CS 240 Note
velo.x

Contents

Waiting to be added:

- Module 3: quicksort and lower bound for comparison based sorting and Non-Comparison-Based sorting

1 Module 1

1.1 Algorithm Design

Definition 1.1.1 (Problems)**.**

- **Problems:** Given a problem instance, carry out a particular computational task.
- **Problem Instance:** Input for the specified problem.
- **Problem Solution:** Output (correct answer) for the specified problem instance.
- **Size of problem instance:** Size(I) is a positive integer that is a measure of the size of the instance I.

Definition 1.1.2 (Efficiency of Algorithms/ Programs)**.**

- Running time: amount of time
- Auxiliary space: amount of additional memory
- * the amount of time and/or memory required by a program will depend on Size(I) (usually denoted by “n"), the size of the given problem instance I.

1.2 Algorithm Analysis

To overcome dependency on hardware/software:

- Algorithms are presented in structured high-level language-independent **pseudo-code**.
- Analysis of algorithms is based on an idealized computer model.
- Instead of time, count the number of **primitive operations**
- The efficiency of an algorithm (with respect to time) is measured in terms of its growth rate (this is called the **complexity** of the algorithm).

Random Access Machine (RAM) model:

A set of memory cells, each of which stores one item (word) of data. Implicit assumption: memory cells are big enough to hold the items that we store.

Any access to a memory location takes constant time.

Any primitive operation takes constant time.

Implicit assumption: primitive operations have fairly similar, though different, running time on different systems

The running time of a program is proportional to the number of memory accesses plus the number of primitive operations.

1.3 Order Notation

Definition 1.3.1.

O: $f(n) \in O(g(n))$ if exist constants $c > 0$ and $n_0 > 0$ that $\forall n \geq n_0$, $|f(n) \leq c|g(n)|$.

Ω: $f(n) \in \Omega(g(n))$ if exist constants $c > 0$ and $n_0 > 0$ that $\forall n \geq n_0$, $c|g(n)| \leq |f(n)|$.

Θ: $f(n) \in \Theta(g(n))$ if exist constants $c_1, c_2 > 0$ and $n_0 > 0$ that $\forall n \geq n_0$, $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$.

o: $f(n) \in o(g(n))$ if for all constants $c > 0$, exists $n_0 > 0$ such that $|f(n) \leq c|g(n)|$ for all $n \geq n_0$.

ω: $f(n) \in \omega(g(n))$ if $g(n) \in o(f(n))$.

Remark: We always want **tight** asymptotic bound.

Proposition 1.3.1. Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$, suppose that

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)} \text{ exists}$$

then

$$f(n) \in \begin{cases} o(g(n)), & \text{if } L = 0 \\ \Theta(g(n)), & \text{if } 0 < L < \infty \\ \omega(g(n)), & \text{if } L = \infty \end{cases}$$

Proposition 1.3.2 (relations between order notations)**.**

- $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$
- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow f(n) \in O(f(n))$
- $f(n) \in o(g(n)) \Leftrightarrow f(n) \notin \Omega(f(n))$
- $f(n) \in \omega(g(n)) \Leftrightarrow f(n) \in \Omega(f(n))$
- $f(n) \in \omega(g(n)) \Leftrightarrow f(n) \notin O(f(n))$

Proposition 1.3.3. Assume $f(n) \geq 0$ and $g(n) \geq 0$ for all $n \geq 0$,

- Identity rule: $f(n) \in \Theta(f(n))$
- Transitivity:
  - If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$
  - If $f(n) \in \Omega(g(n))$ and $g(n) \in \Omega(h(n))$ then $f(n) \in \Omega(h(n))$
- Maximum rule:
  - $O(g(n) + g(n)) = O(\max\{f(n), g(n)\})$
  - $\Omega(g(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$

1.4 Growth

- $\Theta(1)$: constant complexity
- $\Theta(\log n)$: logarithmic complexity
- $\Theta(n)$:linear complexity
- $\Theta(n \log n)$: linearithmic
- $\Theta(n \log^k n)$: for some constant k (quasi-linear)
- $\Theta(n^2)$: quadratic complexity
- $\Theta(n^3)^3$: cubic complexity
- $\Theta(2^n)$: exponential complexity

1.5 Recurrence Relations

Recursion	Resolves to	example
$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$	Binary search
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$	Mergesort
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$	Heapify
$T(n) = T(cn) + \Theta(n),\ 0 < c < 1$	$T(n) \in \Theta(n)$	Selection
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$	Range search
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log \log n)$	Interpolation search

2 Module 2: Priority Queues and Heaps

2.1 ADT

From video: Heap 1

Definition 2.1.1. Stack is an ADT consisting of a collection of items with operations in LIFO order:

- push: inserting an item
- pop: removing the **most** recently inserted item

Definition 2.1.2. Queue is an ADT consisting of a collection of items with operations in FIFO order:

- enqueue: inserting an item
- dequeue: removing the **least** recently inserted item

2.2 Priority Queue

Definition 2.2.1. Priority Queue is consisting of a collection of items each having a **priority**(key) with operations:

- insert: inserting an item tagged with a priority
- deleteMax: removing the item of highest Priority

Realization 1: unsorted arrays

- insert: O(1)
- deleteMax: O(n)

Realization 2: sorted arrays

- insert: O(n)
- deleteMax: O(1)

Realization 3: heap

Algorithm 2.2.1 (Using a priority queue to sort)**.**

```
PQ-Sort(A[0 ... n-1])
1.   initialize PQ to an empty priority queue
2.   for k <- 0 to n-1 do
3.        PQ.insert (A[k], A[k])         (priority and item are equal to A[k])
4.   for k <- n-1 down to 0 do
5.        A[k] <- PQ.deleteMax()
```

2.3 Heap

Definition 2.3.1 (heap)**. Heap** is a certain type of binary tree with two properties:

- Structural property: "complete"
- Heap-order property: for any node i, the key of the parent of i is >= to key of i

Remark: the height of a heap with n node is $\Theta(\log n)$.

Navigation:

- root: 0, last node: $n - 1$
- child: $2i + 1$, $2i + 2$
- parent: $\lfloor \frac{i-1}{2} \rfloor$

Algorithm 2.3.1 (insert in Heap)**.** Time: $O(\log n) = O(height\ of\ heap)$

```
fix-up(A, i)
i: an index corresponding to a node of the heap
1.    while parent(i) exists and A[parent(i)].key < A[i].key do
2.        swap A[i] and A[parent(i)]
3.        i  parent(i)
```

Algorithm 2.3.2 (deletemax in Heap)**.** O(height of heap) = O(log n)

```
fix-down(A, n, i)
A: an array that stores a heap of size n
i: an index corresponding to a node of the heap
1. while i is not a leaf do
2. // Find the child with the larger key
3. j  left child of i
4. if (j is not last(n) and A[j + 1].key > A[j].key)
5. j  j + 1
6. if A[i].key  A[j].key break
7. swap A[j] and A[i]
8. i  j
```

2.4 Tutorial

Question: merge k sorted arrays into one

IDEA: using a minHeap to track smallest elements of each array which is not in the output

Question: an input array L of co-prime integers, output k-th smallest fraction $\frac{L[i]}{L[j]}$, $i < j$.

3 Module 3: Sorting, Selection

3.1 Quick Select

Problem: Given an array A of n numbers and $0 \leq k \leq n$ find the element that would be at position k of the sorted array of A.

– Selection can be done with heaps in time $\Theta(n + k \log n)$, where k is the index.

Algorithm 3.1.1 (quick-select1)**.**

two subroutines:

1. choose-pivot(A): return an index p in A, and use the pivot-value to rearrange the array.
2. partition(A, p): rearrange A and return pivot-index i so that
   - the pivot-value is at A[i]
   - all items in A[0, . . . , i 1] are $\leq$ v, and
   - all items in A[i + 1, . . . , n 1] are $\geq$ v.

```
partition(A, p)
  A: array of size n, p: integer s.t. 0  p < n
    1.  swap(A[n - 1], A[p])
    2.  i <- -1, j <- n - 1, v <- A[n - 1]
    3.  loop
    4.    do i <- i + 1 while i < n and A[i] < v
    5.    do j <- j - 1 while j > 0 and A[j] > v
    6.    if i >= j then break (goto 9)
    7.    else swap(A[i], A[j])
    8.  end loop
    9.  swap(A[n - 1], A[i])
    10. return i
```

main algorithm:

```
quick-select1(A, k)
  A: array of size n, k: integer s.t. 0 <= k < n
    1.  p <- choose-pivot1(A)
    2.  i <- partition(A, p)
    3.  if i = k then
    4.    return A[i]
    5.  else if i > k then
    6.    return quick-select1(A[0, 1, . . . , i - 1], k)
    7.  else if i < k then
    8.    return quick-select1(A[i + 1, i + 2, . . . , n - 1], k - i - 1
```

analysis:

- Worst Case: $\Theta(n^2)$
- Best Case: $\Theta(n)$
- Average Case: $\Theta(n)$

3.2 Randomized Algorithms

Definition 3.2.1. A **randomized algorithm** is one which relies on some random numbers in addition to the input.

The run-time will depend on the input and the random numbers used.

Goal: Shift the dependency of run-time from what we cant control (the input) to what we can control (the random numbers).

Definition 3.2.2. The **expected running time** $T^{(exp)}(I)$ for instance I is the expected value for $T(I, R)$:

$$T^{(exp)}(I) = E[T(I, R)] = \sum_{R} T(I, R) \cdot Pr[R]$$

3.3 Lower bounds for sorting

Theorem 3.3.1. Any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparison operations to sort n distinct items.

3.4 Non-Comparison-Base sorting

3.5 Tutorial

Question 1: - input: an array which is partially sorted for $n - n^{\varepsilon}$, $0 < \varepsilon < 1$ - output: completely sorted array - requirement: O(n)

4 Module 4: BST, AVL

Definition 4.0.1. Dictionary: collection of items each with a **key** and some **data** (**key-value pair**).

- Common assumptions:

- Dictionary has n KVPs.
- Each KVP uses constant space.
- Keys can be compared in constant time.

Definition 4.0.2. AVL tree: BST tree with height-balance property, that is:

$$| \text{height(L) - height(R)} | \leq 1 \, ,$$

where L is the left tree and R is the right tree.

Balance: height(R) - height(L) $\in \{-1, 0, 1\} \Rightarrow \{$left-heavy, equal, right-heavy$\}$.

Remark: Each node consists of a key, a value and a height (or balance). Height of empty tree: -1, height of a single node: 0.

Theorem 4.0.1. An AVL tree on n nodes has $\Theta(\log n)$ height.

- search, insert, delete worst case: $\Theta(\log n)$

Proof. Define $N(h)$ to be the least number of nodes in a height-h AVL tree.

$N(0) = 1$, $N(1) = 2$, $N(2) = 4$, then

$N(h) = N(h-1) + N(h-2) + 1$, one is L and one is R. □

Algorithm 4.0.1 (insertion in AVL tree)**.**

- Steps:

1. insert (k, v) with the usual BST insertion, return the new leaf z where the key is stored.
2. move up the tree from z, updating heights.

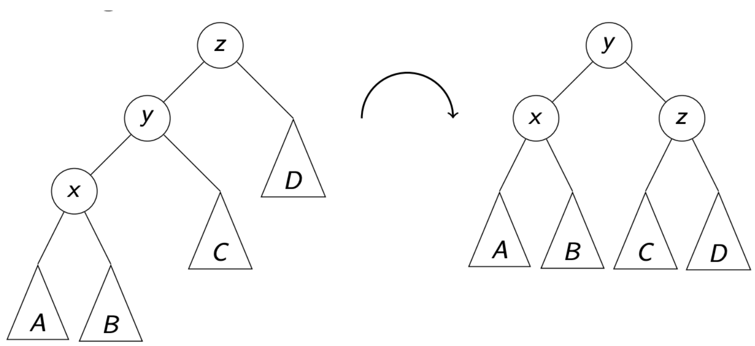
```
AVL::insert(k, v)
    1. z <- BST::insert(k, v) // leaf where k is now stored
    2. z.height <- 0
    3. while (z is not null)
    4.     setHeightFromChildren(z)
    5.     if (|z.left.height - z.right.height| > 1) then
    6.         AVL-fix(x)
    7.         break
    8.     else
    9.         z <- parent of z

setHeightFromSubtrees(u)
    1. u.height <- 1 + max{u.left.height, u.right.height}
```

3. If the height difference becomes ± 2 at node z, then z is unbalanced, re-structure the tree to rebalance.

```
AVL-fix(z)
    if (z.right_height > z.left_height) then
        y <- z.right
        if (y.left.height > y.right.height) then
            x <- y.left
        else x <- y.right
    else
        y <- z.left
        if (y.right.height > y.left.height) then
            x <- y.right
        else x <- y.left
```

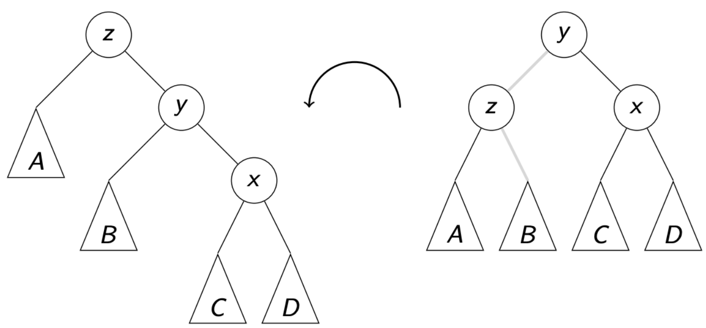
- Different types of unbalance:

- Right Rotation:



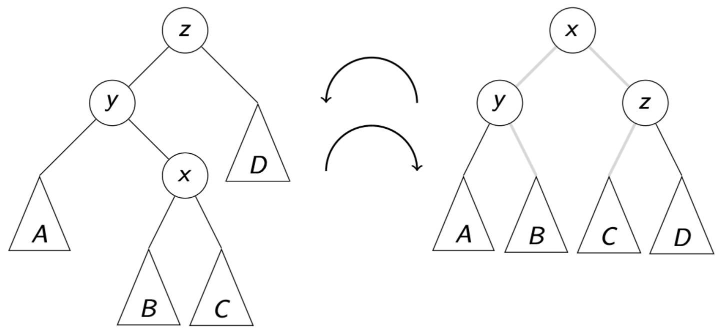

```
rotate-right(z)
    y <- z.left,  z.left <- y.right,  y.right <- z
    setHeightFromSubtrees(z), setHeightFromSubtrees(y)
    return y // returns new root of subtree
```

- Left Rotation:



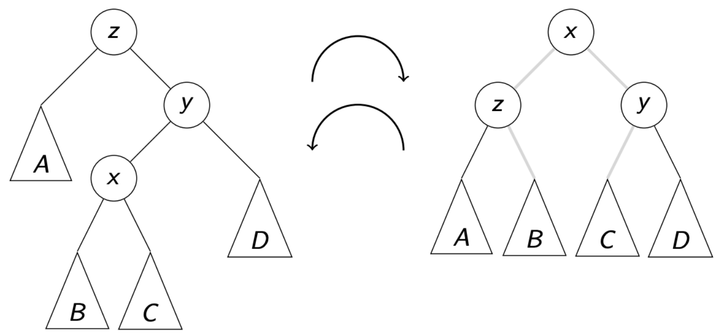

```
rotate-left(z)
    y <- z.right,  z.right <- y.left,  y.left <- z
    setHeightFromSubtrees(z), setHeightFromSubtrees(y)
    return y // returns new root of subtree
```

- Double Right Rotation:




```
double-rotate-left(z):
- First: perform a left rotation at y,
- Second: a right rotation at z.
```

- Double Left Rotation:




```
double-rotate-left(z):
- First: perform a right rotation at y,
- Second: a left rotation at z.
```

4.1 Tutorial

Question 1: comparison based question

5 Module 5

5.1 Skip List

Definition 5.1.1 (Skip List)**.** Skip List is a hierarchy S of ordered linked lists (levels) $S_0, S_1, ...S_h$:

- Each list S_i contains the special keys $-\infty$ and $+\infty$ (sentinals)
- List S_0 contains the KVPs of S in non-decreasing order.
- Each list is a subsequence of the previous one.
- List S_h contains only the sentinels; the left sentinel is the root

Algorithm 5.1.1 (Skip List Search)**.** **Expected Running time:** $O(\log n)$

```
getPredecessors (k)
1.  p <- topmost left sentinel
2.  P <- stack of nodes, initially containing p
3.  while p.below != NIL do
4.      p <- p.below
5.      while p.after.key < k do p <- p.after
6.      P.push(p)
7.  return P


skipList::search (k)
1.  P <- getPredecessors(k)
2.  p0 <- P.top() // predecessor of k in S0
3.  if p0.after.key = k return p0.after
4.  else return not found, but would be after p0
```

Algorithm 5.1.2 (Skip List Insert)**.** **Expected Running time:** $O(\log n)$

```
skipList::insert(k, v)
1. P <- getPredecessors(k)
2. for (i <- 0; random(2) = 1; i <- i+1) {} // random tower height
3. while i >= P.size() // increase skip-list height?
4.     root <- new sentinel-only list, linked to previous root-list
   appropriately
5.     P.append(left sentinel of root)
6. p <- P.pop() // insert (k, v) in S0
7. k_below <- new node with (k, v), inserted after p
8. while i > 0 // insert k in S1, . . . , Si
9. p <- P.pop()
10. k_below <- new node with k, inserted after p with below-reference to
   kbelow
11. i <- i - 1
```

Algorithm 5.1.3 (Skip List Insert)**.** **Expected Running time:** $O(\log n)$

```
skipList::delete(k)
1. P <- getPredecessors(k)
2. while P is non-empty
3.     p <- P.pop()    // predecessor of k in some layer
4.     if p.after.key = k
5.         p.after <- p.after.after
6.     else break // no more copies of k
7. p <- left sentinel of the root-list
8. while p.below.after is the infinity-sentinel
   // the two top lists are both only sentinels, remove one
```

```
    p.below <- p.below.below
    p.after.below <- p.after.below.below
```

5.2 Reordering

Recall: Unordered list/array implementation of ADT Dictionary search: Θ(n)

If the items are accessed unequally likely, and if we have a probability distribution of the items being accessed, and we can use this information to make search more effective.

Optimal Static Ordering: used when we KNOW the probability distribution beforehand.

Claim: Over all possible static orderings, the one that sorts items by non-increasing access-probability minimizes the expected access cost.

Proof Idea: For any other ordering, exchanging two items that are out-of-order according to their access probabilities makes the total cost decrease.

Example:

key	A	B	C	D	E
frequency of access	2	8	1	10	5
access-probability	2/26	8/26	1/26	10/26	5/26

- Order A, B, C, D, E has expected access cost:

$$1 \cdot \tfrac{2}{26} + 2 \cdot \tfrac{8}{26} + 3 \cdot \tfrac{1}{26} + 4 \cdot \tfrac{10}{26} + 5 \cdot \tfrac{5}{26} \approx 3.31$$

- Order D, B,E,A,C has expected access cost:

$$1 \cdot \tfrac{10}{26} + 2 \cdot \tfrac{8}{26} + 3 \cdot \tfrac{5}{26} + 4 \cdot \tfrac{2}{26} + 5 \cdot \tfrac{1}{26} \approx 2.07$$

Dynamic Ordering: when we DO NOT know the probability distribution deforehand.

- **Move-To-Front heuristic (MTF):** Upon a successful search, move the accessed item to the front of the list
  - works well in practice
  - rule of thumb (**temporal locality**): a recently accessed item is likely to be used soon again.
  - can show: MTF is 2-competitive. No more than twice as bad as the optimal static ordering.

- **Transpose heuristic:** Upon a successful search, swap the accessed item with the item immediately preceding it.
  - Transpose does not adapt quickly to changing access patterns.

6 Module 6

6.1 Lower bound for search

Theorem 6.1.1. In the comparison model, $\Omega(\log n)$ comparisons are required to search a size-n dictionary.

Proof. The number of distinct answers is n+1 and they correspond to leaves.

The corresponding decision trees has at least n+1 leaves and there are at most three children for any node at any level, so the decision tree has height at least $\log_3(n+1) \in \Omega(\log n)$. □

6.2 Interpolation Search

For an ordered array,

- insert, delete: $\Theta(n)$
- search: $\Theta(\log n)$

Interpolation Search(A[l, r], k): Compare at index "$l + \frac{k-A[l]}{A[r]-A[l]} \times (r-l)$",

Works well if keys are uniformly distributed,

- recurrence relation is $T^{(avg)}(n) = T^{(avg)}(\sqrt{n}) + \Theta(1)$, which resolves to $T^{(avg)}(n) \in \Theta(\log\log n)$.
- worst performance $\Theta(n)$

`Interpolation-search(A,n,k)`
A: array of size n, k: key

$l \leftarrow 0$
$r \leftarrow n-1$
while $(A[r]! = A[l])$ && $(k >= A[l])$ && $(k <= A[r])$ **do**
  $m \leftarrow l + \frac{k-A[l]}{A[r]-A[l]} \times (r-l)$
  if $A[m] < k$ **then** $l = m+1$
  else if $k < A[m]$ **then** $r = m-1$
  else return m
if $k = A[l]$ **then return** l
else return "not found"

6.3 Tries

Definition 6.3.1. Trie (also know as radix tree): A dictionary for bitstrings, $\Sigma = \{0, 1\}$.

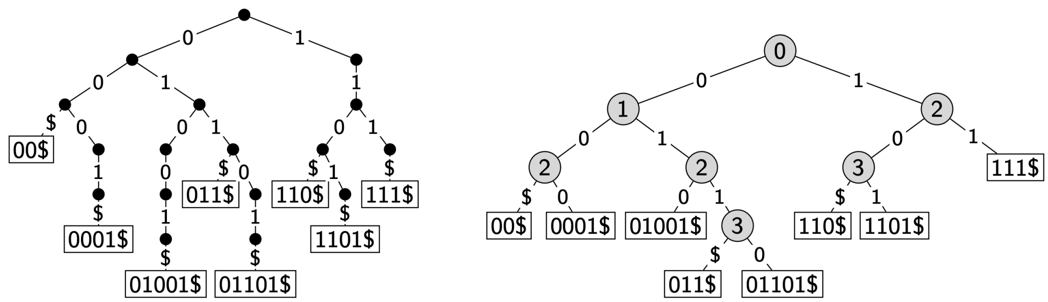
Used for: string, word, |w|, alphabet, prefix, suffix, comparing words,....

Definition 6.3.2. Prefix of a string $S[0, ..., n-1]$: a substring $S[0, ..., i]$ of S for some $0 \leq i \leq n-1$.

Prefix-free: no pair of binary strings in the dictionary where one is the prefix of the other.

Definition 6.3.3. Compressed Trie: compress paths of nodes with only one child

- Each node stores an *index* corresponding to the depth in the uncompressed trie, and store the string at the leaf.
- A compressed trie with n keys has at most $n - 1$ internal nodes.
- All operations take $O(|x|)$ time, where x is the string getting operated on.



7 Module 7: Hashing

7.1 Hashing Introduction

Direct Addressing: For a known $M \in \mathbb{N}$, every key k is an integer with $0 \leq k < M$. We can then implement a dictionary easily by using an array A of size M that stores (k, v) via $A[k] \leftarrow v$.

- search(k): check whether $A[k]$ is NIL.
- insert(k, v): $A[k] = v$
- delete(k,v): $A[k] \leftarrow NIL$.

Hashing: map keys to integers in range $\{0, \ldots, M_1\}$ and then use direct addressing.

- **Hash function** $h : U \rightarrow \{0, 1, \ldots, M-1\}$, where U is some universe that keys all come from.
- **Hash table**: an array T of size M.
- **Collisions:** generally hash function h is not injective, so keys can map to the same integer.
  - we want to insert(k,v), but $T[h(k)]$ is already occupied.
  - We will discuss strategies of solving collisions in the next couple subsections.
  - Probability of having a collision when we pick n values from $\{w = 0, \ldots, M-1\}$:

    - the probability of no collision: $\frac{M(M-1)(M-2)\ldots(M-(n-1))}{M^n}$.

    - the probability of collision: $1 - \frac{M(M-1)(M-2)\ldots(M-(n-1))}{M^n}$.

7.2 Seperate Chaining

IDEA: every slot of the array stores a bucket containing 0 or more KVPs. We will use unsorted linked lists for buckets.

- search(k): Apply MTF-heuristic. $O(1)$.
- insert(k, v): Add (k,v) to the front of the list at $T[h(k)]$. $O(1 + \text{size of bucket } T(h(k)))$
- delete(k): Perform a search, then delete from the list. $O(1 + \text{size of bucket } T(h(k)))$.

Complexity analysis:

- the average bucket-size is $\frac{n}{M} = \alpha$, (**load factor**)

  However, this <u>DOES NOT</u> imply that the average-case cost of search and delete is $O(1 + \alpha)$.

- **Uniform Hashing Assumption:** for any key k, and for any $j \in \{0, \ldots, M-1\}$, $h(k) = j$ happens with probability $\frac{1}{M}$ independently.

  <u>Under this assumption, we can expect search and delete to have average cost $\Theta(1 + \alpha)$.</u>

7.3 Linear Probing

IDEA: Avoid the links needed for chaining by permitting only one item per slot, but allowing a key k to be in multiple slots.

- Hash function: $h(k, i) = (h(k) + i) \mod M$.
- Search and Insert: follow a probe sequence of possible positions for key k, until an empty spot is found.
- Delete: lazy delete, mark deleted spot as "deleted".

Algorithm 7.3.1 (Probe Sequence Insert)**.**

`Linear-Probing::insert(T, (k,v))`

for $j = 0; j < M; j++$ **do**
    if T[$h(k, j)$] is NIL or deleted **then**
        T[$h(k, j)$] = (k, v)
        return success
return failure to insert // need to rehash

Algorithm 7.3.2 (Probe Sequence Search)**.**

`Linear-Probing::insert(T, (k,v))`

for $j = 0; j < M; j++$ **do**
    if T[$h(k, j)$] is NIL **then**
        return item not found
    else if T[$h(k, j)$] has key k **then**
        return $T[h(k, j)]$ // ignore deleted and keep searching
return item not found

7.4 Independent Hash Functions

Two hash functions h_0, h_2 that are independent.

multiplicative method: $h_1(k) = \lfloor M(kA - \lfloor kA \rfloor) \rfloor$

- A is some floating-point number
- $kA - \lfloor kA \rfloor$ computes the fractional part of kA, which is in $[0, 1)$, then multiply with M to get floating-point number in $[0, M)$, and we round it down.
- suggests $A = \varphi = \frac{\sqrt{5}-1}{2} \approx 0.618$.

7.5 Double Hashing

IDEA: open addressing with probe sequence with hash function:

$$h(k, i) = h_0(k) + i \cdot h_1(k) \mod M \ ,$$

which h_0 uses mod method and h_1 is multiplicative using φ. So in linear probing, each time we go to next spot (index + 1), but here, the index increments by $h_1(k)$, so $h_1(k) \neq 0$ for any k.

* If $T[h_0(k)]$ is empty, we do not need to compute h_1.

7.6 Cuckoo Hashing

IDEA: Use two independent hash functions h_0, h_1 and two tables T_0, T_1. An item with key k can only be at $T_0[H_0(k)]$ or $T_1[T_1(k)]$.

Algorithm 7.6.1 (Cuckoo Hashing Insert)**.** Insert always initially puts a new item into $T_0[h_0(k)]$.

If $T_0[h_0(k)]$ is occupied: kick out the other item k', which we then attempt to re-insert into its alternate position $T_1[h_1(k')]$. This may lead to a loop of kicking out. We detect this by aborting after too many attempts. In case of failure: rehash with a larger M and new hash functions.

```
cuckoo::insert(k, v)
1.   i <- 0
2.   do at most 2n times:
3.       if Ti[hi(k)] is NIL
4.           Ti[hi(k)] <- (k, v)
5.           return success
6.       swap((k, v),Ti[hi(k)])
7.       i <- 1 - i
8.   return failure to insert // need to re-hash
```

7.7 Conclusion

Load Factor $\alpha = \frac{n}{M}$:

- $\alpha < 1$ for linear probing and double hashing, $\alpha < \frac{1}{2}$ for cuckoo hashing
- α no constraint for seperate chaining

Avg.-case costs:	*search (unsuccessful)*	*insert*	*search (successful)*
Linear Probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{1-\alpha}$
Double Hashing	$\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$
Cuckoo Hashing	1 (worst-case)	$\frac{\alpha}{(1-2\alpha)^2}$	1 (worst-case)

Summary: All operation has $O(1)$ average-case run-time if the hash-function is uniform and α is kept sufficiently small, but worst-case run-time is usually $\Theta(n)$.

8 Module 8: Range-Searching in Dictionaries for Points

Range Search: look for all itesm that fall within a given range.

- input: an interval I=(x, x'), in higher dimensions, it will be a rectangle.
- output: all KVPs which the key falls in the range.
- The time is at least $\Omega(s)$, which s is the size of points in the range.

Range searches in existing dictionary realizations:

- Unsorted list, array, hash table: $\Omega(n)$
- Sorted array: $O(\log n + s)$ time.
  - Using binary search to find i which $A[i] \approx x$ and find i' which $A[i'] \approx x'$, and report all items in $A[i+1, ..., i'-1]$, report $A[i]$ and $A[i']$ if they are in the range.
- BST: $O(\text{height} + s)$

Multi-dimensional data:

- Each item has d aspects(coordinates): $(x_0, x_1, \ldots, x_d)$
- we concentrate on $d = 2$

D-dimensional range search: given a **query rectangle** A find all points that lie within A

8.1 Quadtrees

We have n points $S = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$,

Structure:

- Root r of the quadtree is associated with region R, if R contains 0 or 1 points, then root r is a leaf that stores the point.
- Else split: Partition R into four quadrants, $R_{NE}, R_{NW}, R_{SW}, R_{SE}$.
- Convention: points on split lines belong to right/top side
- Recursively build tree T_i for points S_i in region R_i and make them children of the root.

Algorithm 8.1.1 (Quadtree Search).

```
QuadTree::RangeSearch(r ← root, A)
if R ⊆ A then report all points below R then return
else if R ∩ A is empty then return
else if r is a leaf then
    p ←point stored at r
    if p is in A then return p
    else return
for each child v of r do QuadTree::RangeSearch(v, A)
```

Analysis of QuadTree:

- The height of a quadtree can be very large for bad distribution of points
- spread factor:
$$\beta = \frac{\text{side length of } R}{\text{minimum distance between points in } S}$$
- Complexity to build initial tree: $\Theta(nh)$ worst-case
- Complexity of range search: $\Theta(nh)$ worst-case even if the answer is $\varnothing$. But in practice much faster.
- A quadtree of 1-dimensional space is a trie.
- Variation:
  - stop spliting earlier and allow up to M points in a leaf
  - store pixelated images by splitting until each region has the same color

8.2 kd-trees

Structure:

- Split the region such that (roughly) half the point are in each subtree.
- Each node of the kd-tree keeps track of a splitting line in one dimension (2D: either vertical or horizontal)
- Convention: points on split lines belong to right/top side
- Continue splitting, switching between vertical and horizontal lines.

8.3 Range Tree

Definition 8.3.1. A **Range-tree** is a tree of trees.

- Primary structure: balanced BST T that stores P and uses x–cordinates as keys. $O(n)$ space
- Associate structure: For each node v of T stores a balanced BST $T(v)$ which
  - let $P(v)$ be all points in subtree v in T
  - $T(v)$ stores $P(v)$ in a balanced BST using y–coordinates as key.
  - **Note: v is not necessarily the root of $T(v)$.**
  - Uses $O(n \log n)$ space

Definition 8.3.2. In $d-$dimensional space

- Space: $O(n(\log n)^{d-1})$
- Construction Time: $O(n(\log n)^{d})$
- Range search time: $O(s + (\log n))^{d}$

8.4 Section Conclusion

- Quadtrees:
  - Simple, works well only if points are evenly distributed
  - wastes space for higher dimensions
- kd-trees:
  - linear space
  - range search time $O(\sqrt{n} + s)$
  - inserts/deletes destory balance
  - care needed if not in general position
- range tree:
  - range search time $O(\log^2 n + s)$
  - wastes some space
  - inserts/deletes/ destory balance

9 Module 9: String Matching

9.1 Pattern Matching Definition

Problem: given a text(or haystack) T[0...n-1] and a pattern(or needle) P[0...m-1], does P occur in T?

Pattern matching algorithm consists of **guesses** and **checks**.

- A **guess or shift** is a position i which P might start at T[i] valid guesses are $0 \leq i \leq n - m$
- A **check** of a guess is a single position j with $0 \leq j < m$ where we compare $T[i + j]$ to $P[j]$. We must perform m checks of a single correct guess, but may make fewer checks of an incorrect guess.
- We represent a single run of any pattern matching algorithm by a matrix of checks, where each row represents a single check

9.2 Brute-force Algorithm

Algorithm 9.2.1. The worst case performance $\Theta((n - m + 1)m)$.

`BruteforcePM`(T[0, ...,n-1], P[0,...,m-1])
T: string of length n, P: string of length m

for $i \leftarrow 0$ to $n - m$ **do**
    for $j \leftarrow 0$ to $m - 1$ **do**
        if T[$i + j$] != P[j] **then** Break
    if $j = m$ **then** **return** i
return FAIL

9.3 Knuth-Morris-Pratt Algorithm

IDEA: Compute the failure array, then

9.4 Rabin-Karp Fingerprint Algorithm

IDEA: use hashing to eliminate guesses, compute hash function for each guess, compare with pattern hash.

- We can use the previous hash to compute the next hash.

Algorithm 9.4.1.

`Rabin-Karp`(T[0, ...,n-1], P[0,...,m-1])
$h_P \leftarrow h(P[0, ..., m - 1])$

for $i \leftarrow 0$ to $n - m$ **do** $h_T \leftarrow h(T[i..i + m - 1])$
    if $h_T = h_P$ **then**
        if strcmp($T[i..i + m - 1], P$)=0 **then return** 'found at guess i'
return FAIL

- Choose table size M at random to be huge prime
- Expected running time is $O(m+n)$
- $\Theta(mn)$ worst-case, but this is unbelievably unlikely

9.5 Boyer-Moore Algorithm

Idea: Brute-force search with three changes:

- **Reverse-order searching**
- **Bad character jumps:** build the last-occurence array L mapping Σ to integers which $L(c)$ is the largest index such that $P[i] = c$, can build this in $O(m + |\Sigma|)$.
- **Good suffix jumps:** $S[j]$ is the maximum l that
  - $P[j+1...m-1]$ is a prefix of $P[l+1...m-1]$ and $P[j] \neq P[l]$
  - $P[j-l...m-1]$ is a prefix of P and $l < 0$.
  - $l = -j$ if neither of the above is possible

9.6 Suffix Trees

Problem: want to find many patterns P within the same fixed text T?

Idea: Preprocess the text T rather than the pattern P.

Observation: P is a substring of T if and only if P is a prefix of some suffix of T.

Algorithm: store all suffixes of T in the trie as indices(begin-end), compress the trie. Text T has n characters and $n+1$ suffixes. We can build the suffix tree by inserting each suffix of T into a compressed trie. This takes time $\Theta(n^2)$. There is a way to build a suffix tree of T in $\Theta(n)$ time(beyond scope of course).

Assume we have a suffix tree of text T, to search for pattern P of length m:

- We assume that P does not have the final $\$$.
- P is the prefix of some suffix of T.

then, we search for P until one of the following occurs:

1. If search fails due to “no such child" then P is not in T
2. If we reach end of P, say at node v, then jump to leaf l in subtree of v.
3. Else we reach a leaf $l = v$ while characters of P left.

For case 2, 3, left index at l gives the shift that we should check. This takes $O(|P|)$ time.

9.7 Summary

	Brute-Force	**Karp-Rabin**	**DFA**	**Knuth-Morris-Pratt**	**Boyer-Moore**	**Suffix Tree**	**Suffix Array**
Preproc.	—	$O(m)$	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(n^2\|\Sigma\|)$ [$O(n\|\Sigma\|)$]	$O(n \log n)$ [$O(n)$]
Search time	$O(nm)$	$O(n+m)$ expected	$O(n)$	$O(n)$	$O(n)$ or better	$O(m)$	$O(m \log n)$
Extra space	—	$O(1)$	$O(m\|\Sigma\|)$	$O(m)$	$O(m+\|\Sigma\|)$	$O(n\|\Sigma\|)$	$O(n)$

10 Module 10: Data Compression

10.1 Run-Length Encoding

- Variable-length code
- Example: multiple source-text characters receive one code-word.
- The source alphabet and coded alphabet are both binary: $\{0, 1\}$
- Decoding dictionary is uniquely defined and not explicitly stored.

Example:

Encoding:

- S = 11111 11001 00000 00000 00000 00000 11111 11111 1
- C = 1 00111 01 01 000010100 0001011

Decoding:

- C = 00001101001001010
- S = 00000 00000 00011 11011
- All all-0 string of length n would be compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits.
- may cause space waste for string with small length

10.2 bzip2

IDEA: uses text transform: change input into a different text that is not necessarily shorter but that has other desirable qualities

Move-To-Front transform:

Example: GOOOOD

11 Module 11

11.1 Motivation

External memory: disk, cloud. (size unbounded, but slow)

Internal memory: registers, main memory. (fast but small)

Want to transfer memory between internal and external.

- accessing a single location in external memory automatically loads a whole block, one block access take as much time as executing 100,000 CPU instructions(need to care about the number of block accesses)
- External memory must be loaded into internal memory before processed by CPU.
- The running time is dominated by block transfers, so we can ignore the running time of CPU instructions.

11.2 External Sorting

Sort array A or n memory, assume n is large so that A is stored in blocks in external memory.

Mergesort adapts well for external memory.

2-Way Mergesort: An array, which first half and second half are both of size k are are both sorted seperately. Then we merge them using mergesort.

- keep track of two fronts of each two halves
- Runtime: $\Theta(2k) = \Theta(n)$. $n =$size of array.

d-Way Mergesort: Generalize the 2-way mergesort to d-way. Each round, merge d blocks of size k together to be a new block of size dk. The number of new blocks in the array is $n/(dk)$.

- use minheap to keep track of minimum of all fronts
- Runtime: we merge d sequences each of size k dk iterations.
  - at each iteration, we perform one deleteMin() on heap of size d which cost $\Theta(\log_2 d)$ time, and one insert() on heap of size d which cost $\Theta(\log_2 d)$ time. So total $\Theta(\log_2 d)$.
  - For each new block, we merged d sequences of size k, therefore, for each block, the time to merge was $\Theta(kd \log_2 d)$.
  - And there are $n/(dk)$ of these blocks in one round, so for one round, the, runtime is $\Theta(\frac{n}{kd} \cdot kd \log_2 d) = \Theta(n \log_2 d)$.
  - in total, there are $\log_d n$ rounds so runtime is $\Theta(n \log_2 n)$.
- In external memory, we only count block accesses. We have $\log_d n$ rounds and the time for each round is not $\cancel{\Theta(n \log_2 d)}$ but $\Theta(n)$ or better in block accesses. Then the total time becomes $\Theta(n \log_d n)$.

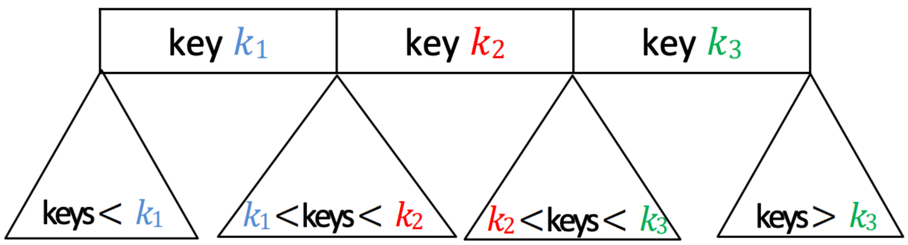
11.2.1 Mergesort external memory

We cannot merge external memory directly, we have to transfer them into internal memory first.

11.3 External Dictionary

11.3.1 2-4 trees

- Structural properties:
  - Every node is either
    - 1-node: 1 KVP and 2 subtrees(possibly empty)
    - 2-node: 2 KVPs and 3 subtrees(possible emtpy)
    - 3-node: 3 KVPs and 4 subtrees(possibly empty)
  - All empty subtrees are at the same level
- Order property: keys at any node are between the keys in the subtrees.
- need nodes that store more than one key



Algorithm 11.3.1 (2-4 tree operations)**.**

24TreeSearch$(k, v \leftarrow$ root, $p \leftarrow$ empty subtree$)$
 if v represents empty subtree
  return "not found, would be in p"
 let $T_0, k_1, \ldots, k_d, T_d$ be keys and subtrees at v, in order
 if $k \geq k_1$
  $i \leftarrow$ maximal index such that $k_i \leq k$
  if $k_i = k$
   return "at ith key in v"
  else *24TreeSearch*(k, T_i, v)
 else *24TreeSearch*(k, T_0, v)

```
24TreeInsert(k)
    v ← 24TreeSearch(k) //node where k should be
    add k and an empty subtree in key-subtree-list of v
    while v has 4 keys (overflow → node split)
        let T_0, k_1, ..., k_4, T_4 be keys and subtrees at v, in order
        if (v has no parent) create a parent of v (empty)
        p ← parent of v
        v' ← new node with keys k_1, k_2 and subtrees T_0, T_1, T_2
        v'' ← new node with key k_4 and subtrees T_3, T_4
        replace < v > by < v', k_3, v'' > in key-subtree-list of p
        v ← p //continue checking for overflow upwards
```

```
24TreeDelete(k)
    w ← 24TreeSearch(k) //node containing k
    if w is not a node with only leaf children
        v ← leaf containing predecessor or successor k' of k
        replace k by k' in w
    delete k' and an empty subtree in key-subtree-list of v
    while v has 0 keys // underflow
        if v is the root, delete it and break
        p ← parent of v
        if v has sibling u with 2 or more keys // transfer/rotate
            let u be that sibling
            if u is a right sibling // say p contains < v, k, u >
                replace key k in p by u.k_1
                remove < u.T_0, u.k_1 > from u and append < k, u.T_0 > to v
            else // symmetrical procedure if u is a left sibling
        else // merge/repeat
            if v has a right sibling
                v' ← new node with list (v.T_0, k, u.T_0, u.k_1, u.T_1)
                replace < v, k, u > by < v > in p
                v ← p
            else ... // symmetrically with left sibling
```

11.3.2 (a,b)-trees

- Structural Property:
  - each node has at least a subtrees, at most b subtrees
  - if node has k subtrees, then it stores $k-1$ KVPs
  - all empty subtrees are at the same level
  - keys in the node are between keys in the corresponding subtrees
- Height of (a,b) trees: not counting the last empty level, $O(\log_a n)$, $\Theta(\log_b n)$.

11.3.3 (B-trees)

A B-tree of order m is a $(\lceil m/2 \rceil, m)$-tree.