Basic building blocks -
Lists + Trees

Examples of data structures
way to represent information
so it can be manipulated
packaged along with routines to manipulate
Abstract Data Type (ADT)
API (Application Program Interface)
hides the internal details

Tool:

use specification (API)
implementation

Linked List of Data operations (API)
create empty list
delete list

O(1) insert data at front of list
O(1) delete data at front
O(n) insert data at end
O(n) delete data at end
O(n) count length
O(n) search for particular data (key)
O(n log n) sort data

"big-Oh of ..."
"order of ..."
Linked List of Integer Nodes:

- handle = header

```
   data  next
   3    11    4    null
```

- empty list
- insert at front: 5

1) build a node A, with value inside
2) A.next = handle.next
3) handle.next = A

Efficiency (optimization):
1) if fast enough, leave it alone.
2) Can you wait a year?
3) Is your algorithm right?
4) Determine where the time is spent.

gcc -O3
Queues, stacks, deques

Queue: 0, 2, 3, 4
Stack: 1, 2
Push pop
Deque: 0, 2, 3, 4

Complexity of an algorithm = O( )
Stack of integers

Operations:
- makeEmptyStack()
- isEmptyStack(s)
- popStack(s)
- pushStack(s, x, int data)

Implementation 1: Linked list of integers
- makeEmptyStack:
- makeEmptyList:
- isEmptyStack:
- deleteFromList:
- popStack:
- insertFrontList:

Implementation 2: Array of integers

```
[3 | 1 | 4 | ]
```

push(3)
push(1)
push(4)

push: array[count] <- data
count += 1

pop: count -= 1
return array[count]
Queue of Integer

\[ q \leftarrow \text{make Empty Queue (} \right) \]
\[ \text{bool} \leftarrow \text{is Empty Queue (} q \right) \]
\[ \text{void insert Queue (} *q, \text{ int data}) \]
\[ \text{int} \leftarrow \text{delete Queue (} *q \right) \]

Implementation 1: Linked List

```
front
\[
\begin{array}{c}
\text{front}\n\hline
\text{rear}\n\hline
\text{dummy node}
\end{array}
\]
```

90 randGen.p | cards
90 cards

```
#include <stdio.h>

std::out
stdin
stderr

\fscanf(stdin, "%d", &nextInt);

main(int argc, char *argv[])
```
empty

insert 3

insert 1

delete

delete
Implementation 2 of queues

Array

[3 1 4 1 5 9 2 3]

Deque of integer,

Implementation 2: array

Similar to Queue

Need "retreat()" function

Implementation 1: doubly-linked list
Searching / Sorting

API:

- n data elements (int)
- data structures D
- insert (int data, D)
- bool search (int data, D)

Representation 1: Linked list

- insert: \( \Theta(1) \) (at \( \Theta(1) \))
- search: \( \Theta(n) \)

Representation 2: Sorted linked list

- insert: \( \Theta(n) \)
- search: \( \Theta(n) \)

Representation 3: Array

- insert: \( \Theta(1) \)
- search: \( \Theta(n) \)
Rep 4: Sorted array

\[
\begin{array}{cccccc}
& & & & & \text{max} \\
\hline
& & & & & \\
\hline
0 & & & & & \\
\hline
\text{counter}
\end{array}
\]

\[
\begin{array}{cccccc}
\text{insert: } O(n) \\
\text{search: } O(\log n) \\
\text{Binary Search}
\end{array}
\]

Insert at end,

\[
\begin{array}{cccccc}
32 & 49 & 56 & 83 & 92 & 70 \\
\text{70} & 83 & 92
\end{array}
\]

Insert at start

Look from start

\[
\begin{array}{cccccc}
\text{32} & \text{49} & \text{56} & \text{83} & \text{92} & \text{70} & \text{83} & \text{92}
\end{array}
\]

\[
\begin{array}{cccccc}
\text{70} & \text{83} & \text{92}
\end{array}
\]
Recursion theorem

Given \( C_n = f(n) + a \cdot C_{n/b} \)

where \( f(n) = \Theta(n^k) \),

Then

<table>
<thead>
<tr>
<th>when</th>
<th>( C_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a &lt; b^k )</td>
<td>( \Theta(n^k) )</td>
</tr>
<tr>
<td>( a = b^k )</td>
<td>( \Theta(n^k \log n) )</td>
</tr>
<tr>
<td>( a &gt; b^k )</td>
<td>( \Theta(n^{\log_b a}) )</td>
</tr>
</tbody>
</table>

Binary search:

\[
C_n = 1 + C_{n/2}
\]

\( f(n) = 1 \quad k = 0 \)

\( a = 1 \)

\( b = 2 \)

\( \Rightarrow C_n = \Theta(n^k / \log n) = \Theta(\log n) \)
\( \Theta (f(n)) = \text{no worse than } f(n) \)
\[ = \text{at most } f(n) \]
\[ \Omega (f(n)) = \text{no better than } f(n) \]
\[ = \text{at least } f(n) \]
\[ \Theta (f(n)) = \text{exactly } f(n) \]

Data structure: sorted binary tree

balanced: all paths have about the same length
path: from root down to a leaf
exact balance:
#nodes: \(2^n - 1\)

balanced:
path lengths are with 1 of each other

"vine"
worst tree

<table>
<thead>
<tr>
<th>num</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>(2^n - 1)</td>
<td>(n)</td>
</tr>
<tr>
<td>(x)</td>
<td>(\sim \log_2 x)</td>
</tr>
</tbody>
</table>

insertion into a balanced tree of \(n\) nodes is \(O(\log n)\)

searching in a balanced tree is \(O(\log n)\)

building a vine: \(O(n^2)\)
building a balanced tree: \(O(n \log n)\)
Trees: operations

2. Build
   1. Insert into
      offline / online
         \[ \Rightarrow \text{one value at a time} \]
         \[ \Rightarrow \text{have all values in advance} \]
         \[ \Rightarrow \text{build a balanced tree} \]
         \[ \text{in } \mathcal{O}(n \log n) \]

3. Search (exact, close)

4. Traversals
   \[
   \text{void symmetric (in-order)}
   \]
   \[
   \text{if tree node is empty then return else}
   \]
   \[
   \text{node->left}
   \]
   \[
   \text{symmetric (left child)}
   \]
   \[
   \text{visit (node)}
   \]
   \[
   \text{symmetric (node->right)}
   \]
   \[\]
```c
void preorder (node)  
{  
    if node is empty  
    {  
    } else  
    {  
        visit (node);  
        preorder (node -> left);  
        preorder (node -> right);  
        // not empty  
    }  
    // preorder()  
}  
void postorder (node)  
{  
    if node is empty  
    {  
    } else  
    {  
        postorder (node -> left);  
        postorder (node -> right);  
        visit (node);  
        // not empty  
    }  
    // postorder()  
}
```

Representation 6: Hashing (scatter storage) later.
insert: O(1)
search: O(1)
Finding the jth element in a set of numbers.

1. Easy situation: numbers are sorted.
   - want highest: \( a[100] \) \( \Theta(1) \)

2. want 3rd highest: \( a[98] \) \( \Theta(1) \)

3. want median: \( a[50] \) \( \Theta(1) \)

4. numbers not sorted.
   - want highest: \( \Theta(n) \)

\[
\text{largest} = -\infty
\]

\[
\text{for each value in array } \exists
\]

\[
\text{if (value > largest)} \exists
\]

\[
\text{largest} = \text{value};
\]

\[
\text{else if (value > nextLargest) } \exists
\]

\[
\text{nextLargest} = \text{value};
\]

\[
\text{else } \exists
\]

\[
\text{nextLargest} = \text{value};
\]
(6) not sorted, want j'th largest.
   \( \Theta(jn) \): keep an array of the j largest so far.

(7) median (unsorted) \( \Theta(n^2) \)
better: \( \Theta(n) \)
based on partitioning (half-hearted sorting) on a pivot.

\[\begin{array}{c}
\text{arbitrary} \\
\uparrow \\
\text{small} \quad P_1 \quad \text{large} \\
\uparrow \quad \uparrow \\
x \quad x \\
\text{after 1 partitioning} \\
\uparrow \\
\text{after 2} \\
\text{...} \\
\text{...} \\
\uparrow \\
P \\
\end{array}\]

Complexity:
recurrence relation
\[C_n = n + C_{n/2}\]
\[a = 1, \quad b^k = 2^k = 2\]
\[b = 2, \quad \Theta(n^k) = \Theta(n)\]
\( O(1) \) constant
\( O(\log n) \) logarithmic
\( O(n) \) linear
\( O(n^2) \) quadratic
\( O(n^3) \) cubic
\( O(2^n) \) exponential

Nico Iomuto’s algorithm

"comb" method.

\[
\begin{align*}
l & \leftarrow \text{\texttt{index}} \\
p & \leftarrow \text{\texttt{pivot index}} \\
\text{\texttt{pivotValue} = array[0]} \\
p & \leftarrow 0 \\
\text{for \( i \)} \\
\text{\texttt{small} \rightarrow \text{\texttt{big}}} \\
\end{align*}
\]
Sorting methods

Insertion sort:

\[ \Theta(n^2) \]

Before

\[
\begin{array}{cccc}
A & B & C & D \\
3 & 2 & 1 & 7
\end{array}
\]

After

\[
\begin{array}{cccc}
A & B & C & D \\
0 & 1 & 2 & 7
\end{array}
\]
Selection sort

\[ \text{sorted} \quad \text{unsorted} \]

\[
\begin{array}{c}
|111\|1111| \\
\text{small} \quad \text{large} \\
\text{in place} \quad \text{smallest}
\end{array}
\]

before

\( \Theta(n^2) \)

QuickSort (C. A. R. Hoare)

\[
\begin{array}{c}
\text{initial} \\
\text{partition} \\
\text{after 1 step} \\
\text{recurse on both sides}
\end{array}
\]

Tony
Initially: \(\Theta(n)\)

Log \(n\) steps, each costing \(\Theta(n)\) for partitioning

\[\Rightarrow \Theta(n \log n)\]

Suggestions for improvement:
1) choose median of 3 or 5 as pivot
2) don't recurse if size is <= 10 after all recursion is done, run 1 pass of insertion sort.

Recursion theorem

\[C_n = f(n) + a \cdot C_{n/b} + \begin{cases} k = 1 \\ a = 2 \\ b = 2 \end{cases} \]

\[\Theta(n^k \log n) = \Theta(n/\log n)\]
Consider a worse partitioning

\[ C_n = n^k + 2 C_{\frac{n}{3^k}} \]

\[ k = 1 \]
\[ a = 2 \]
\[ b = \frac{3}{2} \]

\[ \Theta \left( n^{\log_2 2} \right) \]
\[ \Theta \left( n^{\log_3 2} \right) = \Theta (n^{1.7}) \]

Shell Sort (Donald Shell 1959)

Each pass uses a span \( s \) value.

For each offset \( 0 \ldots s-1 \)
Next pass: smaller span.

\[ \begin{array}{c}
\text{offset 0} \\
\text{offset 1}
\end{array} \]

\[ \begin{array}{c}
\text{last pass: span = 1}
\end{array} \]

**Heaps**

- \[ \begin{array}{c}
\text{2}
\end{array} \]
- \[ \begin{array}{c}
3
\end{array} \]
- \[ \begin{array}{c}
42
\end{array} \]
- \[ \begin{array}{c}
92
\end{array} \]
- \[ \begin{array}{c}
90
\end{array} \]
- \[ \begin{array}{c}
43
\end{array} \]

Sift up: swap with a parent, may be repeated insertion: \( \log n \)
delete from heap always deletes smallest element.

sift down: swap with the better child, maybe repeatedly.

parent is at [index/2]
right child is at index * 2 + 1
left child is at index * 2
Uses of a heap.

1) Priority queue
2) Sorting heap Sort

- Put numbers in an array.
- Make the array a heap.
  - Top-heavy: largest value at root

\[ \text{heap} \quad \text{random} \]

\[ \Theta(n \log n) \]

- Repeatedly delete from heap.

\[ \text{heap} \quad \text{sorted} \]

\[ \Theta(n \log n) \]

Fast heapification

\[ \text{heap} \]

\[ \Theta(n) \]