$st smallest element in the right subtree



# Augmenting Binary Search Trees

- Find $k$th smallest element
  - We're on a vertex whose left subtree is of size $m$
  - If $k = m + 1$, we found it
  - If $k \leq m$, look for the $k$th smallest element in the left subtree
  - If $k > m + 1$, look for the $m - k - 1$st smallest element in the right subtree
- Example: $k = 11$



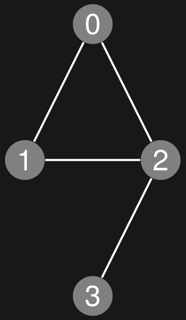
# Representing graphs

- There are many types of graphs:
  - Directed vs. undirected
  - Weighted vs. unweighted
  - Simple vs. non-simple
- Many ways to represent graphs
- Some special graphs (like trees) have special representations
- Most commonly used (general) representations:
  1. Adjacency list
  2. Adjacency matrix
  3. Edge list

# Adjacency list

```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2
```

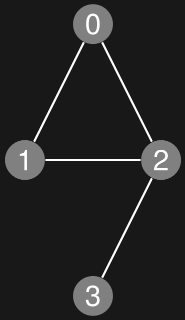
```
vector<int> adj[4];
adj[0].push_back(1);
adj[0].push_back(2);
adj[1].push_back(0);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(2);
adj[3].push_back(2);
```



# Adjacency matrix

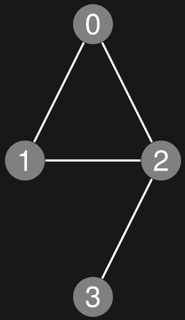
```
0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
```

```
bool adj[4][4];
adj[0][1] = true;
adj[0][2] = true;
adj[1][0] = true;
adj[1][2] = true;
adj[2][0] = true;
adj[2][1] = true;
adj[2][3] = true;
adj[3][2] = true;
```



# Edge list

```
0, 1
0, 2
1, 2
2, 3
```




```
vector<pair<int, int> > edges;
edges.push_back(make_pair(0, 1));
edges.push_back(make_pair(0, 2));
edges.push_back(make_pair(1, 2));
edges.push_back(make_pair(2, 3));
```

# Efficiency

| | Adjacency list | Adjacency matrix | Edge list |
|---|---|---|---|
| Storage | $O(\|V\| + \|E\|)$ | $O(\|V\|^2)$ | $O(\|E\|)$ |
| Add vertex | $O(1)$ | $O(\|V\|^2)$ | $O(1)$ |
| Add edge | $O(1)$ | $O(1)$ | $O(1)$ |
| Remove vertex | $O(\|E\|)$ | $O(\|V\|^2)$ | $O(\|E\|)$ |
| Remove edge | $O(\|E\|)$ | $O(1)$ | $O(\|E\|)$ |
| Query: are $u, v$ adjacent? | $O(\|V\|)$ | $O(1)$ | $O(\|E\|)$ |

- Different representations are good for different situations

# Example problem: Easy Problem from Rujia Liu?

- http://uva.onlinejudge.org/external/119/11991.html