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

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

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

• 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] ≤ 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 \( \leq k \), 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 \( (i = 1 \text{ to } n) \) \( S[A[i]] = 1 \)

\( T(n) = O(n), \text{ Space } O(k) \)

\( \text{SEARCH}(A,x) = ? \)
• 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 $(i = 1$ 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

• Trees: AVL, 2-3, 2-3-4, B-trees, red-black, ...

• Skip lists, deterministic skip lists,

• Let's see binary search trees first
Binary tree

Vertices, aka nodes = \{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 satisfy the binary search tree property:

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

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

Example:
Binary Search

Looking for \( k \) in tree \( T \) given root \( x \):

\[
\text{tree-search}(x, k)
\]

- If \( x = \text{NIL} \) or \( k = \text{key}[x] \)
  - then return \( x \)
- if \( k < \text{key}[x] \)
  - then return \( \text{tree-search}(\text{left}[x], k) \)
  - else return \( \text{tree-search}(\text{right}[x], k) \)

Note: NIL stands for empty tree

Running time = the depth of the tree \( \in [\log n, n] \)

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)

Trees make it easier to think about inserting and removing

Want to support search, insert, and delete in time $O(\log n)$

When inserting and deleting, must restructure the tree to keep it balanced or almost balanced.
• AVL: binary trees. In any node, height of children differ by \( \leq 1 \). Maintain by rotations

• 2-3-4 trees: nodes have 1, 2, or 3 keys and 2, 3, or 4 children. All leaves same level. To insert in a leaf: add a child. If already 4 children, split the node into one with 2 children and one with 4, add a child to the parent recursively. When splitting the root, need to create new root.

  Deletion is more complicated.

• B-trees are a generalization of 2-3-4 trees where can have more children. Useful in some disk applications where loading a node corresponds to reading a chunk from a disk

• Red-black trees: A way to “simulate” 2-3-4 trees by a binary tree. E.g. split 2 keys in same 2-3-4 node into 2 red-black nodes. Color edges red or black depending on whether the child comes from this splitting or not, i.e., is a child in the 2-3-4 tree or not.
We see what may be the simplest variant of these trees.

Each node has a level, which is 1 for leaves.

Convention: Have “sentinels” at level 0.

Every node, even leaves, has two children.

Rule: In the tree, the only allowed path with nodes of the same level is a left-right edge. The end point always has level < starting point + 1. This implies that height is $O(\log n)$.

Rule of thumb for restructuring:
First make sure that only left-right edges are within nodes of the same level, then worry about length of paths within same level.
We will use rotations to re-balance:

Right rotation at node x with key 6:

```

   6                                    4
   4             7        =>         3             6
  3     5                                           5       7
```

Temp = x; x = x.left; temp.left = x.right; x.right = temp;

Left notation is symmetric.
Restructuring operations:

Skew(x): If x has left-child with same level, perform right rotation.

Split(x): If x.right.right has same level as x, perform left rotation.

    Increase the level of x.

Decrease Level(x): If one of x's children is two levels below x, decrease the level of x by one. If x.right has the same level of x, decrease the level of x.right by one too.
Insert(x): { 
  Search(x). Insert x as a new leaf where it should have been.
  Follow the path from x to the root and at each node y do:
    Skew(y).
    Split(y).
}

Delete(x): Suppose x is a leaf
 Delete x.
 Follow the path from x to the root and at each node y do:
  Decrease level(y).
  Skew(y); Skew(y.right); Skew(y.right.right);
  Split(y); Split(y.right); 
}
Fig. 1. Example of insertion into a BB-tree. The levels are separated by horizontal lines.
Fig. 2. Example of deletion.
Delete(x):

If $x$ is not a leaf, find the smallest leaf bigger than $x$.key, swap it with $x$, and remove that leaf.

To find that leaf, just perform search, and when you hit $x$ go, e.g., right. It's the same thing as searching for $x$.key + $\epsilon$

So swapping won't destroy the search properties
Remark about memory implementation:

You can use new/malloc free/dispose to add and remove nodes.

However, this may cause memory segmentation.

It is possible to implement any tree using an array $A$ in such a way that at any point in time if $n$ elements are in the tree, those will take elements $A[1..n]$ in the array only.

To do this, when you remove node with index $i$ in the array, swap $A[i]$ and $A[n]$. Use parent's pointers to update the pointers.
Running time: $O(\log n)$ for Search, insert, and delete.

Space: $O(n)$. For each key we need to store level and pointers to children, and possibly pointer to parent too.

Can we store only the keys and achieve space $n + o(n)$?

Surprisingly, this is possible:

Optimal Worst-Case Operations for Implicit Cache-Oblivious Search Trees
by Franceschini and Grossi

Project: Explain to me how that works.
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>
</tbody>
</table>

Any deterministic function

Search time  

Extra space  

?             

?             

?
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>t = 2^n ), open addressing</td>
<td>( \Omega(1) )</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 $\forall x \neq y, \Pr[f(x)=f(y)] \leq 1/t$</td>
<td>$2^u \log(t)$</td>
</tr>
<tr>
<td>Pseudorandom function A.k.a. hash function</td>
<td>$n/t$ expected $\forall x \neq y, \Pr[f(x)=f(y)] \leq 1/t$</td>
<td>$O(u)$</td>
</tr>
</tbody>
</table>

Idea: Just need $\forall x \neq y, \Pr[f(x)=f(y)] \leq 1/t$
Construction of hash function:
Let \( t \) be prime. Write \( u \)-bit elements in base \( t \).
\( x = x_1 x_2 \ldots x_m \) for \( m = \frac{u}{\log(t)} \)

Hash function specified by an element \( a = a_1 a_2 \ldots a_m \)

\[ f_a(x) := \sum_{i \leq m} a_i x_i \pmod{t} \]

Claim: \( \forall x \neq x', \Pr_a [f_a(x) = f_a(x')] = \frac{1}{t} \)
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

$$h_a(x) := ((ax) \mod p) \mod 2^r$$

Problem: mod $p$ is slow, even with Mersenne primes ($p = 2^i - 1$)

Alternative: let $b$ be a random odd u-bit number and

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

= 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

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

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

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

$$\Rightarrow \exists \text{hash} : \forall x \neq y, \text{hash}(x) \neq \text{hash}(y)$$  (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 \( \leq E[ \sum_{i \leq t} c_i^2 ] \)

Note \( \sum_{i \leq t} c_i^2 = \Theta(\text{expected number of colliding pairs in first level}) = O(\ldots) \)
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 $T=7$?
Number of nodes in $T=13$?
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\((i)\) = \( \frac{i}{2} \)
- 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)

min:= A[1];
A[1]:= A[heap-size];
heap-size:= heap-size – 1;
Min-heapify(A, 1)
Return min;

Let's see the steps
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($A$)

\[
\begin{align*}
\text{min} & := A[1]; \\
\text{heap-size} & := \text{heap-size} - 1; \\
\text{Min-heapify}(A, 1) \\
\text{Return } \text{min};
\end{align*}
\]
Extracting the minimum element
In min-heap A , the minimum element is A[1].

Extract-Min-heap(A)

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

Min-heapify is a function that restores the min property
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

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 $A[i]$ 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 $A[i]$ 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(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)
}

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)

min:= A[1];
A[1]:= A[heap-size];
heap-size:= heap-size –  1;
Min-heapify(A, 1)
Return min;

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)

A[heap-size] := key;

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

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

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

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

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 = \( O\left( \sum_{h < \log n} \frac{n}{2^h} \right) \cdot h \)
= \( n \cdot O\left( \sum_{h < \log n} \frac{h}{2^h} \right) \)
= ?
Build Min-heap
Input: Array A, output: Min-heap A.

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

Running time = \( O(\sum_{h < \log n} \frac{n}{2^h}) \) h
              = \( n \cdot O(\sum_{h < \log n} \frac{h}{2^h}) \)
              = \( O(n) \)
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 = \( \left\lceil n \frac{\log_2 3}{2} \right\rceil \))
  - 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: \(\left\lceil n \log_2 3 \right\rceil \approx n \cdot 1.584\)

  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:** \( \lceil q \log_2 3 \rceil n/q < (q \log_2 3 + 1) n/q \)

\[ = n \log_2 3 + n/q \]

**Retrieval:** Probe \( O(q) \) bits