Data structures

- Organize your data to support various queries using little time and/or space
• Given n elements $A[1..n]$

• Support $\text{SEARCH}(A,x) := \text{is } x \text{ in } A$?

• Trivial solution: scan $A$. Takes time $\Theta(n)$

• Best possible given $A, x$.

• What if we are first given $A$, are allowed to preprocess it, can we then answer $\text{SEARCH}$ queries faster?

• How would you preprocess $A$?
• Given n elements A[1..n]

• Support SEARCH(A,x) := is x in A?

• Preprocess step: Sort A. Takes time O(n log n), Space O(n)

• SEARCH(A[1..n],x) := /* Binary search */
  If n = 1 then return YES if A[1] = x, and NO otherwise
  else
    if A[n/2] ≤ x then return SEARCH(A[n/2..n])
    else return SEARCH(A[1..n/2])

• Time T(n) = ?
• Given n elements A[1..n]

• Support SEARCH(A,x) := is x in A?

• Preprocess step: Sort A. Takes time $O(n \log n)$, Space $O(n)$

• SEARCH(A[1..n],x) := /* Binary search */
  If $n = 1$ then return YES if $A[1] = x$, and NO otherwise
  else
    if $A[n/2] \leq x$ then return SEARCH(A[n/2..n])
    else return SEARCH(A[1..n/2])

• Time $T(n) = O(\log n)$. 
• Given n elements A[1..n] each ≤ k, can you do faster?

• Support SEARCH(A,x) := is x in A?

• DIRECT ADDRESS:
  
  • Preprocess step: Initialize S[1..k] to 0
  For (i = 1 to n) S[A[i]] = 1
  
  • T(n) = O(n), Space O(k)

• SEARCH(A,x) = ?
Given n elements $A[1..n]$ each $\leq k$, can you do faster?

Support $\text{SEARCH}(A,x) := \text{is } x \text{ in } A$?

**DIRECT ADDRESS:**

**Preprocess step:**
Initialize $S[1..k]$ to 0
For $(i = 1 \text{ to } n)$ $S[A[i]] = 1$

$T(n) = O(n)$, Space $O(k)$

$\text{SEARCH}(A,x) = \text{return } S[x]$

$T(n) = O(1)$
• Dynamic problems:

• Want to support SEARCH, INSERT, DELETE

• Support SEARCH(A,x) := is x in A?

• If numbers are small, ≤ k
  Preprocess: Initialize S to 0.
  SEARCH(x) := return S[x]
  INSERT(x) := ...??
  DELETE(x) := ...??
• Dynamic problems:

• Want to support SEARCH, INSERT, DELETE

• Support SEARCH(A, x) := is x in A?

• If numbers are small, ≤ k
  Preprocess: Initialize S to 0.
  SEARCH(x) := return S[x]
  INSERT(x) := S[x] = 1
  DELETE(x) := S[x] = 0

• Time T(n) = O(1) per operation
• Space O(k)
• Dynamic problems:

• Want to support SEARCH, INSERT, DELETE

• Support SEARCH(A,x) := is x in A?

• What if numbers are not small?

• There exist a number of data structure that support each operation in $O(\log n)$ time
  • AVL tree, red-black tree, etc.

• These data structures organize data in a tree
Binary tree is a graph whose vertices $V$ can be divided in three disjoint sets: root, left sub-tree, and right sub-tree.

Alternatively: connected graph without cycles

- Example
  $V=\{a, b, c, d, e, f, g, h, i\}$.
  Root=$\{a\}$.
  Left subtree: $\{c\}$.
  Right subtree: $\{b, d, e, f, g, h, i\}$.
  Parent(b) = a
  Leaves = nodes with no children
  $= \{c, f, i, h, d\}$
**Binary Search Tree** is a data structure where we store data in nodes of a binary tree and refer to them as **key** of that node.

The keys in a binary search tree are always stored in such way to satisfy the **binary search tree property**:

Let \( x, y \in V \), if \( y \) is in left subtree of \( x \) \( \iff \) \( \text{key}(y) \leq \text{key}(x) \)

if \( y \) is in right subtree of \( y \) \( \iff \) \( \text{key}(x) < \text{key}(y) \).

Example:
Binary Search

Looking for $k$ in tree $T$ given root $x$:

tree-search($x$, $k$)

If $x$=NIL or $k$=$\text{key}[x]$

then return $x$

if $k$<$\text{key}[x]$

then return tree-search(left[$x$], $k$)

else return tree-search(right[$x$], $k$)

Note: NIL stands for empty tree

Running time = the depth of the tree. = $O(n)$, = $\Omega(\log n)$

Tree is balanced if depth $\leq 1+\log n \Rightarrow$ search time $O(\log n)$
Binary Search in a tree is a generalization of binary search in an array that we saw before.

A sorted array can be thought of as a balanced tree (we'll return to this shortly)

Trees make it easier to think about inserting and removing.
Insert x in a tree:
Search (x).
If x not found, create new node with x where x should have been.
To maintain the tree balanced, we perform rotations

Delete x from a tree:
Search (x).
Remove the node with x.
To maintain the tree balanced, we perform rotations

Time $O(\log n)$ for both.
DEMO
Problem: Dynamically support $n$ search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

<table>
<thead>
<tr>
<th>Function</th>
<th>Search time</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = x$</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>$t = 2^n$, open addressing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem: Dynamically support $n$ search/insert elements in $\{0,1\}^u$
Idea: Use function $f : \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

<table>
<thead>
<tr>
<th>Function</th>
<th>Search time</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = x$</td>
<td>O(1)</td>
<td>$2^u$</td>
</tr>
<tr>
<td>$t = 2^n$, open addressing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any deterministic function</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>


Problem: Dynamically support $n$ search/insert elements in $\{0,1\}^u$

Idea: Use function $f : \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

<table>
<thead>
<tr>
<th>Function</th>
<th>Search time</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = x$</td>
<td>$O(1)$</td>
<td>$2^u$</td>
</tr>
<tr>
<td>$t = 2^n$, open addressing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any deterministic function</td>
<td>$n$</td>
<td>0</td>
</tr>
<tr>
<td>Random function</td>
<td>? expected</td>
<td>?</td>
</tr>
</tbody>
</table>
Problem: Dynamically support $n$ search/insert elements in $\{0,1\}^u$

Idea: Use function $f: \{0,1\}^u \rightarrow [t]$, resolve collisions by chaining

<table>
<thead>
<tr>
<th>Function</th>
<th>Search time</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = x$</td>
<td>$O(1)$</td>
<td>$2^u$</td>
</tr>
<tr>
<td>$t = 2^n$, open addressing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any deterministic function</td>
<td>$n$</td>
<td>$0$</td>
</tr>
<tr>
<td>Random function</td>
<td>$n/t$ expected</td>
<td>$2^u \log(t)$</td>
</tr>
<tr>
<td></td>
<td>$\forall x \neq y$, $\Pr[f(x)=f(y)] \leq 1/t$</td>
<td></td>
</tr>
</tbody>
</table>

Now what?
We "derandomize" random functions
Problem: Dynamically support \( n \) search/insert elements in \( \{0,1\}^u \)

Idea: Use function \( f : \{0,1\}^u \rightarrow [t] \), resolve collisions by chaining

<table>
<thead>
<tr>
<th>Function</th>
<th>Search time</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) = x )</td>
<td>( O(1) )</td>
<td>( 2^u )</td>
</tr>
<tr>
<td>( t = 2^n ), open addressing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any deterministic function</td>
<td>( n )</td>
<td>0</td>
</tr>
<tr>
<td>Random function</td>
<td>( n/t ) expected</td>
<td>( 2^u \log(t) )</td>
</tr>
<tr>
<td></td>
<td>( \forall x \neq y, \Pr[f(x)=f(y)] \leq 1/t )</td>
<td></td>
</tr>
<tr>
<td>Pseudorandom function</td>
<td>( n/t ) expected</td>
<td>( O(u) )</td>
</tr>
<tr>
<td>A.k.a. hash function</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Idea: Just need ( \forall x \neq y, \Pr[f(x)=f(y)] \leq 1/t )</td>
<td></td>
</tr>
</tbody>
</table>
Construction of hash function:
Let $t$ be prime. Write $u$-bit elements in base $t$.
$x = x_1 \ x_2 \ ... \ x_m$ for $m = \frac{u}{\log(t)}$

Hash function specified by an element $a = a_1 \ a_2 \ ... \ a_m$

$$f_a (x) := \sum_{i \leq m} a_i \ x_i \ \text{modulo}$$

Claim: $\forall x \neq x', \Pr_a [f_a (x) = f_a (x')] = 1/t$
Different constructions of hash function:

u-bit keys to r-bit hashes

Classic solution: pick a prime \( p > 2^u \), and a random \( a \) in \([p]\), and

\[
h_a(x) := ((ax) \text{ mod } p) \text{ mod } 2^r
\]

Problem: \text{mod } p \text{ is slow, even with Mersenne primes (} p = 2^i - 1) \text{)

Alternative: let \( b \) be a random odd u-bit number and

\[
h_b(x) = (((bx) \text{ mod } 2^u) \text{ div } 2^{u-r}
\]

= bits from u-r to u of integer product bx

Faster in practice. In C, think \( x \) unsigned integer of \( u = 32 \) bits

\[
h_b(x) = (b \times x) >> (u-r)
\]
Static search:

Given $n$ elements, want a hash function that gives no collisions.

Probabilistic method: Just hash to $[t] = n^2$ elements

$$\Pr[ \exists x \neq y : \text{hash}(x) = \text{hash}(y) ]$$

$$\leq \frac{n^2}{2} \Pr[\text{hash}(0) = \text{hash}(1)]$$

$$\leq \frac{n^2}{(2t)} = \frac{1}{2}$$

$$\Rightarrow \exists \text{hash} : \forall x \neq y, \text{hash}(x) \neq \text{hash}(y) \text{ (probabilistic method)}$$

Can you have no collisions with $[t] = O(n)$?
Static search:

Given n elements, want a hash function that gives no collisions.

Two-level hashing:
- First hash to \( t = O(n) \) elements,
- then hash again using the previous method. That is, if i-th cell in first level has \( c_i \) elements, hash to \( c_i^2 \) cells at the second level.

Expected total size \( \leq \mathbb{E}[\sum_{i \leq t} c_i^2] \)

Note \( \sum_{i \leq t} c_i^2 = \Theta(\text{expected number of colliding pairs in first level}) = O(???) \)
Static search:

Given $n$ elements, want a hash function that gives no collisions.

Two-level hashing:
- First hash to $t = O(n)$ elements,
- then hash again using the previous method. That is, if $i$-th cell in first level has $c_i$ elements, hash to $c_i^2$ cells at the second level.

Expected total size $\leq E[ \sum_{i \leq t} c_i^2 ]$

Note $\sum_{i \leq t} c_i^2 = \Theta(\text{expected number of colliding pairs in first level}) = O(n^2 / t ) = O(n)$
More about arrays, and heaps.
Stack

Operations: Push, Pop
Last-in-first-out

Queue

Operations: Enqueue, Dequeue
First-in-first-out

Simple implementation using arrays.
Each operation supported in O(1) time.
A binary tree is **complete** if all the nodes have two children except the nodes in the last level.

A complete binary tree of depth $d$ has $2^d$ leaves and $2^{d+1}-1$ nodes.

Example:
- Depth of $T$=?
- Number of leaves in $T$=?
- Number of nodes in $T$=?
A binary tree is **complete** if all the nodes have two children except the nodes in the last level.

A complete binary tree of depth $d$ has $2^d$ leaves and $2^{d+1}-1$ nodes.

Example:
Depth of $T=3$.
Number of leaves in $T=8$.
Number of nodes in $T=15$. 
A binary tree is complete if all the nodes have two children except the nodes in the last level.

A complete binary tree of depth $d$ has $2^d$ leaves and $2^{d+1} - 1$ nodes.

Example:
Depth of $T = 3$.
Number of leaves in $T = 2^3 = 8$.
Number of nodes in $T$ = ?
A binary tree is **complete** if all the nodes have two children except the nodes in the last level.

A complete binary tree of depth $d$ has $2^d$ leaves and $2^{d+1} - 1$ nodes.

**Example:**

Depth of $T = 3$.

Number of leaves in $T = 2^3 = 8$.

Number of nodes in $T = 2^{3+1} - 1 = 15$. 
Heap is like a complete binary tree except that the last level may be missing nodes, and if so is filled from left to right.

Note: A complete binary tree is a special case of a heap.

A heap is conveniently represented using arrays
Navigating a heap:

Root is A[1].

Given index i to a node:

- Parent(i) = i/2
- Left-Child(i) = 2i
- Right-Child(i) = 2i+1
Heaps are useful to dynamically maintain a set of elements while allowing for extraction of minimum.

The same results hold for extraction of maximum.

We focus on minimum for concreteness.
Min-heap

The values stored in the nodes satisfy: \( A[\text{Parent}(i)] \leq A[i] \).
Extracting the minimum element
In min-heap \( A \), the minimum element is \( A[1] \).

Extract-Min-heap(A)

\[
\begin{align*}
\text{min} &:= A[1]; \\
A[1] &:= A[\text{heap-size}]; \\
\text{heap-size} &:= \text{heap-size} - 1; \\
\text{Min-heapify}(A, 1) \\
\text{Return } \text{min};
\end{align*}
\]

Let's see the steps
Extracting the minimum element
In min-heap $A$, the minimum element is $A[1]$.

Extract-Min-heap($A$)

$$\text{min} := A[1];$$
$$\text{heap-size} := \text{heap-size} - 1;$$
$$\text{Min-heapify}(A, 1)$$

Return $\text{min};$
Extracting the minimum element
In min-heap $A$, the minimum element is $A[1]$.

Extract-Min-heap($A$)

heap-size := heap-size – 1;
Min-heapify($A$, 1)
Return min;
Extracting the minimum element
In min-heap $A$, the minimum element is $A[1]$.

Extract-Min-heap($A$)

$\text{min} := A[1]$;
$\text{heap-size} := \text{heap-size} - 1$;
$\text{Min-heapify}(A, 1)$
Return $\text{min}$;

Min-heapify is a function that restores the min property.
Min-heapify restores the min-heap property
given array \( A \) and index \( i \) such that trees rooted at \( \text{left}[i] \) and \( \text{right}[i] \) are min-heap, but \( A[i] \) maybe greater than its children

**Min-heapify**(\( A, i \))

Let \( j \) be the index of smallest node among \( \{A[i], A[\text{Left}[i]], A[\text{Right}[i]]\} \)

If \( j \neq i \) then {

exchange \( A[i] \) and \( A[j] \)

Min-heapify(\( A, j \))

}
Min-heapify restores the min-heap property given array $A$ and index $i$ such that trees rooted at $\text{left}[i]$ and $\text{right}[i]$ are min-heap, but $A[i]$ may be greater than its children.

**Min-heapify($A$, $i$)**

Let $j$ be the index of smallest node among \{${A[i], A[\text{Left}[i]], A[\text{Right}[i]]}$\}

If $j \neq i$ then {
  exchange $A[i]$ and $A[j]$
  Min-heapify($A$, $j$)
}
Min-heapify(A, i)
Let j be the index of smallest node among \{A[i], A[Left[i]], A[Right[i]]\}
If j ≠ i then {
exchange A[i] and A[j]
Min-heapify(A, j)
}
Min-heapify restores the min-heap property given array \( A \) and index \( i \) such that trees rooted at \( \text{left}[i] \) and \( \text{right}[i] \) are min-heap, but \( A[i] \) maybe greater than its children

\[
\text{Min-heapify}(A, i)\\
\quad \text{Let } j \text{ be the index of smallest node among } \{A[i], A[\text{Left}[i]], A[\text{Right}[i]]\}\\
\quad \text{If } j \neq i \text{ then } \{\\
\quad\quad \text{exchange } A[i] \text{ and } A[j]\\
\quad\quad \text{Min-heapify}(A, j)\\
\quad\}\}
\]
Min-heapify restores the min-heap property given array \( A \) and index \( i \) such that trees rooted at \( \text{left}[i] \) and \( \text{right}[i] \) are min-heap, but \( A[i] \) maybe greater than its children.

\[
\text{Min-heapify}(A, i) \\
\quad \text{Let } j \text{ be the index of smallest node among } \{A[i], A[\text{Left}[i]], A[\text{Right}[i]]\} \\
\quad \text{If } j \neq i \text{ then } \\
\qquad \text{exchange } A[i] \text{ and } A[j] \\
\qquad \text{Min-heapify}(A, j) 
\]

Running time = ?
Min-heapify restores the min-heap property given array \( A \) and index \( i \) such that trees rooted at \( \text{left}[i] \) and \( \text{right}[i] \) are min-heap, but \( A[i] \) maybe greater than its children.

**Min-heapify** \((A, i)\)

Let \( j \) be the index of smallest node among \( \{A[i], A[\text{Left}[i]], A[\text{Right}[i]]\} \)

If \( j \neq i \) then {
exchange \( A[i] \) and \( A[j] \)
Min-heapify \((A, j)\)
}

Running time = depth = \( O(\log n) \)
Recall Extract-Min-heap(A)

\[
\begin{align*}
\text{min} &: = A[1]; \\
A[1] &: = A[\text{heap-size}] ; \\
\text{heap-size} &: = \text{heap-size} – 1; \\
\text{Min-heapify}(A, 1) \\
\text{Return } \text{min};
\end{align*}
\]

Hence both Min-heapify and Extract-Min-Heap take time $O(\log n)$.

Next: How do you insert into a heap?
Insert-Min-heap \((A, key)\)

heap-size\[A\] := heap-size\[A\]+1;
A[heap-size] := key;

for \(i:=\) heap-size\[A\]; \(i>1\) and A[parent\(i\)] > A[i]; \(i:= parent[i]\) do
 exchange(A[parent\(i\)], A[i])

Running time = ?
Insert-Min-heap \((A, \text{key})\)

\[
\text{heap-size}[A] := \text{heap-size}[A]+1; \\
A[\text{heap-size}] := \text{key}; \\
\]

for \((i := \text{heap-size}[a]; i > 1 \land A[\text{parent}(i)] > A[i]; i := \text{parent}[i])\)

\[
\text{exchange}(A[\text{parent}(i)], A[i])
\]

Running time \(= O(\log n)\).

Suppose we start with an empty heap and insert \(n\) elements. By above, running time is \(O(n \log n)\).

But actually we can achieve \(O(n)\).
Build Min-heap
Input: Array A, output: Min-heap A.

For (i := length[A]/2; i ≥ 0; i --)
    Min-heapify(A, i)

Running time = ?

Min-heapify takes time O(h) where h is depth.

How many trees of a given depth h do you have?
Build Min-heap
Input: Array A, output: Min-heap A.

For (i := length[A]/2; i < 0; i --)
    Min-heapify(A, i)

Running time = \(O(\sum_{h < \log n} n/2^h)\) h

= n \(O(\sum_{h < \log n} h/2^h)\)

= ?
Build Min-heap
Input: Array A, output: Min-heap A.

For (i := length[A]/2; i < 0; i --)
    Min-heapify(A, i)

Running time = \( O(\sum_{h < \log n} n/2^h \cdot h) \)

= \( n \cdot O(\sum_{h < \log n} h/2^h) \)

= \( O(n) \)
Compact (also known as succinct) arrays
Bits vs. trits

- Store \( n \) “trits” \( t_1, t_2, \ldots, t_n \in \{0,1,2\} \)

- In \( u \) bits \( b_1, b_2, \ldots, b_u \in \{0,1\} \)

- Want:
  - Small space \( u \) (optimal = \( \lceil n \log_2 3 \rceil \))
  - Fast retrieval: Get \( t_i \) by probing few bits (optimal = 2)
Two solutions

• Arithmetic coding:
  Store bits of \((t_1, \ldots, t_n) \in \{0, 1, \ldots, 3^n - 1\}\)

  Optimal space: \(\lceil n \log_2 3 \rceil \approx n \cdot 1.584\)
  Bad retrieval: To get \(t_i\) probe all \(> n\) bits

• Two bits per trit

  Bad space: \(n \cdot 2\)
  Optimal retrieval: Probe 2 bits
Polynomial tradeoff

- Divide n trits $t_1, \ldots, t_n \in \{0, 1, 2\}$ in blocks of $q$.
- Arithmetic-code each block.

Space: \[
\left\lceil q \log_2 3 \right\rceil \frac{n}{q} < (q \log_2 3 + 1) \frac{n}{q} = n \log_2 3 + \frac{n}{q}
\]

Retrieval: Probe $O(q)$ bits