Algorithms I

Definition

An **algorithm** is a *well defined* computation procedure that takes a set of values as input and produces a set of values as output.

Note: the term *well defined* is itself, not well defined.

Definitions

- **Problems** have specific inputs and outputs, input must be finite and not a stream of data.
- **Problem instances** is a specific set of inputs for a problem. A problem can have a Big-O but not a problem instance.
- A program is **correct** if for every input instance, it terminates with the correct output.

Note

- Randomised algorithms is a branch of incorrect algorithms.
- Some algorithms produce incorrect outputs with a probablility (e.g. quantum computing)
- Some algorithms loops infinitely for some inputs, but runs a lot faster that an algorithm that guarantees termination for cases where it terminates. It might be possible to determine whether it will terminate for a specific input before running it.
- Some algorithms gives an output within a margin of error (e.g. A* vs Dijkstra)

Notation

Arrays

- `A[1]` is the first item
- `A[1..n]` is an array of length `n`
- `A.length` is the number of items in the array

We write pseudocode that is

- Imparative
- Block structured
- Fixed form (indentation matters)
- Parameters are passed by values, objects are passed by pointers
- Loop induction (for loops) increments after the final loop

```
for i=1 to 10
  // do stuff
```

After this loop, consider `i=11`

Sorting

Each **key** may have attached payloads.

Insertion Sort

```
for j = 2 to A.length
  Key = A[j]
  i = j - 1
  while i > 0 && A[i] > Key
    A[i + 1] = A[i]
    i = i - 1
  A[i + 1] = Key
```

Use proof by induction for algorithms:

- **Initialisation**: find a property that is true at the start of the program

  P: at the start of each loop, $A[1...j-1]$ contains the $1...j-1$ items in sorted order.

  At the start of the first loop, that is just $[a_1]$, true.

  > **Note**
  > Define "the start of the loop" as: after assigning the value of j, but before running the first line of code in the loop.

- **Maintenance**: show that the property is maintained as the program is running.
- **Termination**: when the program terminates, show the output is correct.

  After the last loop, $A[1...A.\text{length}]$ would have been containing all the items $1...A.\text{length}$ in order.

  And then we can also show the program terminates as it only needs to complete the loop $A.\text{length}$ items.

> **Note**
> Which is the same as the following Hoare logic proof.
>
> Let P, Q be pre and post-conditions, B be body of the loop, C be condition for the loop.
>
> Given:
> 1. $\{P\}\ B\ \{P\}$
> 2. $P \land \neg C \implies Q$
>
> Then $\{P\}$ while C do B $\{Q\}$

Analysis

> **Definition**
> **Analysis** is about predicting the resources (CPU, memory, disk operations) for input instances we haven't ran our algorithm on.

Input measurement	**Description**
$A.\text{length}$	Common for every day senarios, but may be incorrect if each item in array can have variable size (e.g. big integer)
no. of bits/bytes	Useful for algorithm that operates on some bit/byte value.
$2^{A.\text{length}}$	Overestimates the size in most cases, but can be used for search lists.

> **Definition**
> The **running time** of a program is the number of **basic operations**. (as they all cost 1)
>
> | **Basic operation** | **Cost** |
> |---|---|
> | Indexing an array $A[i]$ | 1 |
> | Arithmetic operation | 1 |
> | Comparisons | 1 |

Basic operation	Cost
Assigment to variables	1

One basic operation might not be equal to one clock cycle, if you change the cost of the basic operations, the running time changes.

Note
Comparisons (numbers) is usually done by subtracting one from another, then compare with 0.

Order of Growth

- $\Theta(g(n))$ is the **asymptotic tight bound** for $g(n)$
$$f(n) \in \Theta(g(n)) \Longrightarrow \exists c_1, c_2, n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n))$$
- $O(g(n))$ is the **asymptotic tight upper bound** for $g(n)$
$$f(n) \in O(g(n)) \Longrightarrow \exists c, n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : f(n) \leq cg(n))$$
- $\Omega(g(n))$ is the **asymptotic tight lower bound** for $g(n)$
$$f(n) \in \Omega(g(n)) \Longrightarrow \exists c, n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : cg(n) \leq f(n))$$
- $o(g(n))$ is the **asymptotic non-tight upper bound** for $g(n)$
$$f(n) \in o(g(n)) \Longrightarrow \forall c \in \mathbb{R}^+ : (\exists n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : f(n) < cg(n)))$$
- $\omega(g(n))$ is the **asymptotic non-tight lower bound** for $g(n)$
$$f(n) \in \omega(g(n)) \Longrightarrow \forall c \in \mathbb{R}^+ : (\exists n_0 \in \mathbb{R}^+ : (\forall n \geq n_0 : cg(n) < f(n)))$$

Properties of Orders of Growth

$$\Theta(g(n)) \subseteq O(g(n))$$
$$\Theta(g(n)) \subseteq \Omega(g(n))$$

- **Transitive**: satisfied by all 5 orders
$$f(n) \in \Theta(g(n)) \land g(n) \in \Theta(h(n)) \Longrightarrow f(n) \in \Theta(h(n))$$
- **Reflexive**: satisfied by the tight bounds Θ, O, Ω
$$f(n) \in \Theta(f(n))$$
- **Symmetric**: satisfied by Θ
$$f(n) \in \Theta(g(n)) \Longrightarrow g(n) \in \Theta(f(n))$$

Analysis of Insertion Sort

```
for j = 2 to A.length          // ran (n-1)+1 times
  Key = A[j]                   // ran n-1 times
  i = j - 1                    // ran n-1 times
  while i > 0 && A[i] < Key    // ran sum_(j=2)^n t_j times
    A[i+1] = A[i]              // ran sum_(j=2)^n (t_j - 1) times
    i = i - 1                  // ran sum_(j=2)^n (t_j - 1) times

  A[i+1] = Key                 // ran n-1 times
```

Where t_j is the number of times the while loop is tested on the jth cycle.

- Best case: $t_j = 1$ then $T(n) = pn + q$
- Worst case: $t_j = j$ then $T(n) = pn^2 + qn + r$
- Average case: the claim is that on average, half of the keys in $A[1...j-1]$ will be less than $A[j]$

  $t_g = j/2$ gives $T(n) \in O(n^2)$

The worst case is useful because
- It gives the upper bound on resource
- Often the same as the average case

Insertion sort is an **incremental algorithm**: it builds up an output that satisfies some properties.

Divide and Conquer

1. Split into 2 or more smaller subproblems.
2. call the same algorithm on each subproblem recursively.
3. Combine solutions to the subproblems to build the solution to the original problem.

> **Note**
> Recursion will terminate because the subproblem will get smaller and smaller.

Merge Sort

```
// we are sorting A[p..r]
if p < r
  q = floor((p + r) / 2)
  MergeSort(A, p, q)
  MergeSort(A, q + 1, r)
  Merge(A, p, q, r)
```

And `Merge` defined as

```
n1 = q - p + 1
n2 = r - q

L = new Array(1 .. n1 + 1)
R = new Array(1 .. n2 + 1)

L[1 .. n1] = A[p .. q]
L[n1 + 1] = infinity
R[1 .. n2] = A[q + 1 .. r]
R[n2 + 1] = infinity

i = j = 1

for k = p to r
  if L[i] <= R[j]
    A[k] = L[i]
    i = i + 1
  else
    A[k] = R[j]
    j = j + 1
```

- If the length of the array is not a power of 2, pad ∞ to the end so that it is.
- After sorting, remove the added ∞ at the end of the sorted array.

The input array is modified, `Merge` has no return value.

Recurrence Relations

The input size is lengh of the region to be sorted $n = r - p + 1$

Let $T(n)$ be the cost of solving MergeSort(A, p, r)

- If $p = r$, $T(1) = 1$
- If $p < r$

<table>
<tr><th>Action</th><th>Cost</th></tr>
<tr><td>Calculate q</td><td>$\Theta(1)$</td></tr>
<tr><td>Calls itself on 2 subproblems</td><td>$T(n/2) \times 2$</td></tr>
<tr><td>Calls Merge(A, p, q, r)
<table>
<tr><th>Action</th><th>Cost</th></tr>
<tr><td>Creates 2 arrays of length $n + 2$</td><td>$\Theta(n)$</td></tr>
<tr><td>Loop n iterations: assign into array and increment i or j</td><td>$\Theta(n)$</td></tr>
</table>
</td><td>$\Theta(n)$</td></tr>
</table>

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= \Theta(1) \text{ work} + 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \text{ work} \\
&= k_1 + 2 \cdot T\left(\frac{n}{2}\right) + k_2 \cdot n
\end{aligned}
$$

Definition

A **closed form solution** is not defined in terms of itself through direct or indirect recursion.

$$
\begin{aligned}
T(n) &= k_1 + k_2 \cdot n + 2 \cdot T\left(\frac{n}{2}\right) \\
&= k_1 + k_2 \cdot n + 2 \cdot \left(k_1 + k_2 \cdot \frac{n}{2} + 2 \cdot T\left(\frac{n}{4}\right)\right) \\
&= k_1 + k_2 \cdot n + 2 \cdot \left(k_1 + k_2 \cdot \frac{n}{2} + 2 \cdot \left(k_1 + k_2 \cdot \frac{n}{4} + 2 \cdot T\left(\frac{n}{4}\right)\right)\right) \\
&\vdots \\
&= k_1 \cdot \underbrace{(1 + 2 + 4 + \ldots)}_{\log n \text{ terms}} + k_2 \cdot n \cdot \underbrace{(1 + 1 + 1 + \ldots)}_{\log n \text{ times}} + 2^{\log n} \cdot T(1) \\
&= k_1 \cdot (n - 1) + k_2 \cdot n \log n + n \\
&\in \Theta(n \log n)
\end{aligned}
$$

Note

We preserved the equal signs instead of saying "this term dominates" so we know $T(n) \in \Theta(f(n))$ instead of just $O(f(n))$

If the array length is not a power of 2

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + k_1 + k_2 \cdot n$$

Which gives the same solution.

The Master Theorem

Let $a \geq 1$ and $b > 1$ be constants.

- $T(1) = 1$
- $T(n) = a \cdot T(n/b) + f(n)$

> **Note**
> n/b can be interpreted as ceil or floor, it doesn't matter.

$$f(n) \in O\left(n^{-\varepsilon + \log_b a}\right) \text{ for some } \varepsilon > 0 \implies T(n) \in \Theta\left(n^{\log_b a}\right)$$
$$f(n) \in \Theta\left(n^{\log_b a}\right) \implies T(n) \in \Theta\left(n^{\log_b a} \cdot \lg n\right)$$
$$f(n) \in \Omega\left(n^{\varepsilon + \log_b a}\right) \text{ for some } \varepsilon > 0 \text{ and } f(n/b) \leq cf(n)$$
$$\text{for some } c > 1 \text{ for all sufficiently large } n \implies T(n) \in \Theta(f(n))$$

> **Note**
> There is an extended master theorem for conditions between case 2 and 3.

Quicksort

```
QuickSort(A, p, r)
  if p < r
    q = partition(A, p, r)
    QuickSort(A, p, q - 1)
    QuickSort(A, q + 1, r)

Partition(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
    if A[j] <= x
      i = i + 1
      swap(A[i], A[j])
  swap(A[i + 1], A[r])
  return i + 1
```

The strategy: **divide** (partition array into 3 parts), **conquer** (recurse on L and G), **combine** (no-op).

Requirements for **Partition(A, p, r)** is
- Pick any element as pivot
- Rearrange so array looks like $[\text{items} \leq \text{pivot}, \text{pivot}, \text{items} \geq \text{pivot}]$

$$[L, x, G]$$

Proof

Let P be the statement:
1. If $p \leq k \leq i$ then $A[k] \leq x$
2. If $i + 1 \leq k \leq j - 1$ then $A[k] > x$
3. If $k = r$ then $A[k] = x$

> **Note**
> No statements made for $j \leq x \leq r - 1$ which is the unprocessed region.

- **Initialisation**: (1) and (2) vacuously true, (3) is true by definition.
- **Maintenance**

  - Case does not enters if branch: (1) and (3) remains true, (2) is true as the new item $> x$
  - Case enters if branch: (3) remains true, (1) and (2) are true as their new items both satisfies contraint.
- The final swap ensures post-condition.

Performance

The number of comparisons depends on how Partition splits the array.

- **Best case**: partition splits array in half

$$T(1) = 1$$
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + k \cdot n$$
$$T(n) \in \Theta(n \log n)$$

  by master theorem.

- **Unbalanced partition**, e.g.

$$T(1) = 1$$
$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + kn$$

  The **computation tree** is unbalanced.
  - The shallowest node has depth $\log_4 n$
  - The deepest node has depth $\log_{4/3} n$

  Both gives $\Theta(n \log n)$, so any ratio split gives $\Theta(n \log n)$
- **Worst case** when the pivot is the largest/smallest item in array.

$$\begin{aligned} T(n) &= T(n-1) + T(0) + kn \\ &= T(n-1) + kn \\ &\in \Theta(n^2) \end{aligned}$$

  The probability of worst case on each split is $2^n / n!$, when n is small, this is quite significant.
- **Constant case** where a constant number of items are partitioned to one side. Still $\Theta(n^2)$

Order Statistic

Definition

The ith order statistic is the ith smallest item in a set.

- Input: a set A, and an integer i
- Output: $x \in A$ so it is larger than exactly $i - 1$ other elements.

- Selecting the minimum and maximum item is simple at $O(n)$
- Selecting the ith item can be done in $\Theta(n \lg n)$
  1. Sort the array
  2. Get the ith element

```
QuickSelect(A, p, r, i)
  if p = r
    return A[p]
```

```
q = Partition(A, p, r)
k = q - p + 1
if i == k
  return A[q]
else if i < k
  return QuickSelect(A, p, q - 1, i)
else
  return QuickSelect(A, q + 1, r, i - k)
```

Worst case:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + kn \\ &\in \Theta(n^2) \end{aligned}$$

Optimisations

- Randomise input data before starting
- Choose pivot at random
- Medium of 3: choose the median of 3 selected items as pivot.

  Hitting worst case if:
  - One of the 3 element is max or minimum element
  - The other median element is right next to the max/minimum element.

  This has probability $2/n^2$
- Median of medians:
  1. Consider array as groups of 5 elements.
  2. Pick the median of each group with insertion sort.
  3. Use quick select to select the median of the medians as pivot.

  The final median is a median of $\lceil n/5 \rceil$ "medians"
  - Half the "medians" are greater than the median.
  - Number of elements greater/smaller than the pivot is

$$\text{number of elements} \geq 3 \cdot \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil - 2 \right\rceil \geq \frac{3n}{10} - 6$$

> **Note**
> -2 removes 2 groups for worst case, includes the last group that is possibly incomplete, and the group that the pivot is in.

Worst case is $3n/10 - 6$ on one side and $7n/10 + 6$ on the other.

Suppose

$$\begin{aligned} T(n) &= k \quad \text{if } n < 140 \\ T(n) &= T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + kn \quad \text{as we are considering the worst case} \end{aligned}$$

If we guess that $T(n) \in \Theta(n)$, then

$$\begin{aligned} T(n) &\leq c \cdot \left\lceil \frac{n}{5} \right\rceil + c \cdot \left(\frac{7n}{10} + 6 \right) + kn \\ &\leq \frac{cn}{5} + c + \frac{7cn}{10} + 6c + kn \\ &= cn + \left(-\frac{cn}{10} + 7c + kn \right) \end{aligned}$$

If $(-cn/10 + 7c + kn) \leq 0$, then $T(n) \in O(n)$
- (Not in handout) Also need to check lower bound is in $\Omega(n)$ to show $T(n) \in \Theta(n)$

HeapSort

Definition
- A **heap** is a full tree, except the lowest level, which is filled from left to right.
- The **ordering property**: for a **min-heap** the key of every node is less than its children. Similar for a **max heap**.

Heaps can be stored as a tree or in an array:
- Root node at $A[1]$
- Left/right child of athe node at $2i$ and $2i + 1$
- Parent of a node at $\lfloor i/2 \rfloor$
- A child exist only if $2i$ and/or $2i + 1$ is $\leq$ the array's length
- A node has no parent if $\lfloor i/2 \rfloor = 0$

Storing in array uses less memory as no pointers need to be stored.

Heaps are **semi-structures** - it only maintain **partial order**.
- Smallest item at the root
- 2nd smallest item in 2 possible places
- 3rd smallest item in 3 places, etc.

A semi-structure is cheaper to build than fully structured data structures.

Consider operations on a **max-heap**.

Operation	$O(n)$	**Description**
MaxPeek	$O(1)$	Just return the root node
MaxReheapify	$O(\lg n)$	Call on heaps where the root node is larger than one of its children to repair the heap. 1. Swap root node with child if needed, this damages the child heap 2. Call reheapify on the child heap.
MaxFullHeapify	$O(n)$	Perform swaps on an array so it is a valid heap, by calling reheapify on all nodes from $\lfloor n/2 \rfloor$ down to 1.
MaxExtract	$O(\lg n)$	1. Swap the root with bottom right leaf. 2. Remove the now bottom right leaf. 3. Call reheapify to repair the heap. This is better than removing the root node then merge the heaps, because the ordering property is easier to repair than the structural properity.

So the HeapSort algorithm.

```
MaxFullHeapify(A) # O(n)

for i = A.length downto 2: # O(n lg n)
  # ASSERT: A[i..] is sorted
  swap(A[1], A[i]) # essentially a MaxExtract
  A.length = A.length - 1
  MaxReheapify(A)
```

Cost of `MaxFullHeapify`

- Best case: array satisfies heap properties, no swaps, 2 comparisons per key on $n/2$ keys. $\Theta(n)$
- Worst case: every recursive call of `MaxReheapify` results in a swap. $\Theta(n)$

HeapSort MaxFullHeapify once then calls MaxReheapify $n-1$ times

$$\begin{aligned} T(n) &= k_1 n + k_2 \lceil \lg n \rceil + k_2 \lceil \lg(n-1) \rceil + k_2 \lceil \lg(n-2) \rceil + \ldots \\ &\le k_1 n + k_2 \lg n + k_2 \lg(n-1) + \ldots + k_2 n \\ &= (k_1 + k_2) n + k_2 \lg(n!) \\ &\le (k_1 + k_2) n + k_2 (n \lg n - n) \\ &\in O(n \lg n) \end{aligned}$$

$O(n)$ Sorting Algorithms

Number of required comparisons for comparison sorts are $\in \Omega(n \lg n)$

CountingSort(A, B, k)

Where A is input, B is output, k is top limit on range of values.

```
C = new Array[0..k]

for i = 0 to k:
  C[i] = 0

for j = 1 to A.length:
  C[A[j]] = C[A[j]] + 1

for i = 1 to k:
  C[i] = C[i] + C[i - 1]

# ASSERT: C[n] is the ending index for
# runs of `n`

for j = A.length downto 1:
  B[C[A[j]]] = A[j]
  C[A[j]] = C[A[j] - 1]
```

The time complexity is $\Theta(n + k)$

RadixSort(A, d)

```
for i = 1 to d:
  # sort array A on digit i with any stable sort
```

Where 1 is the least significant digit.

Definition
A **stable sort** preserves the order of inputs when their keys are equal.

- Time complexity is $\in \Theta(d(n+k))$ where k is the number of values a digit can take.
- Essentially run CountingSort once on each digit.

BucketSort(A)

Used for sorting values distributed uniformly in a range.

1. Put values into n buckets
2. Sort values inside each bucket
3. Merge

```
n = A.length
B = new Array[0..n-1]

for i = 0 to n-1:
  B[i] = empty_list

for i = 1 to n:
  # insert A[i] into list B[floor(n * A[i])]

for i = 0 to n-1:
  InsertionSort(B[i])

# concatenate B[0], B[1], .. B[n-1] in ordered
```

$$\begin{aligned} T(n) &= \Theta(n) + \sum_{i=0}^{n-1} O\left((n_i)^2\right) \\ &\vdots \\ &= \Theta(n) \end{aligned}$$

Ending note
The two main types of strategies studied are:
- Incremental
- Divide and conquer

Dynamic Programming

- **Divide and conquer** splits into subproblems that don't overlap.
- **Dynamic programming** is useful when the subproblems do overlap.

Definition
Optimal substructure problems involves maximising or minimising something.

We want any optimal solution if there are many.

Top-down:
1. Start with the problem
2. Split into subproblems
3. Continue until base case is solved

Bottom-up:
1. Start with base case
2. Solve every problem that combines solved subproblems
3. Continue until the original problem is encountered

Avoid solving same subproblem twice by memorising results in a table.

Virtual Machine Hosting Problem (Rod Cutting Problem)

Top-Down

Subdivide a server with n cores into virtual machines, a server machine with i cores has value $v[i]$.

```
def VMify(v, n):
  if n == 0:
    return 0

  q = -1
  for i = 1 to n:
    q = max(q, v[i] + VMify(v, n - i))

  return q
```

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 1 + \sum_{i=0}^{n-1} T(i) \\
&\in O(2^n)
\end{aligned}
$$

If we add caching for the results

$$
\begin{aligned}
T(n) &= 1 + n \\
&= O(n^2)
\end{aligned}
$$

Bottom-Up

```
m[0 .. n] = new Array()
m[0] = 0
for j = 1 to n:
  q = -1
  for i = 1 to j:
    q = max(q, v[i] + m[n - i])
  m[j] = q

return m[n]
```

Greedy Algorithms

Chooses between the subproblems without evaluating all of them, some problems do not have greedy solutions.

Examples of problems with greedy solutions.
- Minimum spanning tree
- Huffman encoding (data compression)
- Calculating change
- Scheduling problems

Activity Selection Problem

- A set of activities uses share resources: an activity runs in interval $[s_i, f_i)$
- Find a maximum size subset of compatible activities.

In an interval $[i, k)$, the subset with the maximum cardinality is

$$S(i,k) = \underset{j}{\operatorname{argmax}} \left(S(i, s_{aj}) \cup \{a_j\} \cup S(f_{aj}, k)\right)$$

We can avoid evaluating for all s_{aj} if we pick the element with the earliest finish time:
- It leaves the fewest conflicts in future choices.

So the greedy algorithm:

1. Sort activites
2. Repeatedly select the activity that finishes first

Note
Greedy solutions only work if locally optimal choice still allows optimal choice to be reached.

Data Structures

Definitions
- The value of a **pointer** is the object's base address.
- **NIL** does not point to any object.

Stack

A LIFO structure with push(item) and pop()

Condition for empty: index for stack top is 0 or 1 (depends if index points to or above top item).

Queue

A FIFO structure with enqueue(item) and dequeue()

Implementation using arrays *(circular buffer)* requires two pointers:
- Head points to first item
- Tail points to or after last item

Need to distinguish between a full queue and an empty queue, as their pointer values are the same.

Solution: only allow $n - 1$ items to be stored in a queue implemented by an array of length n
- Small amount of memory is wasted
- But removes a lot of special case code

Singly Linked List

Definition
List cells are tuples storing the data and a pointer to the next cell.

This allows for:
- Straight linked lists
- Cycle in linked list (a cell points to an earlier cell)
- Two linked lists sharing the same tail

Note
List search will not work on linked lists with cycles, it will require cycle detection.

Doubly Linked List

Similar to singly linked list, but if the list has a cycle, the cycle must include all cells.

Dong Lea's Malloc Algorithm

We want to allocate/deallocate objects in the heap in any order.

Represent free and busy chunks in order they are found in memory in a linked list.
- **Allocate** memory:
  1. Search list for first chunk that is big enough

2. Split the chunk into two: one of them the size to be allocated

- **Free** maintains no two chunks next to each other are blank.

No metadata need to be stored about the size allocated: the allocated size is the space between the two linked list cells.

Memory allocation is often a multiple of 4 bytes, this leaves 0s at the end of the pointer value.
- Store free/busy status in one of the bits
- Store free/busy status of the neighbouring blocks in another bit

---

Rooted Trees

Definition
Rooted trees have a single entry point and no cycles.

Binary Trees

Each node holds:
- A data item
- Pointer to two children
- (Optional) pointer to parent

Binary Search Trees

- The data is a $(\text{key}, \text{payload})$ tuple
- The **binary tree property**:
  - Keys in left subtree $<$ key at node
  - Keys in right subtree $>$ key at node
- No duplicate keys, otherwise bad asymptotic running times.

All operations runs in $O(\lg n)$ time, all recursive algorithms can be rewritten as an iterative algorithm (most languages don't perform tail call optimisations).

<table>
<tr><th>Operation</th><th>Description</th></tr>
<tr><td>$\text{search}(p, k)$</td><td><pre>if p == Nil:
  return Nil
if p.key == key:
  return p
if k < p.key:
  return search(p.left, k)
else:
  return search(p.left, k)</pre></td></tr>
<tr><td>$\text{minimum}(p)$</td><td><pre>if p == Nil:
  return Nil
if p.left == Nil:
  return p
else:
  return minimum(p.left)</pre></td></tr>
<tr><td>$\text{maximum}(p)$</td><td>Similar</td></tr>
<tr><td>$\text{predecessor}(p)$</td><td><pre>if p.left != Nil:
  return maximum(p.left)
y = p.parent
while y != Nil and p = y.left:
  p = y</pre></td></tr>
</table>

Operation	**Description**
	`  y = y.parent` `return y`
successor(p)	Similar
insert(k, v)	Either replace a node with key k with data v, or reach a leaf node and replacing it with node (k, v).
delete(k)	• Find the node, if it has 0 children, replace the node with leaf. • If it has 1 children, replace the node with that children. • If it has 2 children, replace the node with its predecessor (as the predecessor has no right node, so at most 1 child), and delete the predecessor from where it came from.

B-Trees<T>

B-trees are balanced to achieve $O(\lg n)$ worst case.

- **Leaf nodes** hold no keys
- **Internal nodes** hold varying number of keys and payloads

For a B-tree of degree T:
- Internal nodes holds at least $T - 1$ (except root), at most $2T - 1$ (inc. root) keys and payloads
- Node with t keys have $t + 1$ children
- Leaves all exist in same level below root
- Keys in an internal node divides the range of keys in the children

At least 1 key at root level, $2(T - 1)$ keys at first level, $2T(T - 1)$ in second, $2T^2(T - 1)$ in third. Summing up, number of levels to hold N keys is $\log_T((N + 1)/2)$ levels, $\therefore$ performance $\in O(\lg n)$

Search is obvious.

Insert
1. If root node is leaf, replace root node.
2. Traverse down to find k, if found, replace the entry.
3. If reached the bottom level of internal nodes without finding k, increase key count of the internal node by 1.
4. If bottom node key count $> 2T - 1$, split it by the median key into two nodes.
   - Insert the median key into the parent.
   - Add the second half of the node into the parent.
5. If the parent is also full, split it and recurse up if needed. Replace the root pointer if the root node is split.

Insert (Improved)
1. If root node is leaf, replace root node.
2. Traverse down to find k, if found, replace the node. While traversing, split any full nodes encountered.
3. If reached the bottom level of internal nodes without finding k, insert key into the node and increase key count by 1.
4. Instead of recursing up, split by the median key into two nodes of $T - 1$ keys and the median. Insert the median into the root node, and insert the second half as child of the root.

Delete

Start at root search for key k

- If not found, do nothing
- If it is the only key in root, replace root with leaf
- If it is found, but not in bottom level, swap with its predecessor, then remove it
  - If after doing that, the node is less than minimum size, try:
    - Redistributing keys from max in left sibling $\rightarrow$ key in parent $\rightarrow$ min in this node
    - Or equivalent for the right sibling
    - If all siblings are at minimum size, take a key from the parent node, if parent node is also of minimum size, cascade up towards the root.
    - If the only key in root is being taken, merge the root with its immediate children and borrow from there, this is the new root

> **Note**
> When distributing keys between siblings, make the number of keys in each node equal so minimise future merging/splitting.

Red-Black Trees

A BST with nodes coloured red and black:

1. The root is black
2. The leaves are black, and contains no payload
3. Children of a red node are black
4. **Black height balanced**: For each node, all paths to a leaf has equal number of black nodes

Let bh be the **black height** of the tree. The number of nodes in a RB Tree n

$$2^{\mathrm{bh}} - 1 \leq n \leq 2^{2\mathrm{bh}} - 1$$
$$\mathrm{bh} \leq \frac{1}{2} \lg(n+1)$$
$$\mathrm{height} \leq \lg(n+1)$$

A RB tree is **isomorphic to BTree<2>** (see lecture notes)

A **tree rotation** rotates the tree about an edge so the ordering property is preserved.

Priority Queue

Using a Heap

Use the normal MinHeapExtract and MinInsert.

```
def pq_decrease_key(pq, index, k):
  if pq[index] < k:
    error("key already less than k")

  while index > 1 && pq[parent(index)] > pq[index]:
    swap(pq[parent(index)], pq[index])
    index = parent(index)
```

Using a Red-Black Tree

```
def pq_min(pq):
  if pq.root == NIL:
```

```
    error("empty pq")
  x = pq.root
  while x.left != NIL:
    x = x.left
  return x.key

def pq_extract_min(pq):
  min = pq_min(pq)
  rb_delete(pq, min)
  return min

def pq_decrease_key(pq, oldk, k):
  rb_delete(pq, oldk)
  rb_insert(k)
```

Operation	**Min heap**	**Red-black tree**
PQ-Minimum	$O(\lg n)$	$O(\lg n)$
PQ-Extract-Minimum	$O(\lg n)$	$O(\lg n)$
PQ-Decrease-Key	$O(\lg n)$	$O(\lg n)$
PQ-Insert	$O(\lg n)$	$O(\lg n)$

But the min heap has a larger constant of proportionality.

Hash Tables

Hash tables are used as **sparse arrays** or index with non-interger keys.

A hash function : $T \to \text{int}$

- We want the hashed key to be uniformly distributed in a range. Then use remainder to give a uniformly distributed index.
- It is hard for a hash function to give a uniform output: if the key is short, there is not enough **entropy** to map them in different places in the array.

Collision Resolutions (Chaining)

- **Chaining**: each entry in a hash table stores a pointer to a list cell.
- **Negative cache**: if there are a lot of items in the table, store what is *not* on the table instead of what is.
- **Chaining (sorted)**: in normal chaining, if an item is not on hash table, it will have to search for the entire list. If the list is sorted, then it only need to search until the keys are larger than what's being sought after.
- **Push on head**: push nodes recording write/deletes on the head of the list, downside time complexity cannot be expressed in terms of number of items in the table.
- **Push on head**, but delete removes the first write node of k it finds. Essentially calling delete *undo* a write.

Collision Resolutinos (Open Addressing)

A **probe function** tries new positions, use the same **probe sequence** when searching. We want the probe sequence to hit every position.

- **Linear probing** tries the next, and next… positions. Stops when reaches a Nil or returns to the same index.

  Prone to **primary clustering**: build up of long sequences that doesn't contain keys with the same hash value.

- **Quadratic probing** prevents one key hitting the probing sequence of another key.

  $$\text{probe}(\text{key}, i) = (h(\text{key}) + ai + bi^2) \bmod T.\text{size}$$

  Prone to **secondary clustering**: keys that hash to the same value collide with each other in every position.

  Note
  Need to prove that it hits every cell.

- **Double hashing** : $\text{probe}(T, \text{key}, i) = (h_1(\text{key}) + ih_2(\text{key})) \bmod T.\text{size}$

  We want to prevent h_2 from being zero, so use $h_2(\text{key}) = h_2{}'(\text{key}) \bmod (T.\text{size} - 1) + 1$

To delete an item in open addressing, leave a marker behind that is:
- Treated as an entry when searching
- Treated as invisible when inserting

Maintain counters n_{keys} and n_{markers}, and resize to prevent overloading.

Senario	Action
$n_{\text{keys}} \gg n_{\text{markers}}$	Rehash into larger table
$n_{\text{keys}} \approx n_{\text{markers}}$	Rehash into table of same size
$n_{\text{keys}} \ll n_{\text{markers}}$	Rehash into smaller table

---

`END Algorithms I`