Fib(n) =
\begin{align*}
&5.0 \quad \text{if } n = 0 \\
&1 \quad \text{if } n = 1 \\
&Fib(n-1) + Fib(n-2) \quad \text{if } n > 1
\end{align*}

0, 1, 2, 3, 5, 8, 13, ... the ratio goes to \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.62 \)

\[Fib(n) = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}} \approx \frac{\phi^n}{\sqrt{5}}\]

\[= O(\phi^n)\]

Use some algorithm to compute Fib(n): Recursive Algorithm

\[\text{Fib}(4)\]
\[\text{Fib}(3) \quad \text{Fib}(2)\]
\[\text{Fib}(2) \quad \text{Fib}(1) \quad \text{Fib}(0)\]
\[\text{Fib}(1) \quad \text{Fib}(0) \quad \text{Fib}(0)\]
\[\text{Fib}(1) \quad \text{Fib}(0) \quad \text{Fib}(0)\]

Time:
\[T(n) = \begin{cases} & T(n-1) + T(n-2) + O(1) \\
& O(1) \quad \text{if } n < 2 \end{cases}\]

\[
\text{leaves} = \# \text{internal node} + 1 \Rightarrow \text{# nodes} \geq Fib(n) \\
\leq 3 Fib(n-1)
\]

\[T(n) = O(Fib(n)) = O(\phi^n)\]
We usually analyse the size of the output when we analyse time, but here we analyse the magnitude.

Size = \# bits needed to represent input

Space =

$$\mathcal{O}(n)$$

Another:

```python
fib_{-2} = 0
fib_{-1} = 0
for i = 2, i < n, i++ do
  fib_i = fib_{i-1} + fib_{i-2}
  fib_{i-1} = fib_{i-2}
  fib_{i-2} = fib_{i-1}
end for
return fib_{n-1}
```

$$T(n) = \mathcal{O}(n)$$

$$\mathcal{O}(1)$$

Problem: You keep doing the same thing over and over again.

⇒ Memorizing ⇒ Some solutions to any subproblems that you solve
for (i = 0; i < MAX_N; i++) {
    mem0[i] = -1;
}
mem0[0] = 0;
mem0[1] = 1;

int fib(int n) {
    if (n < 0) exit(0);
    if (mem0[n] > -1) {
        return mem0[n];
    } else {
        mem0[n] = fib(n-1) + fib(n-2);
        return mem0[n];
    }
}

<MEMORIZATION>

T(n) = O(n)
S(n) = O(1)

Dynamic Programming
- solve all subproblems
- use explicit schedule
- "Dynamic Optimality": solution to any problem can be composed of solutions to subproblems

Divide and Conquer
1. Divide the problem into smaller or easier subproblems
2. Solve problems
3. Use solutions to construct solution to original problem
\[ DP = \]

\[
\begin{aligned}
&\text{for } i = 0 \text{ to } \text{MAX-N, } i \in \mathbb{N} \}
\text{fib}(i) = i; \\
&\text{fib}(0) = 0; \\
&\text{fib}(1) = 1; \}
\end{aligned}
\]

\[
\begin{aligned}
&\text{for } i = 2 \text{ to } n \}
\text{fib}(i) = \text{fib}(i-2) + \text{fib}(i-1); \}
\end{aligned}
\]

\[
\begin{aligned}
T(n) &= O(n) \\
S(n) &= O(n)
\end{aligned}
\]
Thursday Aug29th

Asymptotic Analysis:

Big O notation: upperbound
A function f(n) is O(g(n)) if g(n) multiplied by a constant
factor is the upper bound of f(n).

T(n)=O(log(n)): binary search
T(n)=O(n^2): borderline

Space often matters more than time.
We want algorithms with linear time and space requirements.

Omega: lowerbound
theta: both upper and lower bound (tight bound)

Dynamic Programming:

1) Solve all subproblems (identify the subproblems)
2) Develop a schedule for solving the subproblems (usually the
   smallest and easiest first, usually bottom-up)

Knapsack:

MAX_WEIGHT—Capacity of the knapsack
n items 1...n
V[i] the value of item i
W[i] the weight of item i
Goal: maximize value without exceeding the MAX_WEIGHT (subject
to the weight constraint)

S[i,j]=max value possible possibly using items 1,2,...,i with
total weight at most j. (We want to get S[n, MAX_WEIGHT])

for (k=0; k<=MAX_WEIGHT; k++){
    S[0,k]=0;
}

for (l=1;l<=n;l++){  
    for (k=0; k<=MAX_WEIGHT; k++){
        if (k<W[l]){
            S[l,k]=S[l-1,k];
        }else{
            if (V[l]+S[l-1,k-W[l]]>S[l-1,k]){  
                S[l,k]=V[l]+S[l-1,k-W[l]];
            }else{
                S[l,k]=S[l-1,k];
            }
        }
    }
}
To reconstruct the solution: see whether you use a previously constructed solution and mark.
Divide and Conquer

1. Divide the problem into simpler subproblems.
   (Small problems of the same form)
2. Solve the subproblems. (Usually recursively)
3. Combine the solutions to the subproblems to form the solution to the original problem.

Express Running Time using a Recurrence:

\[ T(n) = T_{\text{divide}}(n) + T_{\text{solve}}(n) + T_{\text{combine}}(n) \]

Often \[ a \cdot T\left( \frac{n}{b} \right) \]

size of subproblems

Mergesort:

Goal = Sort an array of items.

Input: \[ 28 41 57 8 \]

13 48 \[ 25 78 \]

Merge:

\[ 12345788 \]

\[ T_{\text{merge}}(\frac{n}{2}) = O(n) \]

& Running Time of Merge Sort:

\[ T(n) = O(1) \]
\[ T(n) = T_{\text{divide}} + T_{\text{solve}} + T_{\text{combine}} \]
\[ = O(1) + 2T\left( \frac{n}{2} \right) + O(n) \]

Mergesort
Master Recurrence Theorem:

\[ T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
\frac{a}{2} T\left(\frac{n}{b}\right) + O(n^c) & \text{if } n \geq 2 
\end{cases} \]

**Theorem:** \( T(n) = \begin{cases} 
aT\left(\frac{n}{b}\right) + O(n^c), n \geq 2 \\
O(1), n = 2 \end{cases} \)

If \( C > \log_b a \) then \( T(n) = O(n^c) \)

If \( C < \log_b a \), then \( T(n) = O(n^{c \log_b a}) \)

If \( n^c = \Theta(n^{k \log_b a} \cdot \log^k n) \), \( k > 0 \), \( k < 1 \)

then \( T(n) = O(n^{c \log^k n}) \)

In our case, \( a = 2 \), \( b = 2 \), \( c = 2 \), \( k = 0 \)

\[ n^c = n^2 = n^{\log_2 b} \cdot \log^2 n = n \]

\[ \Rightarrow T(n) = O(n \log n) \quad (OK) \]

Sorting requires \( \Omega(n \log n) \) comparison \( \Rightarrow \) Proposed is optimal in terms of time, but requires linear extra space.

**Brute Force:**

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]

\[ = 2 \left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n \]

\[ = 4T\left(\frac{n}{4}\right) + 2n \]

\[ = 4 \left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n \]

\[ = 8T\left(\frac{n}{8}\right) + 3n \]

\[ \vdots \]

\[ = n \cdot T(1) + (\log_2 n \cdot n) \]

\[ = n(\log_2 n + 1) \]
\[ T(n) \]

\[ \frac{\log_2 n}{2} = n \]

\[ \frac{2}{3} \times 2 \]

(II add each level)

\[ \text{# of levels} = \log_2 n \]

\[ \text{Time } \geq n \cdot \log_2 n \]

\[ T(n) = 2(T(\frac{n}{2})) + n^2 \quad \text{By Master} = O(n^2) \]

\[ T(2) \]

\[ \frac{4 \cdot (4)}{4} = 4 \]

\[ \frac{4}{3} \times 2^0 = 2n \]

\[ \eta(T(n)) = n \]
Pseudo Code For Merge:

global array A ← to be sorted
global array B ← scratch space

void mergesort (int start; int len) {
    if (len == 1) return;
    int half = len/2;
    int middle = start + half;
    mergesort (start, half);
    mergesort (middle, len - half);
    merge (start, half, middle, len - half);
    return;
}

merge (start1, len1, start2, len2) {
    int j = 0; int k = 0;
    for (int i = 0; i < (len1 + len2); i++) {
        if (j == len1) { B[i+j] = A[start2+k]; k++;
        else if (k == len2) { B[i+j] = A[start1+i]; j++;
        else if (A[start1+i] < A[start2+k]) { B[i+j] = A[start1+i]; j++;
        else B[i+j] = A[start2+k]; k++;
    }
    for (int i = 0; i < (len1 + len2); i++) { A[start1+i] = B[i+j]; }
Sept 5th Thursday

Master Recurrence Theorem

\[ T(n) = a T\left(\frac{n}{b}\right) + n^c \log^n n \]

Case 1. \( C > \log_b a \Rightarrow T(n) = O(n^C) \)
Case 2. \( C < \log_b a \Rightarrow T(n) = O(n \log^n n) \)

Case 3. \( T(n) = a T\left(\frac{n}{b}\right) + n^c \log^n n \)
\[ \Rightarrow \log^n a = C \text{, then } T(n) = O(n \log^{n+1} n) \]

Radix Sort

- MSD = Most Significant Digit first (Divide and Conquer)
- LSD = Least Significant Digit first

Gen = sort items with binary representations.

eg.: 4-bit integers

\[
\begin{array}{c}
\text{Int:} & 2 = 0010 & 0010 & 0010 & 0001 & 0001 \\
7 = 0111 & 0111 & 0011 & 0001 & 0001 \\
1 = 0001 & 0001 & 0000 & 0000 & 0000 \\
13 = 1101 & 1101 & 1101 & 1101 & 1101 \\
0 = 0000 & 0000 & 0000 & 0000 & 0000 \\
9 = 1001 & 1001 & 1001 & 1001 & 1001 \\
\end{array}
\]

Divide and Conquer:

4. Divide = sort according to first digit
   copy A into B with keys starting with 0 ahead
   keys starting with 1
5. Conquer = recursively sort
   keys that start with 0
   keys that start with 1
6. Combine: Nothing to do
Reversal Tree:

- \( O(n) \)
- \( n + n = n \quad - O(n_0) + O(n_0) = O(n) \)
- \( n \quad - O(n) \)

\( \text{No} \quad \text{No} \quad \text{No} \quad \text{No} \)

\( \exists (\text{No} \quad \text{No} \quad \text{No} \quad \text{No}) \)

\( \text{Next} \quad \text{Next} \quad \text{Next} \quad \text{Next} \)

\( n = n \)

\( \Rightarrow \text{Total} = O(n \cdot k) \)

```c
void md-redisx (int *A, int n, int K, int pos)
{
    if (n <= 1) return; // Base case.
    int zeros = 0;
    for (int i = 0; i < n; i++)
        zeros += abs((A[i] + 2*(K-1)-pos) & 1);
    int first_zero = 0;
    int first_one = zeros;
    int *B = malloc (n * sizeof (int));

    for (int i = 0; i < n; i++)
        if ((A[i] + 2*(K-1)-pos) & 1 == 0)
            B[first_zero] = A[i];
            first_zero ++;
            else
                B[first_one] = A[i];
                first_one ++;

    for (int i = 0; i < n; i++)
        A[i] = B[i];
}```
\[ \text{msd-radix}(A^*, \text{zeros}, k, \text{pos}+1); \]
\[ \text{msd-radix}(A^* \text{zeros}, n-\text{zeros}, k, \text{pos}+1); \] // recursive steps

**LSD Radix Sort:**

Sorted according to the last LSD

<table>
<thead>
<tr>
<th>0010</th>
<th>0010</th>
<th>0000</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>0000</td>
<td>0101</td>
<td>0001</td>
</tr>
<tr>
<td>0001</td>
<td>1111</td>
<td>1101</td>
<td>1001</td>
</tr>
<tr>
<td>1101</td>
<td>0010</td>
<td>1101</td>
<td>1001</td>
</tr>
<tr>
<td>0000</td>
<td>1101</td>
<td>0101</td>
<td>1101</td>
</tr>
<tr>
<td>1101</td>
<td>1001</td>
<td>0111</td>
<td>1101</td>
</tr>
</tbody>
</table>

Set according to 2nd LSD

Sort according to 3rd LSD

The LSD last must be stable

---

Stable sorting algorithm: preserves order of keys with same value.

**T(n) = O(k \cdot n)**

```c
void msd-radix (int *A, int n, int k) {
    for (int pos = k; pos > 0; pos --) {
        for (int i = 0; i < n; i++) {
            int zeros = zeros + (1 - (A[i] >> (k - 1 - pos)));
            int first-zero = 0; int first-one = zeros;
            int *B = malloc (n * sizeof (int));
            for (int i = 0; i < n; i++) {
                if ((A[i] >> (k - 1 - pos) & 1) == 0) {
                    B[first-zero] = A[i];
                    first-zero ++;
                } else {
                    B[first_one] = A[i];
                    first_one ++;
                }
            }
            // Copy B back to A
            for (int i = 0; i < n; i++)
                A[i] = B[i];
        }
    }
}
```
Tuesday.

1. Quicksort
2. Quicksort
3. Linear-time deterministic selection.
   - All three are divide-and-conquer algorithms.

**Quickstart**

4. Pick a partition element
   (e.g., a random element, first element, median).

5. Separate the elements <= partition from those > partition.

6. Sort the two groups separately.
   - In memory/In place
   - Not stable

---

```plaintext
next := 0; p := pos + 1

partition (int *A, int n, int pos)
  if (n == 0) return -1;
  if (n == 1) return 0;
  pivot = A[pos]; next = 0;
  swap (A, pos, n-1);
  for (int i = 0; i < n; i++)
    if (A[i] <= pivot)
      swap (A+i, next);
  next++;
  return next;

for (int k = 0; k < 6; k++)
  A[k] = k + 1;
```
quicksort (A, n) {
    if (n<2) {return;
    int pos = random (0, n-1);
    int middle = partition [A, n, pos];
    quicksort (A, middle-1);
    quicksort (A+middle, n-middle);
}

- If everything works well:
  T(0) = O(1)
  T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)

- Worst case:
  T(n) = T(n-1) + O(n) \Rightarrow O(n^2)

- Bad idea to pick 1st element as pivot.
Selection = Given a list of n elements, identify Kth largest.

- Observation = Easy to reduce to sorting.  
  Idea = sort A, pick A[K].

- Key Observation = If we can determine the order of any one element, we can eliminate some other candidates for Kth largest.

- Assume without loss of generality that all elements are distinct.

Quick selection =

```c
int QuickSelect (int *A, int n, int k) {
    if (n == 0) return -1;
    int pos = rand(0, n-1);
    int middle = partition [A, A+pos];
    if (middle == k) return A[middle - 1];
    if (middle < k) return QuickSelect (A + middle, n - middle, k);  
    if (middle > k) return QuickSelect (A, k-middle, k);  
}
```

- If all goes well:
  \[ T(n) = O(1) \]
  \[ T(n) = T(n/2) + O(n) \Rightarrow O(n) \]

- Worst Case:
  \[ T(n) = T(n-1) + O(n) \Rightarrow O(n^2) \]
Deterministic Linear-Time Selection

1. Break \( n \) elements into groups of 5, and sort each group = \( O(n) \) time.

2. Find recursively the median of \( \frac{a}{5} \) medians.

3. Determine the order of median of medians:

\[
\frac{a}{5} < \text{order} < \frac{3a}{5}
\]

\( O(n) \Rightarrow \text{compare with every element} \)

4. Recursively discard at least \( \frac{a}{5} \) elements.

Recurrente: \( T(n) = O(1) \)

\( T(n) = T(\frac{4n}{5}) + T(\frac{2n}{5}) + O(n) \)

\( = O(n) \)
Thursday

Searching

1. Binary Search
   - Half recursion.

2. Binary Search Trees

3. Balanced Search Trees
   - Top-down 2-3-4 Trees

**Binary Search**

```c
int search (int *A, int l, int r, int key) {
    if (l > r) { return -1; }
    int mid = (l + r) / 2; /* C runs down */
    if (A[mid] == key) { return mid; }
    if (A[mid] < key) { return search (A, mid + 1, r, key); }
    else { return search (A, l, mid - 1, key); }
}
```

- Tail recursion: after recursive call, just return result without using any local variables.
Iteratively:

```c
int search (A, l, r, key) {
    if (l > r) { return -1; }
    while (l <= r) {
        mid = (l+r)/2;
        if (A[mid] == key) { return mid; }
        if (A[mid] < key) { l = mid + 1; }
        else { r = mid - 1; }
    }
    return -1;  // Problem: Hard to insert/delete
}
```

Binary Search Trees

```c
node* Search (node *root, int key) {
    if (root == NULL) { return NULL; }
    if (root->key == key) { return root; }
    if (root->key < key) {
        root = root->right;
    } else {
        root = root->left;
    }
    return search (root, key);
}
```

- Each node has 2 children, including empty children.
- All keys in left subtree are < key of root; or keys in right subtree are > key of root. In for every sub-tree.

```c
typedef
struct node {
    node *left;
    node *right;
    int key;
} node;
```
```c
node* insert (node* root, int key) {
    if (root->key > key) {
        if (root->left == NULL) {
            root->left = malloc(sizeof(node));
            root = root->left;
            root->key = key;
            root->left = NULL;
            root->right = NULL;
        } else {
            return insert(root->left, key);
        }
    } else {
        return insert(root->right, key);
    }
    return root;
}
```

Insert 4, 2, 3, 1, ...

Depth of tree = 0 - 1

Not Balanced
Deletion

Root

- Assume we always delete root.

Promote Smallest Key
Larger than roots key

- Promote node with smallest key in right subtree.
  - That node has no left child.
  - New left child is left child of old root.
  - New right child is right child of old root.
  - New left child of new root is old parent's old right child.
Balanced Search Trees

- top-down 2-3-4 trees
  - worst case $O(\log n)$ time per search, insert, delete operations.

- Spay Trees
  - Binary search trees
  - $O(\log n)$ amortized time per search, insert, delete
  - operations take $O(n \log n)$ time in worst case.

**Top-Down 2-3-4 Trees**

Store 1, 2, or 3 keys in each node.

# children = # keys + 1

```
+---+---+---+
| 4 | 70 | 79 |
+---+---+---+
   |   |   |
   |   |   |
   +---+---+---+
      |   |   |
      |   |   |
      +---+---+---+
         |   |   |
         |   |   |
         +---+---+---+

-4   4≤ keys < 9      6≤ keys < 90   >90
```

- All leaves are at some distance from root.
  - If there are $k$ keys, how deep can such a tree be?
    - Worst case = all nodes have degree 2, depth = $O(\log n)$

$\Rightarrow$ search takes $O(\log n)$ time
Insert:

1. Search top-down for where new key should be added to tree
2. Break any nodes with 4 children encountered along path.

\[ \text{At root} = 4, 20, 30 \]

\[ \text{ABCD} \longrightarrow \text{ABC} \]

\[ \text{ABCD} \]

\[ \text{Distance to all leaves increased by 1} \]

- 4-child node not at root = Two cases
  - parent has 2 children
  - parent has 3 children

\[ \text{30, 60, 57} \]

\[ \text{ABC} \]

\[ \text{BCDE} \]

\[ \text{30, 32, 65} \]

\[ \text{+2} \]

\[ \text{LDEF} \]
Splay Trees

- Binary search trees
- Every insert, delete, search, starts with a Splay operation

\[ \text{Splay}(T, x) \]

- Move \( x \) to root if \( x \) is in \( T \) to tree
- Move predecessor of \( x \) to root if \( x \) is not present

- First look for \( x \) (or its predecessor) using standard binary search tree search

\[ x = 34 \]

Use rotations to bring \( x \) to root (bottom-up)

- Zig Zag

1. \( x \) is a child of root

2. \( x \) is the left child of \( y \) at left child

\[ \text{zig-zig} \]

\[ \text{zig-zag} \]

\[ T = O(1) \]

For each rotation
After Splay, to delete \( X = \) 

After Splay, to insert \( X = \)

\( X \) has no child

1. Bring \( X \) to root
2. Bring \( X' \) to root of left subtree

To prove \( n \) operations take \( O\log n \) time, we use a potential function argument.

\[ \Phi(T) = \text{potential of tree } T \geq 0, \quad \Phi(\text{empty}) = 0 \]
Accounting Scheme

For each insert, delete, search =
  1) count real work performed
  2) count change in tree potential $\delta$

Why?

- It's easier to count both together than just real work.
1. Amortized Analysis of Splay Trees
2. Heaps.

Before search, insert, or delete, splay a key \( x \) to the root.

\[ \phi(T) \quad \text{potential of tree } T \]

Amortized time \( t \) for one operation =

- actual cost of the operation \( t \)
- change in \( \phi(T) \).

Amortized cost of sequence of \( n \) operations

\[ t_1 + t_2 + \ldots + t_n + (\phi(T_1) - \phi(T_0)) + (\phi(T_2) - \phi(T_1)) + \ldots + (\phi(T_n) - \phi(T_{n-1})) \]

Cost of performed rotations

\[ \phi(T_n) - \phi(T_0) \]

\( T_i \) = tree after \( i \) operations

\( b_i \) = actual cost of \( i^{th} \) operation

Actual cost \( \geq b_i \) = total amortized cost + \( \phi(T_0) - \phi(T_n) \)

Difficult to measure \( \implies \) easier \( \implies \) \( O(\log n) \) \( \implies \) \( O(n \log n) \)
\[\text{Size}(X) = \text{II descendants of } X, \text{ including } X\] (size of \(X\)'s subtree)

\[\text{Size}(X) = 1\]

\[\text{Rank}(X) = \log_2 \text{Size}(X)\]

\[\Phi(T) = \sum_{X \in T} \text{Rank}(X)\]

High \(\Phi(T)\):

\[\text{Low } \Phi(T) = \log(n)\]

\[\Phi(T) \approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \cdots\]

\[\approx 12(n)\]

Balanced?
\[ ATR = 2 + \Phi(T) - \Phi(T) \]
\[ = 2 + (|c(x) - c(x)| + (|c(y) - c(y)| + (|c(z) - c(z)|) \]
\[ = 2 + c'(x) - 2c(x) + c'(z) + (2c(y) - c(x) - c(z)) \]
\[ \leq 3(|c(x) - c(x)|) \]

Show: \[ 2c'(x) - c(x) - c'(z) \leq 2 \]
\[ \Rightarrow \frac{(\text{size}(x))}{\text{size}(z)} \leq \log 4 = 2 \]

\[ \text{size}(x) > \text{size}(y) + \text{size}(z) \]
\[ \text{Assume size}'(x) = \text{size}(x) + \text{size}'(z) \]
\[ \text{size}(x) = \text{size}'(z) = \frac{1}{2} \text{size}(x) \]

\[ \text{Amortized Cost} = 3(|c(x) - c(x)|) + 3(|c(x) - c(x)|) - 3(|c(x) - c(x)|) + 2 \]
\[ = 3(|c(x) - c(x)|) + 2 \]
\[ \leq 3\log n + 1 \]

Total Amortized Cost = 3n\log n + n

Real cost could not be much off from amortized cost.
MODIFIED NOTES =

\[ \text{\#(X)} = \text{rank} \cdot \text{size}(X) \]
\[ \text{\#(X)} = \text{Size} \cdot (X) \]
\[ \text{\#'(X)} = \text{rank} \cdot (X) \cdot \text{after rotation} \]

Amortized Time For Zig-Zag:

\[ \text{Cost} = 2 - h(x) + h'(y) - h(y) + h'(z) \]
\[ \leq 2 - h(x) + h'(y) - h(y) + 1/2 \]
\[ = 2 + h'(x) - 2h(x) + h'(z) \]
\[ > h(x) \leq h(y) \]

New Show:

\[ 2 \leq 2h'(x) - h(x) - h'(z) \]

so that amortized time

\[ \leq 2h(x) - h(x) - h'(z) + h'(x) - 2h(x) + h'(z) \]

\[ = 3(h'(x) - h(x)) \]
**Stack**

Operations:
- Push — ?
- Pop — O(1)
- Top — O(1)

Represent using an array:

```
  top →  3
  2      size = 4
  1      fixed sized
  0      array
```

— If the array is full:

  Allocate new memory for an array twice as large, and then copy the old contents to the new array.

— No worries about wasted space.

typedef struct {
  int size;
  int top;
  int *contents}
  Stack;

void push (Stack S, int new) {
  if (top == size - 1) {
    int *new_contents = malloc (2 * size * sizeof(int));
    size = size * 2;
    for (int i = 0; i < top; i++) {
      new_contents[i] = S->contents[i];
    }
    S->contents = new_contents;
    S->size = size;
    S->top = S->top + 1;
    S->contents[S->top] = new;
  } else {
    // Handle case where array isn't full
  }
}
\[ \Phi = 2 \times \text{top} - \text{size} \]

Amortized operation cost for push:

1. Stack not full:
   - Actual cost = 1
   - Charge of potential = 2
   \[ 1 + 2 = 3 \]

2. Stack is full:
   \[ 4 + \text{size} + 2 - \text{size} = 3 \]

Amortized over \( n \) operations:

\[ \sum_{i=1}^{n} \text{Amortized (Up)} = \sum_{i=1}^{n} (\text{Actual} + \Phi) \]

\[ = \sum_{i=1}^{n} \text{Actual} + \Phi_{\text{final}} - \Phi_{\text{initial}} \leq 0 \]

\[ \Rightarrow \sum_{i=1}^{n} \text{Actual} = \sum_{i=1}^{n} \text{Amortized} + \Phi_{\text{initial}} - \Phi_{\text{final}} \]

\[ \Rightarrow \sum_{i=1}^{n} \text{Actual} \leq 3 \cdot n \]

\[ 2 \times \text{top} - \text{size} \geq 0 \]

because
\[ 2 \times \text{top} > \text{size} \]
Heap = Full binary tree.
- Every level is full except possibly the lowest level.
- Lowest level is full from left to right.
* Key root of any subtree ≤ all keys in subtree.

Operations = Insert, Remove-min, delete, Change value, etc.

Can be represented using an array =

\[ a[0, 1, 2, 3, 4, 5, 6] \]

Rule = the left child of \( i \) is at \( 2i + 1 \).
- The right child is at \( 2i + 2 \).
- The parent is at \( \left\lfloor \frac{i-1}{2} \right\rfloor \).

Remove-Min =
1. Record value of key at root.
2. Replace key at root with rightmost leaf key.
3. Remove rightmost leaf.
4. Fix heap property =

If a key is greater than 1 or both children swap with the smaller child.
Fix Heap =

\[ \begin{array}{c}
\text{K=0,} \\
V = a[K] \leftarrow K \text{ has a left child.} \\
\text{while } (2 \times K + 1 < n) \{ \\
\quad J = 2 \times K + 1 \leftarrow \text{has a right child} \\
\quad \text{if } (J < n \&\& a[J] < a[J + 1]) \text{ \{ } \\
\text{if } (V = a[J]) \text{ break; } \\
\quad \text{else } \{ a[K] = a[J] \text{, } K = J \} \\
\} \\
a[K] = V
\end{array} \]