day / date:

What is data structure?

→ way of organizing & storing data.
→ used in designing of efficient algorithms

ABSTRACT DATA TYPE

→ List interface which can be used to model a data container with multiple items
- can be implemented using an array based or a linked list (pointer)

→ Array based implementation is
- used when we have very few insertions / deletions or a lot of direct accesses to data elements.

→ Linked list
→ Queue → add → enqueue, remove → dequeue
→ Stack

↓

add(x) → push(x)
remove() → pop()

→ Deque: It is a queue which is double ended queue you can add or remove from both sides of queue.

addFirst(x), removeFirst(), addLast(x), removeLast()

→ A set has distinct objects so whenever we are adding an element, we'll first loop on it to check if it already exists or not.

* homogenous means all elements have same type.
* heterogenous means different data type. day / date:

→ Hashing is a proper way to use set
→ if it is sorted set, we can use function compare(x,y) <0, >0 and =0
→ in sorted set, find(x) operation is called successor search.
- it returns the first value >= to the target value or null if no such value exists

ARRAY (Recap)
→ static/fixed size
→ contiguous memory
→ random access (because memory)

→ GET O(1)
→ ~~SEARCH~~ INSERT (no resize)
→ O(1) for each element
→ O(n) shifting (resize)
→ DELETE
→ O(1) for each element
→ O(n) for all element
→ SEARCH O(n)

Advantages:
→ constant time access

Disadvantages:
→ adding or removing an element near the middle of a list is cumbersome.
→ arrays cannot expand & contract
→ if it's heterogenous then we cannot direct access because each data type can have different size.

day / date:

BACKING ARRAY

→ growing & shrinking array
→ when it get's full then the size automatically gets doubled
→ dynamic array
→ you can increase/decrease by cutain factor
→ there is no such significant impact as cauplexity remais sare.

Advantages:

→ caustant time access to elmets

Disadvantages:

→ Adding a remaving of an elmet near the middle of a list is comsesare.
↳ a large masch of elmets in the array need to be shifted to make raam for newly added elmet
→ Arrays cannot expand and catract
↳ a new array of double size is cealed and old elmets are copied

day / date:

Implementation of Bacching Array.

→ In order to keep track of size of an array we have defined a class array that keeps track of it's length

```
T *a;          ← array
int length;
```

→ The size of an array is specified at time of creation

```
array(int len) {
    length = len;
    a = new T[length];
}
```

→ The elments of an array can be indexed:

```
T& operator[] (int i) {
    assert (i >= 0 && i < length);
    return a[i];
}

array<T>& operator = (array<T> &b) {
    if (a != NULL) delete [] a;
    a = b.a;
    b.a = NULL;
    length = b.length;
    return * this
}
```

day / date:

ARRAY STACK

→ An ArrayStack implements the list interface using an array a, called the backing array

→ The list element with index i is stored in a[i]

→ An integer n is used to keep track of number of elements actually stored in a.

→ The list elements are stored in a[0], ..., a[n-1] and at all times, a.length ≥ n

```
array<T> a;
int n;
int size() {
    return n;
}
```

→ Accessing and modifying the elements of an ArrayStack using get(i) and set(i,x) is straightforward. After performing any necessary bounds-checking we simply return a set, respectively, a[i]

add/remove → O(1+n-i)
- O(1) inserting/removing elmnt
- O(n-i)

day / date:

```
T get(int i){
    return a[i];        → GETTER   O(1)
}
T set(int i, T, x){
    T y = a[i];         → SETTER   O(1)
    a[i] = x;
    return y;
}
```

→ To implmt the add(i, x) operation, we first check if a is already full. If so, we call the method resize() to increase the size of a.

→ After a call to resize(), we can be sure that a.length > n

→ We now shift the elmnts a[i] ... a[n-1] right by one position to make room for x, set a[i] equal to x and increment n.

```
void add(int i, T x){
    if (n+1 > a.length) resize();
    for (int j=n; j>i; j--)
        a[j] = a[j-1];      → moving to right.
    a[i] = x;
    n++;                    → ADDING   O(n-i)
}
```

day / date:

→ Implementing the remove(i) operation is similar
→ We shift the elements a[i+1], ..., a[n-1] left by one position and decrease the value of n.
→ After doing this, we check if n is getting much smaller than a.length by checking if a.length ≥ 3n
→ If so then we call resize() to reduce size of a.

```
T ArrayStack<T>::remove(int i) {
    T x = a[i];
    for (int j=i; j<n-1; j++)
        a[j] = a[j+1]  → moving left
    n--;
    if (a.length >= 3*n) resize();
    return x;
}
```

→ The resize() method allocates new array b whose size is 2n.
→ It copies the n elements of a into the first n positions in b and then sets a to b

```
void resize() {
    array<T> b(max(2*n, 1));
    for (int i=0; i<n; i++)
        b[i] = a[i];          → RESIZE
    a = b;                        O(n)
}
```

8
6 ⟌ 50
48
2

day / date:

ARRAYQUEUE

→ ArrayStack is a poor choice for an implementation of a FIFO queue.

→ Not good choice because we must choose one end of list upon which to add elmts & then remove elmts from the other end.

→ using **modular arithmetic** we can store the queue elmts at array locations.

a[j % a.length], a[(j+1) % a.length], ..., a[(j+n-1) % a.length]. → j is the front of queue. index

Implementation of ArrayQueue.

```
→ array<T> a;
  int j;
  int n;
→ bool add (T x) {
    if (n+1 > a.length) resize();
    a[((j+n) % a.length)] = x;
    n++;
    return true;
  }
```

day / date:

```
→ T remove(){
    T x = a[j];
    j = (j+1) % a.length;
    n--;
    if (a.length >= 3*n) resize();
    return x;
  }
→ void resize(){
      array<T> b (max(1, 2*n));
      for (int k=0; k<n; k++)
          b[k] = a[(j+k) % a.length];
      a = b;
      j = 0;
  }
```

get(i) set(i) O(1)

ARRAYDEQUE

→ It allows for efficient addition and removal at both ends.

→ Implents the List interface by using the same circular array technique used to represent an ArrayQueue.

day / date:

Implementation of Array Deque.

```
→ array <T> a;
  int j;
  int n;
→ T get (int i) {
      return a[(j+i) % a.length];
    }
→ T set (int i, T x) {
      T y = a[(i+j) % a.length];
      a[(j+i) % a.length] = x;
      return y;
  }
→
```

→ for Add function, checks the given i
  ① if it's $< n/2$, it moves all elmts from a[0] to a[i-1] are place to left.
  ② otherwise it moves all elmts from a[i] to a[n-1] are place to right.

→ guarantees that add(i, x) never has to shift more than $\min\{i, n-1\}$ elmts

→ $O(1 + \min\{i, n-i\})$

day / date:

BACKING ARRAY

```
resize()
    b ← new_array(max(1, 2n))
    b[0, 1, ..., n-1] ← a[0, 1, ..., n-1]
    a ← b
```

ARRAY QUEUE

```
→ initialize()
    a ← new_array(1)
    j ← 0
    n ← 0
→ get(i)
    return a[(i+j) mod length(a)]
→ set(i, x)
    y ← a[(i+j) mod length(a)]
    a[(i+j) mod length(a)] ← x
    return y
→ remove()
    x ← a[j]
    j ← (j+1) mod length(a)
    n ← n-1
    if length(a) ≥ 3·n then resize()
    return x
```

day / date:

ARRAY DEQUE

→ add (i, x)

if n = length(a) then resize()

if i < n/2 then

j ← (j-1) mod length(a)

for k in 0, 1, 2, ..., i-1 do

a[(j+k) mod length(a)] ← a[(j+k+1) mod length(a)]

else

for k in n, n-1, n-2, ..., i+1 do

a[(j+k) mod length(a)] ← a[(j+k-1) mod length(a)]

a[(j+i) mod length(a)] ← x

n ← n+1

→ remove (i)

x ← a[(j+i) mod length(a)]

if i < n/2 then

for k in i, i-1, i-2, ..., 1 do

a[(j+k) mod length(a)] ← a[(j+k-1) mod length(a)]

j ← (j+1) mod length(a)

else

for k in i, i+1, i+2, ..., n-2 do

a[(j+k) mod length(a)] ← a[(j+k+1) mod length(a)]

n ← n-1

if length(a) ≥ 3·n then resize()

return x.

AMORTIZED ANALYSIS

day / date:

Complexity Analysis.

→ When we are trying to find complexity of a function/procedure/algorithm/program, we are not interested in the exact number of operations that are being performed. Instead, we are interested in the relation of the number of operations to the problem.

Big O Notation: Let f and g be function from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$ Q) to see if $f(n) = O(g(n))$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

→ if $f(x)$ is $O(g(n))$ this means that $f(n)$ grows asymptotically no faster than $g(n)$

→ if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$

→ if $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n)+g(n)$ is $O(h(n))$

→ The function $\log_a n$ is $O(\log_b n)$ for any +ve numbers.

* Big-O notation refers to upper bounds while big-Ω is for lower bands. day / date:

BINARY SEARCH TIME COMPLEXITY

```
while (counter < n)  ⟶ O(n)
{
    if (n % 2 == 0)
    {
        binarySearch (presatedArray, counter);   → O(log(N))
    }
    else
    {
        bruteForceSearch (presatedArray, counter);   → O(N)
    }
    counter += 1;
}
```

* always have to go to wast case so $O(N^2)$

```
int counter = 1;
while (counter < n)
{
    binarySearch (preSatedAway, counter);
    counter *= 2;                    O(log(n)²)
}
```

get / set O(1)

ARRAY STACK ⟶ resize / add / remove

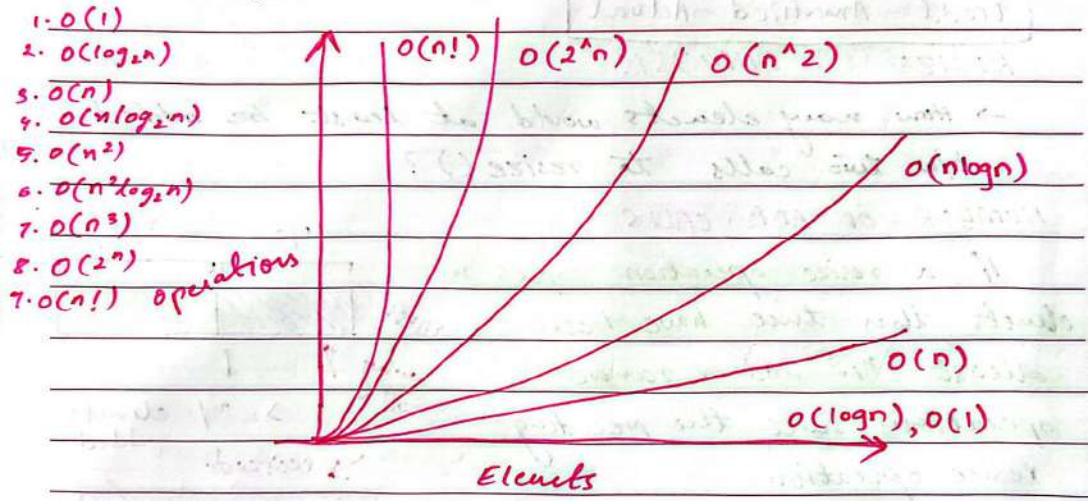
O(1 + n - i)

→ performing a series of n append operations on an initially empty dynamic array using fixed increment with each resize takes $\Omega(n^2)$ time.

day / date:

ASYMPTOTIC ANALYSIS.

→ By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time, - it's rate of growth. When we drop the constant coefficients and the less significant terms, we use asymptotic notation.



AMORTIZED ANALYSIS

→ It is used for algorithm
operation is very slow, bu
operations are faster.

→ In amortized analysis, you
measure is NOT input dep

STACK

1 push operation → O(1)
n push operation → O(n)
1 pop operation → O(1)
n pop operation → O(n)
1 multipop operation → O(n)
n multipop operation → O(n²)
Worst Case T.C = O(n²)

big-O (asymptotic upper bound) big-Θ (" " tight bound)
big-Ω (" " lower bound)

day / date:

→ If a data structure has an amortized running time of $f(n)$, then a sequence of n operations takes at most $nf(n)$ time.

Some individual operations may take more than $f(n)$ time but the average, over the entire sequence of operations, is at most $f(n)$

Credit = Amortized - Actual

← show that the total no of elem copied by resize() is at most $2n$.

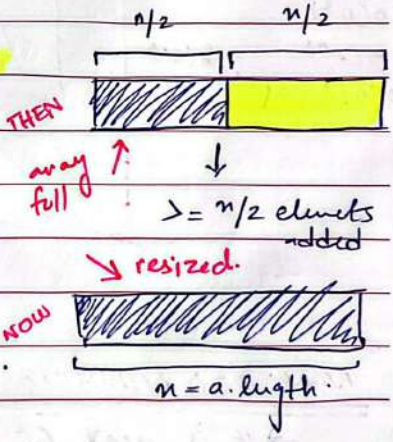
RESIZE () ANALYSIS

→ How many elements would at least be added b/w two calls to resize()?

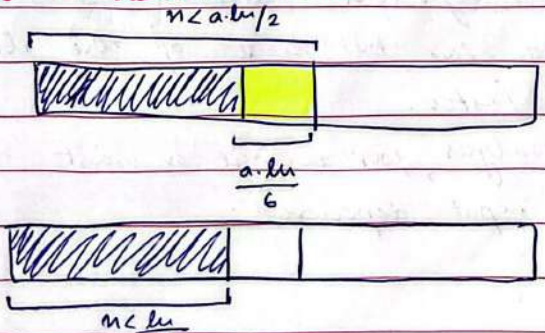
NUMBER OF ADD CALLS

If a resize operation copies n elements, then there have been atleast $n/2$ add or remove operations since the preceding resize operation.

→ $n/2$ elements were added since the last resize call.



NUMBER OF REMOVE CALLS.



* total number of items (n) copied by resize is at most $2n$, the total time spent in all calls to resize is $O(n)$

→

$N = n_1 + n_2 + n_3 + \dots$

$m \geq n_1 + n_2 + n_3 + \dots$ $\quad m \geq N/2$

$m \geq n_{1/2} + n_{2/2} + n_{3/2} + \dots$ $\quad 2m \geq N$ $\quad N =$

→ If a resize operation copie

have been at least $n/$

or remove operations s

resize operation.

Pseudocode for Multipop Operation

```
while not STACK_EMPTY(S) and
      k > 0
    pop(S)
    k = k - 1
```

1 multipop operation → $O(n)$

n multipop operation → $O(n^2)$

worst-case T.C $O(n^2)$

* Total Cost < 2 × (no of operation)

AGGREGATE METHOD. T(

→ Determine the worst-c

sequence of operation.

→ Divide this cost by number of operations in the sequence, n.

$T(n)$ = cost of all add/remove calls + cost of all resize() calls.

$$\sum_{j=0}^{\log(n-1)} 2^j = 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{\log_2(n-1)}$$

$$= a_0 (r^m - 1)/(r-1) \quad \text{(sum of geometric series)}$$

First term $= a_0 = 2^0 = 1$

Ratio b/w terms $= r = 2$

* Amortized cost of one operation $= \frac{T(n)}{n}$.

Total terms $= m = \log_2 (n-1) + 1$

$$= (2^{\log_2(n-1)+1} - 1)/(2-1)$$

$$= (2 \cdot 2^{\log_2(n-1)} - 1) = (2(n-1) - 1)$$

$$= 2n - 2 - 1 = 2n - 3$$

$$\sum_{j=0}^{\log(n-1)} 2^j = 2n - 3$$

→ Amortized cost of n add operations $T(n) = n + 2n - 3 = 3n - 3 < 3n$

→ Amortized cost of 1 " " $= T(n)/n = \frac{3n}{n} = 3 = O(1) = O(n)$

Credit = Amortized − Actual
(add previous.

ACCOUNTING METHOD

→ It involves charging
to save other, earlier
similar to taxation
focus on where each
spent rather than

A[0]: n times
A[1]: n/2 times
A[2]: n/4 times
A[3]: $< n \sum_{i=0}^{\infty} 1/2^i$
A[4]:
A[5]: $= 2n$
A[i]: $n/2^i$ times

total cost $\sum_{i=0}^{k-1} n/2^i$

→ amortized running time $\frac{2n}{n} = 2$ O(1)

Each time an m^{th} elmet is inserted, 3 units (instead of 1) are charged:

→ One unit is used to insert that elmet immediately into the table.

→ One unit is used to move that elmet first time the table is grown afterwards m is inserted.

→ One unit is donated to the corresponding elmet in the first half of the array and is used to move that elmet the first time the table is grown after m is inserted.

value	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	how often is A[i] flipped?
0	0	0	0	0	0	0	
1	0	0	0	0	0	1	
2	0	0	0	0	1	0	
3	0	0	0	0	1	1	
4	0	0	0	1	0	0	
5	0	0	0	1	0	1	
6	0	0	0	1	1	0	
7	0	0	0	1	1	1	
8	0	0	1	0	0	0	

current size of array.

$2^{n-1} \rightarrow O(\log n)$

$T(n/2) \rightarrow O(\log_2 n)$

Date: ACCOUNTING METHOD

EXAMPLE 1

let's assign amortized cost

PUSH 2

POP 0

MULTIPOP 0

Operations	Amortized cost	Actual cost	Credit
Push(A,S)	2	1	1
Push(B,S)	2	1	1+1=2
POP(S)	0	1	2-1=1
PUSH(C,S)	2	1	1+1=2
POP(S)	0	1	2-1=1
POP(S)	0	1	1-1=0
Push(D,S)	2	1	0+1=1
POP(S)	0	1	1-1=0

Total Amortised cost = (2+2+0+2+0+0+2+0)

= 8 (for 4 push op)

= 2 × 4

for n operations = 2 × n.

Amortised cost = O(n).

Credit = Amortized − Actual
(add previous.

METHOD

A[0]: n times
A[1]: n/2 times
A[2]: n/4 times

Date:

BINARY COUNTER

Value	A[3]	A[2]	A[1]	A[0]	Cost	Total cost
0	0	0	0	0	0	0
1	0	0	0	1	1	0+1=1
2	0	0	1	0	2	1+2=3
3	0	0	1	1	1	3+1=4
4	0	1	0	0	3	4+3=7
5	0	1	0	1	1	7+1=8
6	0	1	1	0	2	8+2=10
7	0	1	1	1	1	10+1=11
8	1	0	0	0	4	11+4=15

DYNAMIC TABLE

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
size	1	2	4	4	8	8	8	8	16	16	16	16	16	16	16	16
cost	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1

Wast case TC for single insert → O(n)
for n insert → O(n²)

current size of array.

i	1	2	3	4	5
ni	1	2	4	4	8
ci	1	2	3	1	5
Total cred	1	~~2~~	~~3~~	0	

after resizing we add '2' elements

1+2=3

add

Cost of i^{th} add (

$2^{\log n} \rightarrow O(\log n)$

$T(n/2) \rightarrow O(\log_2 n)$

$S_\infty = \frac{a}{1-r}$

* Multipop (s, k)
→ remove k top objects of stack s.

Cost of n add operations $= n + \sum_{j=0}^{\log_2(n-1)} 2^j$

→ Sum of first n natural numbers
$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

→ Sum of parts of 2: $2^0 + 2^1 + 2^2 + \ldots + 2^k = 2^{k+1} - 1$

→ Sum of geometric series: $a + ar^1 + ar^2 + \ldots + ar^{n-1} = a\frac{1-r^n}{1-r}$

→ Sum of squares: $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$

→ Harmonic Sum: $1 + 1/2 + 1/3 + \ldots 1/n = \ln n$.

→ Sum of odd series: $1 + 3 + 5 + \ldots + (2n-1) = n^2$

→ Sum of squares: $1^2 + 2^2 + \ldots n^2 = n^3/3$

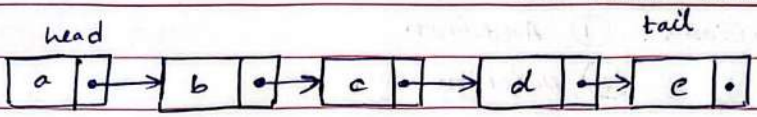
→ In a k-bit binary counter the cost for the number of bits flipped per operation ($= O(k)$)

↳ worst case running time of sequence of n increments is $O(kn)$

analyze → worst case running time $T(n)$ of n increment operations?

LINKED LIST

collection of nodes, with each node storing data & link to the next node.



addtoHead (x) —— O(1)

add to Tail (X) —— O(1) (tail ptr)

delete fromHead () —— O(1)

delete from Tail () —— O(n)

addNode (x, i) —— O(n)

deleteNode (i) —— O(n)

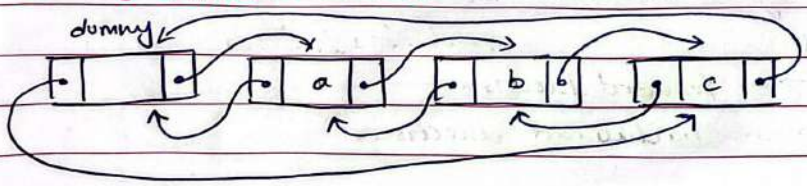
isInList (x) —— O(n)

O(n)

* a singly LL is composed of nodes
* each node has it's two components
  → a data
  → link to next node.

STACK	LINKED LIST OPERA..	TIME COMPL.
Push	addfromHead	O(1)
Pop	removefromHead	O(1)
QUEUEU		
Enqueue	addfrom Tail	O(1)
Dequeue	removefromHead	O(1)

DOUBLY LINKED LIST



→ dummy node make it circular
→ keeps track of both head & tail

time complexity when doing forward/backward traversal find(x) ⇒ O(n) Date:

Operations. (1) Addition
(2) Deletion

How To FIND A NODE?

→ either start at the head of the list (dummy → next) & walk forward.
→ or start at the end/tail of the list (dummy → prev) and walk backward.

What key issue do we ~~use~~ face in Linked List?

→ no backward traversal
→ difficult to find data.

→ A doubly linked list has the following elements.
(1) data
(2) next pointer
(3) prev pointer
→ In doubly linked list we can traverse both ways
→ in forward direction
→ in backward direction.

day / date:

SKIPLIST

→ It is a sequence of linked list

→ Each node has different number of pointers

→ each list contais sobset of linked list L_{r-1}

→ on basis of probability we decide where the elemet will go

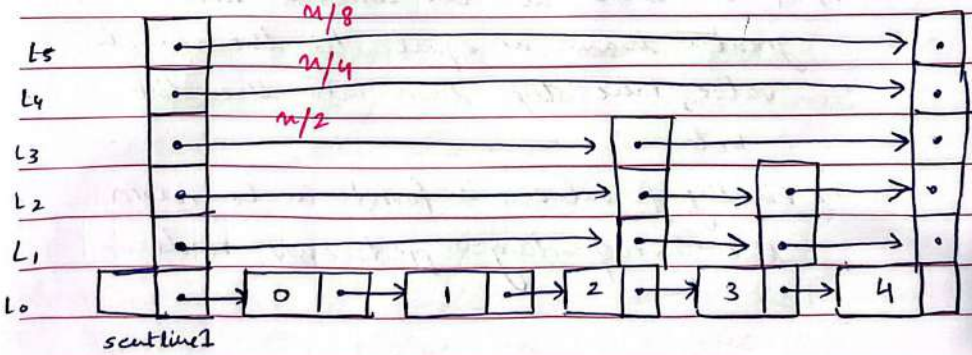
→ height of skiplist can be determind by whatever list is on top.

→ we start with the input list L_0 that contais n items and construct

→ L_1 from L_0

→ L_2 from L_1 and so on.

→ If the list L_{r-1} has n items, how many items can be expected in the list above?



No of levels in Skip list = $\log_2(n)$

$L_r = np^r$ $\quad p = \frac{1}{2}$

day / date:

→ List L_0 has height 0 and it's nodes contain {0, 1, 2, 3, 4, 5, 6}

→ List L_1 has height 1 and it's nodes {2, 3, 4, 5, 6}

→ ...

→ List L_5 has height 5 and it's only node contains {4}. (the top node has only one node)

→ There is a short path, called the search path from the sentinel in L_h to every node in L_0.

→ Steps to construct a search path for node, u is as follows:

1) Start at the top left corner of the skiplist (the sentinel in L_h) and move right unless we reach the end of the list OR

2) If the value at the current node is greater than or equal to the search value, take step down into the list below.

3) Finally, if value is found in L_0, return the corresponding predecessor node.

how many times do you think you will toss (max) for n elmts? (insertion) day / date:

SEARCH STRATEGY STEPS

→ To construct the search path for the node u in L_0, we start at the sentinel, w, in L_h.

→ Next, we examine w.next

→ If w.next contains an item that appears before u in L_0 then we set w = w.next

→ Otherwise, we move down and continue the search at the occurence of w in the list L_{h-1}.

→ We continue this way until we reach the predecessor of u in L_0.

→ to move right

to move down. ← $2\log n + O(1) = O(\log n)$

↳ in singly LL, the search path lngth is $O(n)$ (on average it will be $n/2$)

SKIPLIST INSERTION

→ Given a new elmnt x and a skiplist s, the new elmnt is placed in it's correct sorted position within the base list L_0.

→ However, in order to decide the height of the node with x, we conduct coin tosses.

→ The height of node with h is the number of coin tosses that result in the first "heads".

day / date:

ADD METHOD ()

→ Note that the add method first finds the predecessor node of x

→ The expected time complexity of this is O(logn)

→ It then adjusts pointers of adjacent nodes upto a height of h.

→ The overall time complexity O(logn)

REMOVING ELEMENT

→ Removing an element is much simpler than adding an element

→ The removal can be done as we are following search path

→ We search for x and each time the search moves downward from a node u, we check if $u \rightarrow next \rightarrow x = x$

→ If so, we splice u out of list. O(logn)

add (x)
remove (x)
find (x)
] O(logn)

each list has $\frac{n}{2^r}$ nodes

day / date:

EXPECTED NUMBER OF NODES

The expected number of nodes in a skiplist containing n elements not including occurrences of the sentinel, $2n$.

Proof: The probability that any particular element, x, is included in list L_r is $1/2^r$ so the expected number of nodes in L_r is $n/2^r$. Therefore the total expected number of nodes in all lists is

$$\sum_{r=0}^{\infty} n/2^r = n(1 + 1/2 + 1/4 + \dots) = 2n$$

($n/2^r$: in one list; $2n$: Total Nodes.)

$0 < p < 1 \quad L_r = L_0 * p$

$= p^r n \qquad L_r = n * p = np$

EXPECTED HEIGHT

The expected height of a skiplist containing n elements is at most $\log n + 2$

Proof: For each $r \in \{1, 2, 3, \dots, \infty\}$, define the indicator random variable

$$I_r = \begin{cases} 0 & \text{if } L_r \text{ is empty} \\ 1 & \text{if } L_r \text{ is non empty} \end{cases}$$

The height of skiplist is given by

$$h = \sum_{r=1}^{\infty} I_r$$

$\Rightarrow$ height of skiplist $o(\log n)$

day / date:

Note that I_r is never more than the length of L_r so

$$E[I_r] \leq E[L_r] = n/2^r$$

Therefore we have

$$E[h] = E\left[\sum_{r=1}^{\infty} I_r\right] \quad \text{(Expected height)}$$

$$= \sum_{r=1}^{\infty} E[I_r]$$

$$= \sum_{r=1}^{\lceil \log n \rceil} E[I_r] + \sum_{r=\lceil \log n+1 \rceil}^{\infty} E[I_r]$$

$$\leq \sum_{r=1}^{\log n} 1 + \sum_{\log n+1}^{\infty} n/2^r$$

$$\leq \log n + \sum_{r=0}^{\infty} 1/2^r$$

$$= \log n + 2$$

Expected Number of Nodes

The expected number of nodes in a skiplist containg n elments including all occurences of the sentinel is $2n + O(\log n)$

day / date:

Expected Search Path Length

The expected length of a search path in a skiplist is at most $2\log n + O(1)$

Q) Design and implement an SLList method second_last() that returns the second-last element of an SLList.

```
if head == NULL & head.next == NULL
        return -1
secadlast = head
last = head.next
while last.next is not null
        secadlast = secadlast.next
        last = last.next
return secadlast.value.
```

Q) Implement the List operations get(i), set(i,x), add(i,x) and remove(i) on an SLList. Each of these operations should run in $O(1+i)$.

We need to traverse the list up to the i-th element for each operation. This traversal is what dictates $O(1+i)$ time complexity because we have to visit each node leading up to ith node.

day / date:

Q) Write a DLList method is_palindrome() that returns true if list is a palindrome.

① Create doubly linked list where each node contains only one character of a string.

② Initialize two pointers left at start of list and right at end list.

③ Check if data on left node is equal to right node. If it is equal, then increment left and decrement to right till middle of list.

```
bool isPalindrome(Node* left){
    if (left == null)
        return true;
    Node *Node* right = left;
    while (right -> next != NULL)
        right = right -> next;
    while (left != right){
        if (left -> data != right -> data)
            return false;
        if (left -> next == right)
            break;
        left = left -> next;        // incrementing
        right = right -> prev;      // decrementing
    }
    return true;
}
```

day / date:

Q) Implement a method rotate(r) that "rotates" a DLList so that list item i becomes list item (i+r) mod n.

```
void rotate(int r){
    if (head==NULL || head->next== NULL || r==0){
        return;
    }
    int effectiveRotation = r % size;
    if (effectiveRotation <0){
        effectiveRotation+= size;
    }
    int stepsTohead= size - effectiveRotation
    if (effectiveRotation <stepsToNewHead){
        stepsToNewHead= effectiveRotation;
    }
    Node newTail = head;
    for (int i=1 ; i <stepsToNewhead ; i++){
        newTail =newTail->next
    }
    Node newHead= newTail-> next
    newHead->prev-> next= null
    newHead-> prev = null
    tail-> next = head
    head -> prev = tail
    tail= newTail
    head = newNode
```

day / date:

Q) Write a DLList method, absorb(l_2) that takes an argument a DLList, l_2, empties it and appends its contents in order to the receiver.

```
void absorb (DLList l2){
    if (l2.head == null) {
        return;
    }
    if (this.head == null){
        this.head = l2.head
        this.tail = l2.tail
    }else {
        this.tail.next = l2.head
        l2.head.prev = this.tail
        this.tail = l2.tail
    }
    this.size += l2.size
    // Empty l2
    l2.head = l2.tail = null
    l2.size = 0
}
```

day / date:

Q) Show that during an add(x) or remove(x) operation the expected the number of pointers in a skiplist that get changed is constant.

Q) A skiplist adds a new value to higher level if the coin came up as 'Head'. Instead of using a fair coin, we decided to use an unfair coin with 3/4 probability of 'Tail'. What is the expected height of skiplist? Will it increase or decrease as compared to fair coin?

When using a fair coin (with a probability of getting 'Head') the expected height of skiplist for n elements grows logarithmically, specifically O(log n) because each level is expected to have half nodes of the level.

If we switch to using an unfair coin with a 3/4 probability of getting 'Tail' (which implies

$n \cdot p^h \approx 1$ $n \cdot (1/4)^h \approx 1$ $\log_4(n) = h$

day / date:

a 1/4 probability of getting head) the probability of adding a newer values to higher level decreases. This results in fewer elments being promoted to higher levels, which in turn affects the expected height of skiplist.

The expected height of a skiplist with n elments using an unfair coin can be considered in terms of the expected number of levels an elment reaches. With lower probability of moving up, each level is expected to have fewer nodes than in fair coin setup, leading to potentially more levels for given number. because fewer nodes get promoted

d) Why does get_node(i) function of DLList has a factor min(i, n-i) in its time complexity O(1+min(i, n-i)) ?

DDlist can use either next or pre pointer to start traversal in forward or backward direction from dummy node depending on which is nearest to dummy node that's why min(i, n-i)

HASHING

day / date:

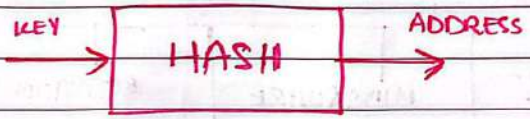
→ generating address from key

→ key is a record's property on which a record is stored in memory (called prime area)

→ A hash search is a search in which the key is modified into memory address called base address.

→ The process of converting a key to memory address is called Hashing.

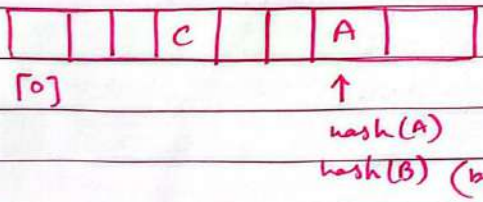
→ keys are unique ; values not necessarily unique



Collisions

→ A hash collision occurs when an key is inserted into an occupied memory location.

→ More than one key that hash to same location are called synonyms.



→ Calculation of memory address and it's test for collision is called probe. !?

day / date:

HASH FUNCTION

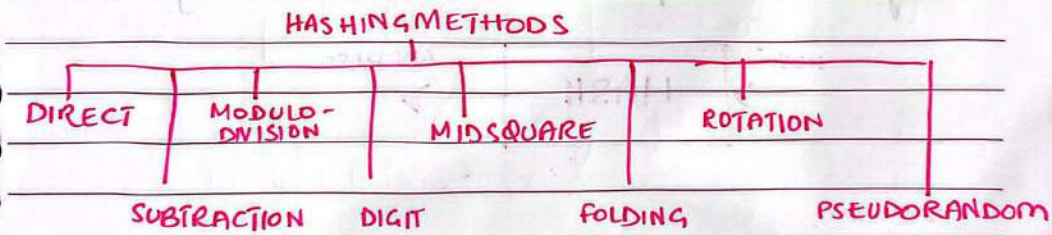
→ the hash function maps keys to indices in the hash table

→ Hash function should be such that:

① indices are uniformly distributed

② ~~range~~ of is low probability of collision / reduce collisions.

③ computationally fast. fast to compute



① DIRECT METHOD.

The key becomes the address

$h(key) = key$.

→ PERFECT HASH FUNCTION.
→ maps each item into unique slot is referred to as perfect hash
→ good hash function reduce collisions
→ outputs are uniformly distributed along hash table
→ fast to compute.

day / date:

(2) SUBTRACTION METHOD.

→ if keys do not start from 0 or 1, you get an index by subtracting a constant value from key.

→ the direct and subtraction hash functions both guarantees a search effort of one with no collisions

→ They are one to one hashing methods that is only one key hashes to each address.

(3) MODULO-DIVISION METHOD.

→ also known as division remainder or modulo-division method

→ divides the key by list/array size and uses the remainder as index.

→ works with any size but prime numbers gives less collisions.

h(key) = key % table_size.

Key should be greater than table_size to use this

* Significance of having table size that is Prime Number.

→ when keys are randomly distributed it probably won't matter what the size of your table is.

prime no → most widespread → less collisions. day / date:

→ However in the case of non random keys a hash table of prime number length will produce most widespread distribution of integer to indices. which will reduce collisions.

(4) DIGIT EXTRACTION METHOD.

→ extract certain digits and then use obtained number as address.

→ Assuming that we have an array size of 100 we can use three digits

→ Example (extracting 1st, 3rd and 4th digit)
14 56 74 ⇒ 156.

```
def digit_extraction(key):
    d3 = (key // 100) % 10
    d2 = (key // 1000) % 10
    d1 = (key // 100 000) % 10
    have_address = (d3 * 100) + (d2 * 10) + d3
```

day / date:

→ key² = then taking middle value

(5) MIDSQUARE METHOD.

The midsquare method squares the key value, and then takes at middle r bits of results, giving value in range 0 to 2r-1

(6) FOLDING.

e.g. 123456789

↳ (1) Fold shift (2) Fold boundary

(1) Fold shift

123
123 + 456 + 789 = (1)368
789
↳ discarded.

→ key is divided into parts whose size matches required address size

→ then the left and right parts are shifted and added to middle part.

(2) Fold Boundary

reversed reverse
321 + 456 + 987 = (1)764

→ the key is divided into parts as in fold shift

→ digits of left and right parts are reversed and then added to middle part.

→ overflowing digits are discarded.

day / date:

(7) MULTIPLICATION METHOD.

1) Choose a constant value A such that $0 < A < 1$
2) Multiply key value with A.
3) Extract fractional part of kA
4) Multiply the result of above step by size of hash table ie m
5) The resulting hash value is obtained by taking floor of result obtained in step 4.

$$h(k) = \text{floor}(m(kA \bmod 1))$$

(m: size; k: constant key value; A: constant value)

Example:-

$k = 12345$

$A = 0.357840$

$m = 100$ (size)

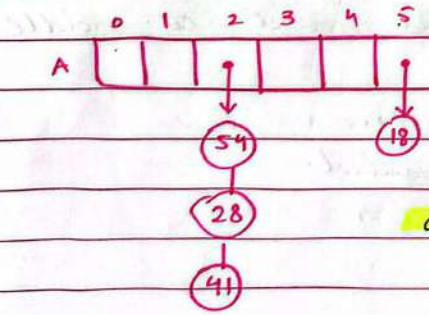
$$h(12345) = \text{floor}[100(12345 * 0.357840 \bmod 1)]$$

(100: size x; 12345 * 0.357840 mod 1: extract fraction part)

$$\text{floor}[53.48] = 53.$$

COLLISION RESOLUTION METHODS.

(1) Chained Hash Table.



$\lambda = \frac{\text{no of items}}{\text{table size}}$

→ expected length of each chain would be n/N | Load factor

day / date:

→ we want λ to be less than 1

→ ensures that map operation run in constant time

→ if $\lambda > 1$

↳ more items than size of table

↳ result in chain growing larger and larger, ruining time complexity.

↳ means more elements per index, chances of collisions.

↳ optimal load factor is 0.7 & 0.8

(2) Open Addressing.

→ in case of collisions, you look for vacant space within prime area where you can store incoming key.

→ unlike chaining, you do not introduce an additional data structure.

(i) LINEAR PROBING.

→ you perform hashing on a key k, and get an address j

→ that slot A[j] is already occupied collision.

→ start at an original hash value position and then move in a sequential manner

day / date:

through slots until we encounter first empty slot.

↳ $h(k) = j$ $A[j]$ is occupied.
↳ try looking in $A[(j+1) \bmod N]$
↳ if collision then $A[(j+2) \bmod N]$
↳ $A[(j+3) \bmod N]$ ...
↳ keep looking until you found empty slot.

→ so you search until you either,
→ find a key you are looking for
→ reach an empty slot
→ return back to index from where you started.

→ Probability of Empty Cells to be filled :- $1/M$
→ Probability of cells following cluster to be filled :- $(\text{size of cluster} + 1)/M$.

→ if you want to remove a key from hash table
→ you can pass your key in hash function
→ get index
→ and set value at that index to None.

* instead of placing None, place "available" or "tombstone" → breaks search path

day / date:

pair take steps bigger than size are
to avoid PRIMARY CLUSTERING.
- A[(j+skip) % N]

(ii) QUADRATIC PROBING.
→ A[(h(k) + f(i)) % N]
→ in linear probing, f(i) = i for i = 0, 1, 2 ...
→ in quadratic probing, f(i) = i^2 for i = 0, 1, 2, ...
→ A[h(k)+0], A[h(k)+1], A[h(k)+4], A[h(k)+9] ...
→ avoids primary clustering and instead creates SECONDARY CLUSTERING.
→ It is problematic because collision path is dependedent
→ on open addressing
→ on have address.

(3) Double Hashing.
→ The problem of secondary clustering is best addressed with double hashing
→ This method utilizes two hash functions, are for (1) accessing primary position of key, h and second function, h for resolving conflicts (2)
$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$.

day / date:

→ In case of double hashing if two keys k_1 and k_2 are hashed at same index their probing sequence can still be different as that depends upon $h_2(k)$.

→ The value returned by h_2 must never be zero or (M) because that will immediately lead to an infinite loop as probe sequence makes no progress.

ANALYSIS OF HASHING

→ it depends on the size of ~~hashing~~ hash table and how many elements it has in the table

→ inefficiency is usually evident in case of unsuccessful search.

Example:-

→ Consider linear probing to be used for collision resolution.

→ if K is not in the table, thus starting from position h(K), all consecutively occupied cells are checked.

→ the larger the cluster, the longer it takes to determine that K is not in table.

day / date:

- In the extreme case, when the table is full we have to check all cells starting from $h(k)$ and ending with $(h(k)-1) \bmod T.size$.

$h(k) \longrightarrow (h(k)-1) \bmod table\_size$.

ANALYSIS OF CHAINING

Theorem 22.1 In a hash table in which collisions are resolved by chaining, an UNSUCCESSFUL search takes average-case time $O(1+\alpha)$, under assumption of simple uniform hashing.

→ The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$ n_h

→ $T[h(k)]$ has expected length $E[n_h(k)] = \alpha$

$n_h(k)$ – $n_h(k)$ denotes number of values that hash to same location as key k, which depend on load factor α of hash table.

→ This means that expected number of elements examined in unsuccessful search is α

→ Total time required (including time for ~~complexity~~ computing $h(k)$) is $O(1+\alpha)$.

day / date:

Theorem 11.2 In a hash table in which collisions are resolved by chaining, a SUCCESFUL search takes average-case time complexity $O(1+\alpha)$ under assuption of simple uniform hashing.

→ Number of elments examed during successful search for an elment x is are more than the number of elments that appear before x in x's list.

→ To find expected number of elments examed,

① we take average, over n elments.

② this includes 1 plus expected number of elments added to x's list after x was added to list.

x_i denotes ith elment inserted into table.

k_i denotes key of x_i

We define an indicator random variable:

$$I_{ij} = \begin{cases} 1 & h(k_i) = h(k_j) \\ 0 & \text{otherwise.} \end{cases}$$

under assumption of simple uiform hashing

$$Pr\{h(k_i) = h(k_j)\} = 1/m$$

day / date:

Expected value $E[I_{ij}] = 1/m$

Expected number of checks examined in successful search.

$$E[I_{ij}] = E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} I_{ij}\right)\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} E(I_{ij})\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} 1/m\right)$$

$$= \frac{1}{n}\left(n+\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1/m\right) = 1+\frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1/m$$

$$\frac{1}{m}\sum_{j=i+1}^{n} 1 = \frac{n-i}{m}$$

$$= 1+1/n\sum_{i=1}^{n}\left(n-i/m\right)$$

$$= 1+1/mn\sum_{i=1}^{n}(n-i)$$

$$= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \Rightarrow 1+\frac{1}{nm}\left(n^2-\frac{n^2}{2}-\frac{n}{2}\right)$$

$$= 1+\frac{n^2}{2nm}-\frac{n}{2nm} \Rightarrow 1+\frac{n}{2m}-\frac{1}{2m}$$

$$\Rightarrow 1+\frac{\alpha}{2}-\frac{\alpha}{2n} \Rightarrow O\left(1+1+\frac{\alpha}{2}-\frac{\alpha}{2n}\right)$$

$$\Rightarrow O(1+\alpha)$$

day / date:

ANALYSIS OF OPEN ADDRESSING

Theorem 11.6 Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an UNSUCCESSFUL SEARCH is at most $\frac{1}{1-\alpha}$, assuming uniform hashing.

→ We define a random variable X to be number of probes made in an unsuccessful search

→ We define an event A_i for $i = 1, 2, 3, \ldots$ to be event that an ith probe occurs and it is to an occupied slot.

event $\{X >= i\}$ is intersection of events :

$A_1 \cap A_2 \cap A_3 \cap \ldots \cap A_{i-1}$

$$P_r\{X_i \geq i\} = P_r\{A_1 \cap A_2 \cap A_3 \cap \ldots \cap A_{i-1}\}.$$

$$= P_r\{A_1\} \cdot P_r\{A_2 | A_1\} \cdot P_r\{A_3 | A_1 \cap A_2\}, \ldots,$$

$$P_r\{A_{i-1} | A_1 \cap A_2 \cap \ldots \cap A_{i-2}\}$$

So $P_r\{A_1\} = n/m$

$$P_r\{A_j\} = \frac{n-j+1}{m-j+1}$$

day / date:

$$P_r\{x_i \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}$$

$$\leq \left(\frac{n}{m}\right)^{i-1}$$

$$\leq \alpha^{i-1}$$

Expected number of probes $E[x]$:

$$E[x] = \sum_{i=1}^{\infty} i \cdot P_r[X \geq i]$$

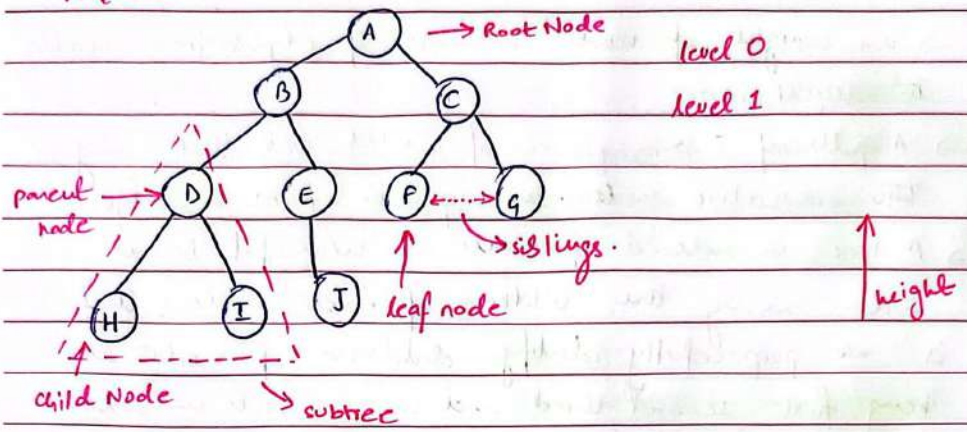
$$\leq \sum_{i=1}^{\infty} \alpha^{i-1} \Rightarrow \sum_{i=0}^{\infty} \alpha^i$$

using sum of geometric series.

$$S = \frac{a}{1-r} \Rightarrow \frac{1}{1-\alpha}$$

TREES

day / date:



→ A tree is a collection of nodes connected by directed (or undirected) edges.

→ One node is distinguished as a root.

→ every node is connected by directed edge from exactly one other node; a direction is parent → children.

→ Each node can have arbitrary number of children

→ Nodes with no children are called leaves. or external nodes. In above tree H I J F G are leaves

→ Nodes which are not leaves are called internal nodes. Internal nodes have atleast one child.

→ Nodes with same parent are called siblings

→ The depth of node is number of edges from root to node.

* Number of Edges : if there are n nodes, there will be n-1 edges.

day / date:

→ The height of node is number of edges from node to deepest leaf.

→ A path of T is sequence of nodes such that any two consecutive nodes in sequence form an edge.

→ A tree is ordered if there is meaningful linear order among the children of each node; that is we purposefully identify children of node as being first, second, third and so on. Such an order is usually visualized by arranging siblings left to right, according to their order.

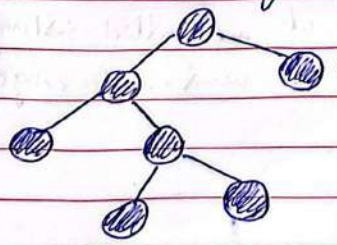
BINARY TREE

→ A binary tree is an ordered tree with following properties:

1. Every node has at most two children.
2. Each child node is labeled as being either a left child or a right child.
3. A left child precedes a right child in the order of children of node.

Full BINARY TREE

A binary tree is full if every node has 0 or 2 children

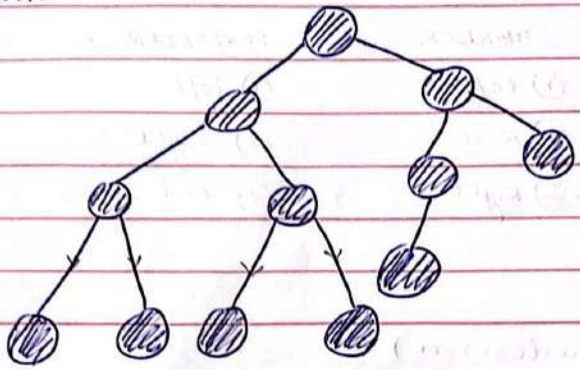


* a binary tree is improper if it has ONE CHILD.

day / date:

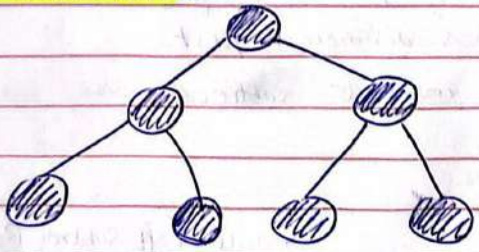
COMPLETE BINARY TREE

A binary tree is cayplete binary tree if all levels are cayplctily filled excpt possisly the last level and last level has all keys as left as possisle



PERFECT BINARY TREE

A binary tree is perfect binary tree in which all internal nodes have two children and all leaves are at same level.



day / date:

TREE TRAVERSAL

→ For traversal we can use BFS or DFS.

→ BFS is refered to as level-order traversal.

→ DFS ~~is refered~~ has three variants - Preorder, Inorder, Postorder.

PREORDER	INORDER	POSTORDER
① Root	① Left	① Left
② Left	② Root	② Right
③ Right	③ Right	③ Root.

(A) DFS.

→ visit node (root)

→ move to left child / left subtree

→ move to right child / right subtree.

PREORDER ~~(ASCENDING)~~

→ Order: Parent Left Subtree Right Subtree.

→ Root node is discarded first

→ after that you visit subtrees.

POSTORDER

→ visit subtrees

→ visit left subtree

→ visit right subtree

→ root node visited last

→ Order: Left Subtree Right Subtree Parent.

→ Time complexity for BFS and DFS is $O(n+m)$
where $n = |V|$ and $m = |E|$ $(m = n-1)$ day / date:
→ For all 4 traversals complexity is $O(n)$

INORDER (ASCENDING)

→ cannot be done on any type of tree
→ order: left subtree, Parent, Right subtree.

PSEUDO CODES

```
def inorder(root):
    if root is not null:
        inorder(root.left)
        print(root.val)
        inorder(root.right)
```

```
def preorder(root):
    if root is not null:
        print(root.val)
        preorder(root.left)
        preorder(root.right)
```

```
def postorder(root):
    if root is not null:
        preorder(root.left)
        preorder(root.right)
        print(root.val)
```

→ if you know you need to explore roots before inspecting any leaves, you pick PRE-ORDER because you encounter all roots before all of leaves.
→ if you know you need to ~~create~~ explore all leaves before any nodes you select POST-ORDER

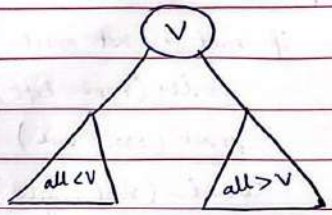
→ insertion O(log n) → deletion O(log n)
→ search O(log n)

day / date:

BINARY SEARCH TREE

* time complexity O(log n)

→ data printed in sorted format in BST
→ The value of all nodes in left subtree of V, are lesser than value of V
→ ... value of all nodes in right subtree are greater



* if you do inorder traversal on BST you get all elements in ASCENDING ORDER
→ if you want DESCENDING ORDER
① Visit Right subtree
② Root
③ Left subtree.

Q) How to find min or max value in a tree?
To find min value in the tree traverse the left children from root until you reach leaf. The min value is in last leaf node you encounter.
For maximum value traverse the right children from root until you reach leaf. The max value is in last right node you encounter.

day / date:

BST Searching

→ Base cases in this searching are

① finding target

② reaching leaf node (target not found)

Analysis

→ Since each function call takes O(1) time and you make (h+1) function calls, time required to do search would be O(h) where h is BST's height.

Worst possible
$H_{max} = n-1$, $O(n)$

Best possible
$H_{min} = \log(n+1) - 1$, $O(\log n)$

BST Insertion

→ Finding appropriate place to insert it, after insertion, the tree should still be BST.

→ search along tree until you reach an empty subtree.

Analysis

→ Total time that adding an element to BST will take is also O(h)

day / date:

BST Deletion

→ There are three cases if you find an element.

Case 1: The node to be deleted is leaf node (zero children)

Case 2: The node to be deleted has single child.

Case 3: Node to be deleted has two children.

CASE 1

→ Just break the link b/w node and it's parent

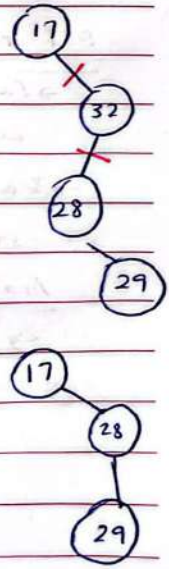
→ overwrite the leaf node with None to indicate that it has been deleted.

CASE 2

→ deleting a node which has single child would leave behind floating edges.

→ when node to be deleted only has one child

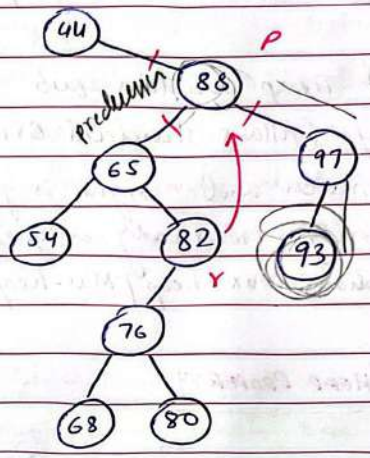
→ make that child/ subtree of deleted node a child of parent of deleted node.



day / date:

CASE 3

→ deleting the node will again leave floating edges.



* predecessor = one step left and the right
* successor = one step right and left

let's call the node to be deleted as p

→ in left subtree of p, find the largest possible node (we call this r)

→ r would be rightmost node in left subtree.

→ once you find r, overwrite the value of p with r.

for node r, choose one of the following

→ rightmost value in left subtree

→ leftmost value in right subtree.

OR

① find inorder successor of node

of inorder successor to node.

successor.

predissor can also be used.

TREAP (Tree + Heap)

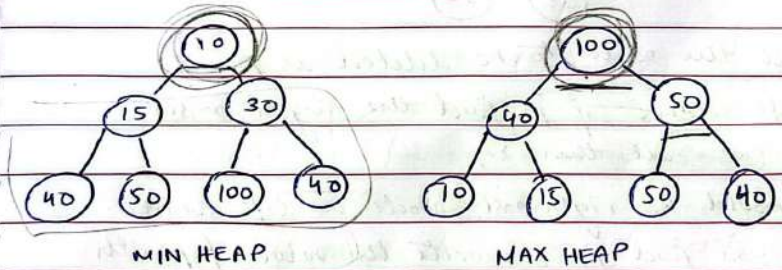
day / date:

A random binary search tree can be constructed in $O(n \log n)$ time. In random binary search tree, the find(x) operation takes $O(\log n)$ time.

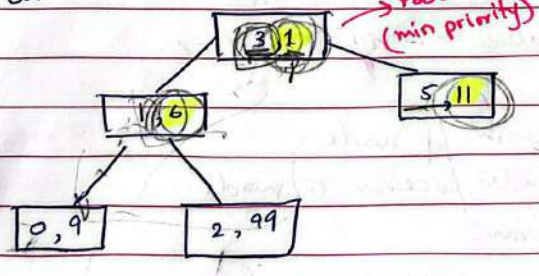
Every value of Treap maitain two values:

1. Key (follows standard BST ordering left is smaller and right is greater)
2. Priority (randomly assigned values that follows Max-heap / Min-heap property).

HEAP PROPERTY



Each node has priority smaller than that of it's two children.



* The expected length of search path is $2\log n + O(1)$ day / date:

→ The heap property tells us that node with minimum priority has to be root, r, of Treap.

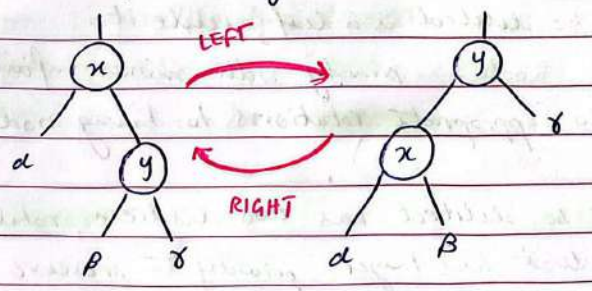
→ The BST property tells us that all nodes with keys smaller than current key are stored in left subtree rooted and all nodes with keys larger than current key are stored in right subtree.

→ The important point about priority values in a Treap is that they are unique and assigned at random.

TREE ROTATION

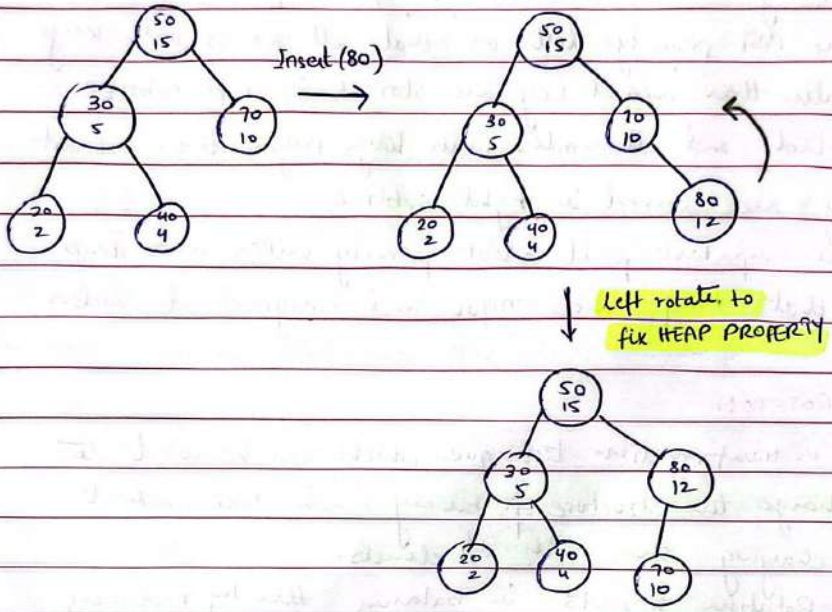
→ is transformation technique which can be used to change the structure of binary search tree without chaging the order of elments.

→ Rotation supports in balaring the by reducing depth of node by one and increasing depth of it's parent by one.



day / date:

INSERTION IN TREAP



DELETION IN TREAR

→ If node to be deleted is a leaf, delete it.

→ else replace node's priority with minus infinite ($-\infty$) and do appropriate rotations to bring node down to leaf. (max treap)

→ If node to be deleted has two children, rotate it with child that has larger priority to preserve heap ordering.

day / date:

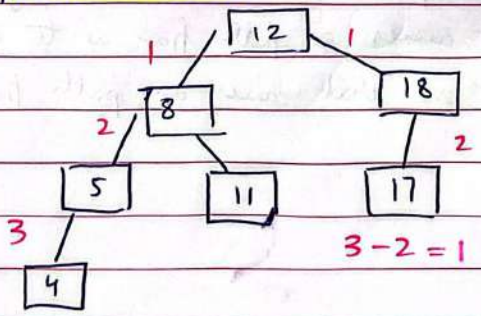
ction	O(nlogn)
(x)	O(logn)
add(x)	O(logn)
remove(x)	O(logn)

AVL TREES

→ IMPORTANCE OF HEIGHT

- → height of BST with n nodes vary from logn (balanced) to n (degenerate)
- → complexity of find, insert, delete operation depend on height of tree.
- → a binary tree is height-balanced if the difference in height of both subtrees of any node in tree is either zero or one.

→ AVL tree is self-balanced BST where difference b/w heights of left & right subtrees cannot be more than one for all nodes.



day / date:

→ We define balance factor as

— BALANCE FACTOR = $H_{right} - H_{left}$

→ for an AVL Tree balance factors of every node should be $+1, 0$ or -1.

— every subtree in AVL Tree is also an AVL Tree

→ If balance factor of any node is >1 or <-1 we must balance it to make balance factor between $+1, 0, -1$

→ An AVL Tree is rebalanced after each insertion or deletion.

— height balance property ensures that height of an AVL Tree with n nodes is $O(\log n)$.

AVL Insertion

→ you want to insert a new node w in an AVL Tree.

→ perform BST insert for w.

→ starting from w, travel up and find first unbalanced node.

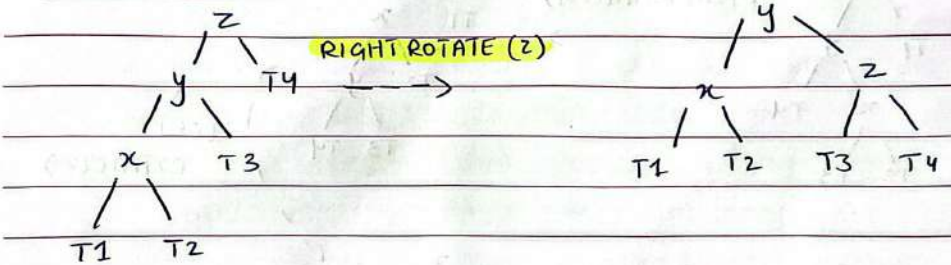
→ Let z be the first unbalanced node, y be the child of z that comes on path from w to z and x be grandchild of z that comes on path from w to z.

day / date:

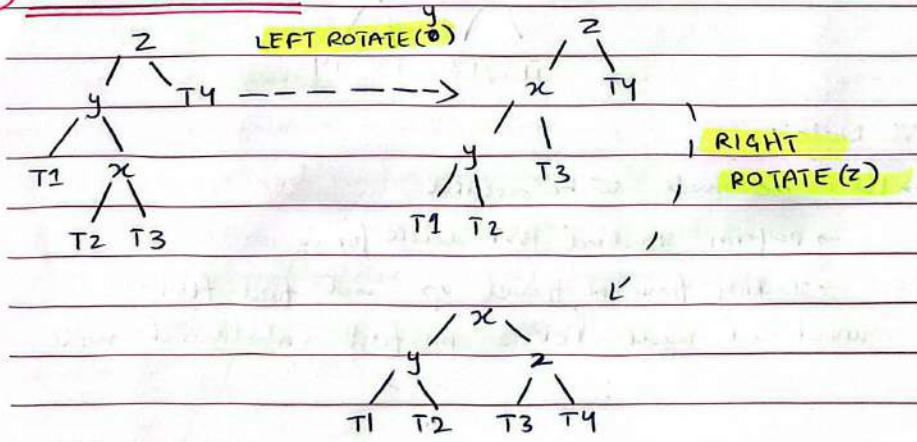
→ Rebalance the tree by performing appropriate rotations on subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.

1. Left - Left
2. Left - Right
3. Right - Right
4. Right - Left

(1) LEFT-LEFT CASE

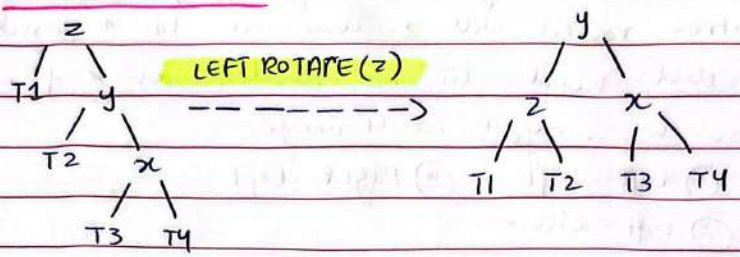


(2) LEFT-RIGHT CASE

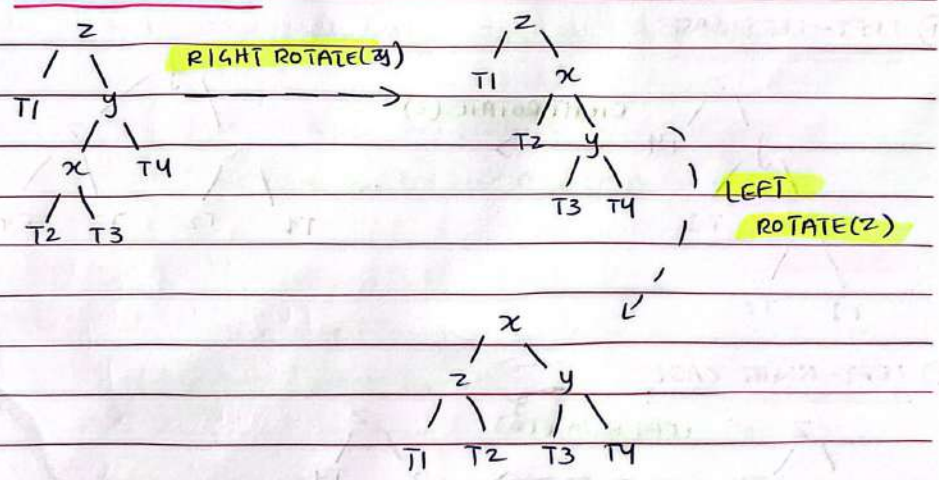


day / date:

(3) RIGHT - RIGHT CASE



(4) RIGHT - LEFT CASE



AVL Delction

→ Let w be node to be deleted

→ perform standard BST delete for w

→ starting from w, travel up and find first unbalanced node. Let z be first unbalanced node,

day / date:

y be the larger height child of z and x be the larger height child of y.

→ Rebalance the tree by performing appropriate rotations on subtree rooted with z.

Q) Prove that a binary tree having $n \geq 1$ nodes has $n-1$ edges.

To prove this,

Base Case: for $n=1$, there are zero edges. Hence statement holds true $1-1=0$.

Inductive Step: Assume statement is true for binary tree with k nodes, which has $k-1$ edges. When adding one more node to tree, we add one more edge So $k+1$ nodes will have $k+1-1=k$ edges.

Q) Prove that binary tree having $n \geq 1$ real nodes has $n+1$ external nodes.

Let n be number of real nodes in binary tree. Each real node can have 0, 1 or 2 children. Let E represent number of external nodes (leaf nodes).

In binary tree, we have $n-1$ edges and since each edge connects two nodes $2(n-1)$. However each real node connect to it's parent with one edge, $n-1$ edges.

Number of External Nodes = $2(n-1) - (n-1) + 1$

$$E = n + 1$$

B-TREE

day / date:

→ When data is too large to fit in main memory the number of disk accesses becomes important.

→ Many algorithms and data structures efficient for manipulating data in primary memory are not efficient for manipulating large data in secondary memory.

→ AVL trees are not suitable for representing huge tables residing in secondary memory.

→ The height of AVL tree increases and hence the number of disk accesses required to access particular record increases, as no of record increases.

day / date:

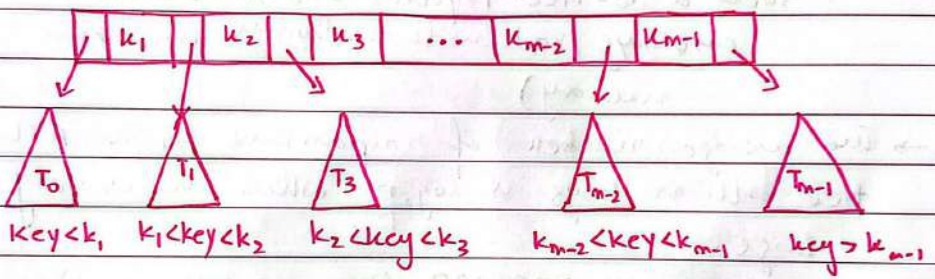
MULTIWAY TREES

→ A multiway (or m-way) search tree of order m is a tree in which

→ each node has atmost m - subtrees where the subtrees may be empty.

→ each node consists of atleast 1 and at most $m-1$ distinct keys.

→ the keys in each node are sorted



→ The keys and subtrees of a non-leaf node are arranged as:

$T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:

→ all keys in subtree T_0 are less than k_1

→ all keys in subtree T_i, $1 <= i <= m-2$ are greater than k_i but less than k_{i+1}

→ all keys in subtree T_{m-1} are greater than k_{m-1}

```
template class BTreeNode
{ public:
    BTreeNode();
    BTreeNode(const T&);
  private:
    bool leaf;
    int keyTally;
    T keys[m-1];
    BTreeNode *pointer[m]
    friend BTree;
```

B-TREE HEIGHT

→ let $d = \lceil m/2 \rceil$ ... of a B-tree is:

$h_{max} = \lfloor \log_d ($

→ if $m = 300$

$h \simeq 4.$

→ thus in wasl case ... such a B-tree requires 3 disk accesses (assuming root node is always in main memory)

→ The average number of comparisons for an AVL tree with n keys is $\log_2 n$, where n is very large.

→ if $n = 16,000,000$ the average number of comparisons is 24.

→ thus in avg case finding a key in such an AVL tree requires 24 disk accesses.

B-TREE

→ There are certain conditions that must be true for a B-tree.

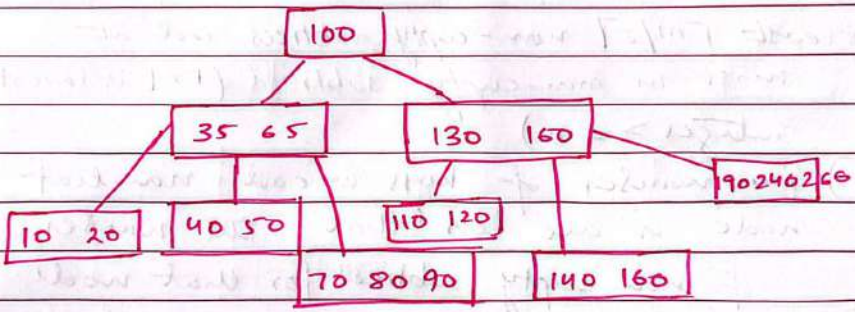
day / date:

① the hight of tree must remain as minimum as possible

② Above the leaves of tree, there shald^ not be any empty subtrees.

③ The leaves of tree must occur at the same level

④ all nodes must have same minimum number of children except leaf nodes.



WHY USE B TREES?

→ Reduces the number of reads made on disk

→ B trees can be easily optimised to adjust it's size (that is no of nodes) according to disk size

→ Specially designed technique for handling bulky amount of data.

Order → max no of children a node can have. day / date:

→ useful for databases and file systems

→ A B-tree of order m (or branching factor m) where $m > 2$ is either an empty tree or multiway search tree with following properties.

1. The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
2. Each non-leaf node, other than root, has atleast $\lceil m/2 \rceil$ non-empty subtrees and at most m non-empty subtrees ($\lceil x \rceil$ is lowest integer $>= x$)
3. The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
4. All leaf nodes are at same level; tree is perfectly balanced

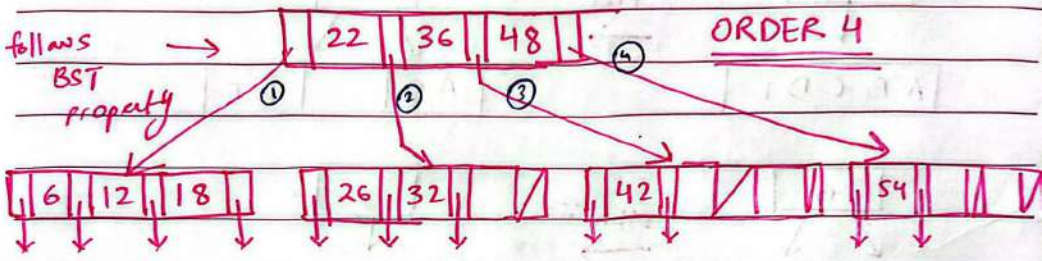
	Root Node	Non-Root Node
Min No of Keys	1	$\lceil m/2 \rceil - 1$
Min No of Non Empty Subtree	2	$\lceil m/2 \rceil$
Max No of Keys	$m-1$	$m-1$
Max No of Non Empty Subtree	m	m

	Children	Root	Intermediate Node
*Block Ptr = p	Max	p	p
	Min	2	$\lceil p/2 \rceil$

day / date:

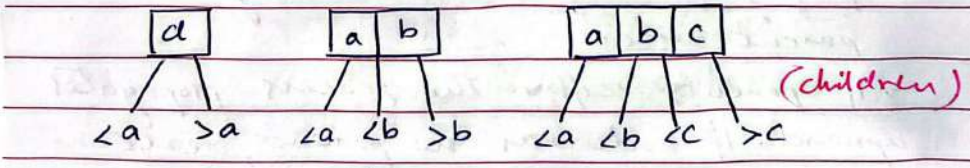
→ if p is the order
→ we can have p−1 keys → points to secondary memory
→ " " record pointer = p−1
→ where Block Pointer = p → points to children; keys = record ptr
→ keys inserted in sorted order

Max = 4−1 = 3 keys



2-3-4 OR 2-4 TREE

→ 2-4 (or 2-3-4 tree) is a B-Tree of order 4.
→ This type of tree can contain nodes of 3 types.

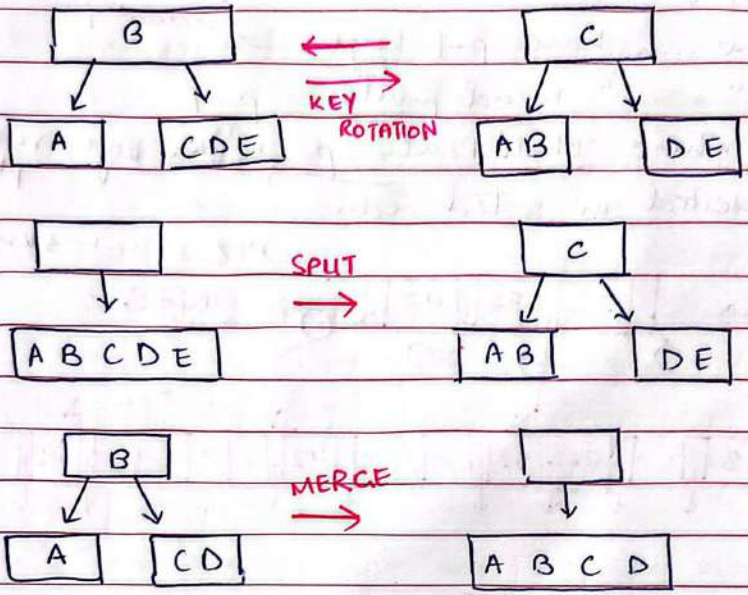


(a) 2 NODE (b) 3 NODE (c) 4 NODE

* size property: every internal node has at most four children

day / date:

OPERATORS



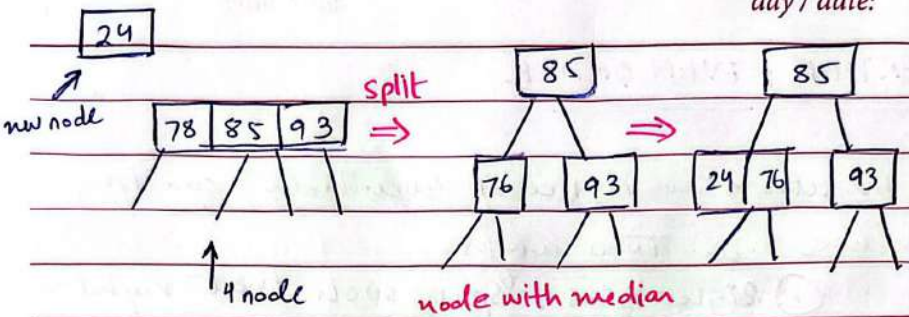
INSERTION

→ A new node is inserted into leaf node

→ if this insertion results in overflow of that leaf nodes then split the node into two, propagate the middle value key to parent node.

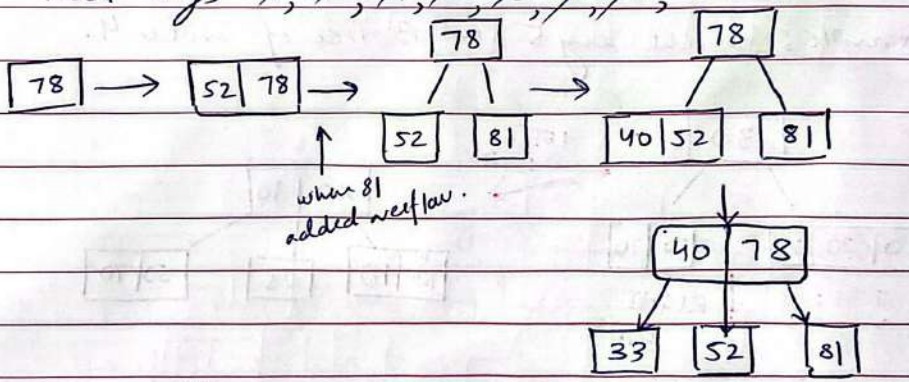
→ If parents overflow the process propagates upward. If node has no parent, create a new root node.

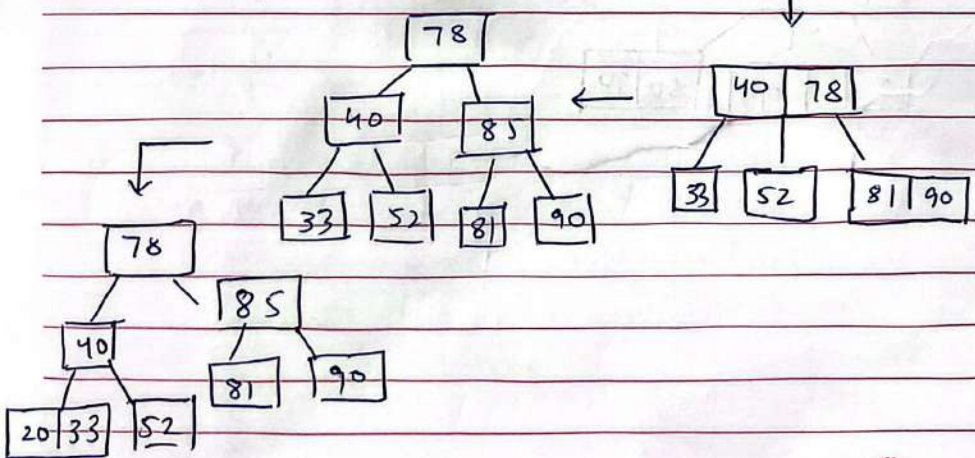
day / date:



INSERTION: ODD ORDER (order = 3) (keys = 2)

Insert keys 78, 52, 81, 40, 33, 90, 85, 20 and 38





day / date:

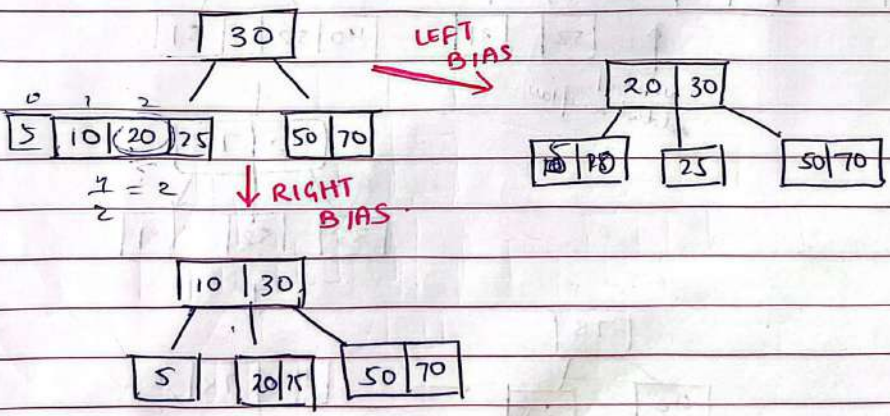
INSERTION : EVEN ORDER

→ Insertion in B-tree of even order can be done in two ways

(1) Right Bias : Split such that right subtree has more keys than the left

(2) Left Bias: Split such that left subtree has more keys than right.

Example: Insert key 5 in B-tree of order 4.



day / date:

A NODE

ode

CASE ① → with more than min keys:
→ delete the key, rest is good.

CASE ② → with min key
→ try to borrow a key from left or right sibling if they have more than min child. This would happen via rotation

* more than min no of keys.

CASE ③ → else
→ merge with sibling using parent
→ if parent is also min node, apply merging at parents level too.

* if neither left or right sibling will be having more than min no of keys.

* merge parent node b/w both left and right.

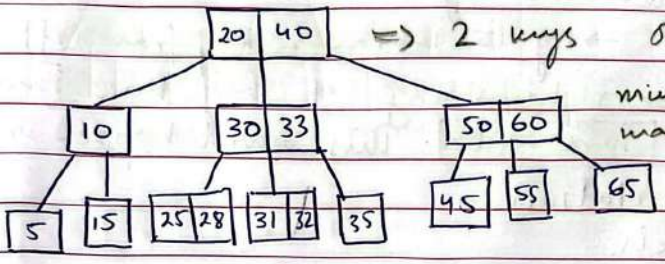
② → Internal Node.
→ if node has more than min child
→ delete key rest if good
→ replace by inorder predecessor / inorder successor if left / right child has more than min child.
→ else merge left and right child.
→ if node to be deted has less than min nodes then merge this node with sibling

day / date:

DELETING A LEAF NODE

CASE (1)

The deletion of key does not violate property of minimum no of keys a node should hold.

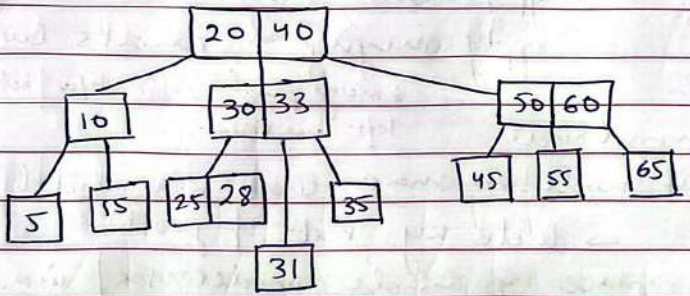


=> 2 keys Order = 3

min keys = 1

max keys = 2

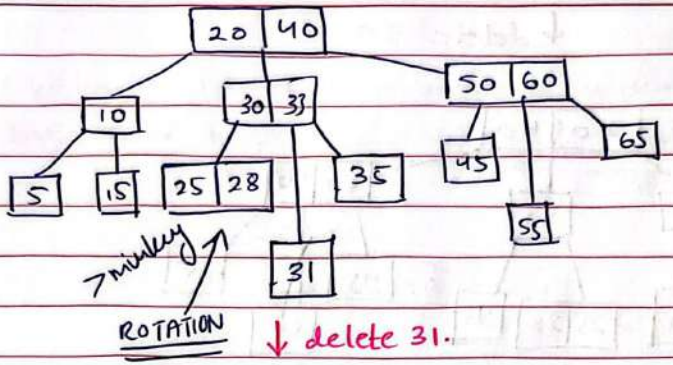
↓ delete 32.

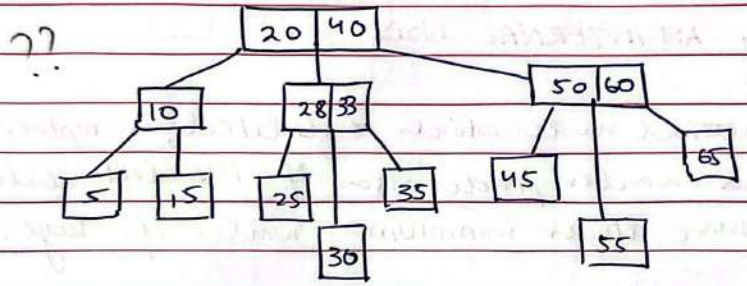


CASE (2)

The deletion of key violates property of minimum number of keys a node should hold. In this case, we borrow a key from it's immediate neighbouring sibling node in order of left to right.

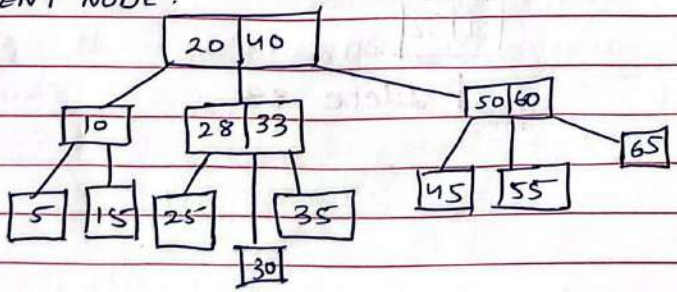
day / date:





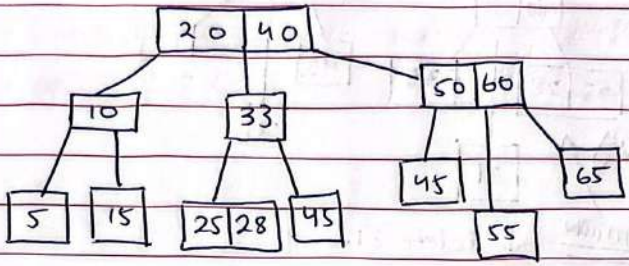
CASE (3)

If both immediate sibling nodes already have minimum number of keys, then merge the node with either left sibling node or right sibling node. THIS MERGING IS DONE THROUGH PARENT NODE.



day / date:

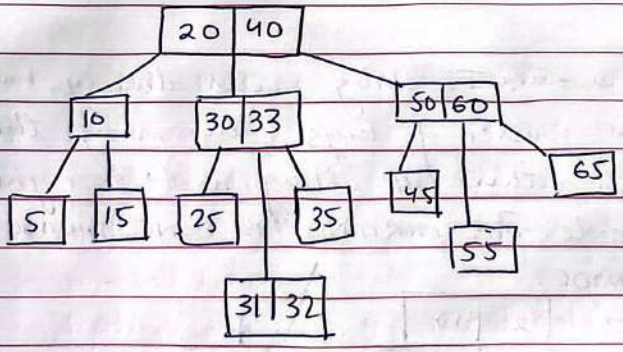
↓ delete 30



DELETING AN INTERNAL NODE

CASE ①

The internal node which is deleted, is replaced by an inorder predecessor if the left child has more than minimum number of keys.

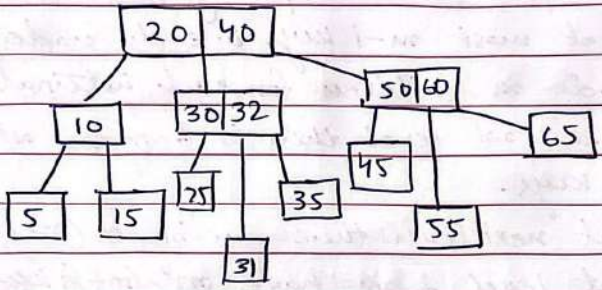


↓ delete 33

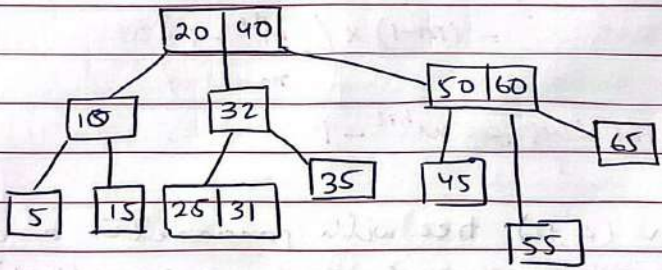
day / date:

CASE (3)

If either child has exactly a minimum number of keys then merge left & right children.

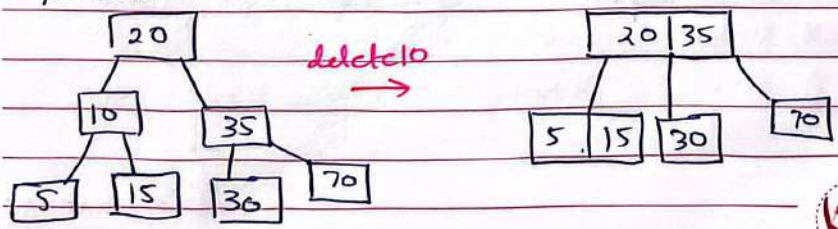


↓ delete 30



CASE (4)

In this case, the height of tree shrinks. If sibling also has only a minimum no of keys then merge the node with sibling along with parent.



day / date:

→ A B-Tree of order 'd' would be an (a,b) tree with $a = \text{ceil}(d/2)$ and $b = d$.

→ what is the maximum number of keys in a B-tree of order m of height h?

→ at most m-1 keys in each node and m children for each internal node.

→ at root level there is 1 node with maximum m-1 keys.

→ at next level there can be $m(m-1)$ keys.

→ at level 2 we have $m^2(m-1)$ keys.

$$\text{Total keys} = (m-1) \times (1 + m + m^2 + \dots m^h)$$

$$= (m-1) \times \left(\frac{m^{h+1} - 1}{m-1} \right)$$

$$\text{Total Keys} = m^{h+1} - 1$$

→ An (a,b) tree with parameter a and b are integer such that $2 \leq a \leq (b+1)/2$

size property: each internal node has atleast a children and has at most b children

depth property: all external nodes have same depth.

day / date:

PERFORMANCE OF (2,4) TREE

→ the height of a (2,4) tree storing n entries is $O(\log n)$

→ a split, transfer or fusion operation takes $O(1)$ time.

→ a search, insertion or removal of an entry visits $O(\log n)$ nodes.

B-TREE vs BST / Binary Search.

→ $O(\log_d n) \Rightarrow O(\log n / \log d)$

Min Children = $\lceil \frac{m}{2} \rceil$

Max Children = P

Min Keys = $\lceil \frac{m}{2} \rceil - 1$

Max Keys = $p-1$

COMPLEXITY OF FIND, INSERTION & DELETION.

① FIND

→ by ~~search~~ size property of 2-4 Trees, we have at most 4 nodes at depth 1, 4^2 at depth 2 and 4^h nodes at depth h.

→ by depth property of 2-4 trees, all external nodes have same depth. This implies that we have atleast 2 nodes at depth 1, 2^2 nodes at depth 2 and 2^h nodes at depth h.

day / date:

→ Since an n-way multiway trees has n+1 external nodes we have

$$2^h \leq n+1 \leq 4^h$$

→ taking $\log_2$ both sides.

$$h \leq \log_2(n+1) \leq 2h$$

→ This implies that height of 2-4 is $O(\log(n))$
→ Since a 2-4 Tree is balanced and complete the find function will take $O(\log n)$ time.

② INSERT

→ The maximum no. of children of any node in 2-4 tree (dmax) is at most 4.
→ The original search for placemet of new key k uses $O(1)$ time at each level.
→ This implies $O(\log(n))$ time overall since the height of tree is $O(\log(n))$
→ The modification to single node to insert a new key and child can be implemeted to run in $O(1)$ time, as can a single split operation.
→ The number of cascading split operations is bounded by height of tree and so that phase of insertion process also runs in $O(\log(n))$ time.

day / date:

→ Delete

→ Removing an item from a 2-4 tree preserves depth property but it might violate the size property.

→ To remedy an underflow, we check wether an immediate sibling of w is a 3-node or a 4-node.

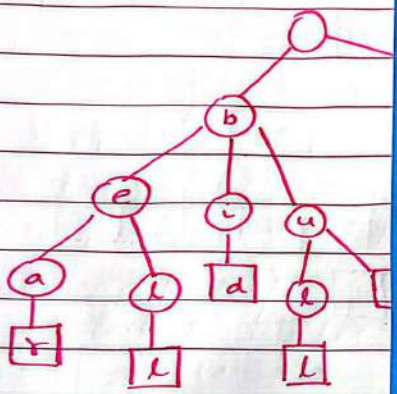
→ In wast case, the underflow propagates all the way to the root node which is then deleted.

→ Since the total number of fusion operations are bounded by height of 2-4 tree that is $O(\log n)$ the total time to perform deletion in (2,4) tree is $O(\log n)$

TRIES → ordered tree to maila day / date: a set of strings

→ A trie is tree-based data structure of storing strings that support fast pattern matching

→ Primarily support pattern matching and string matching.




```
struct TrieNode
{
  char data;
  TrieNode* child[26];
  int wc;
}
```

Standard trie for strings { bear, bell, bid, bull, buy, sell, stock, stop }

→ Let S be a set of s strings from alphaset Σ such that no string in S is prefix of another string. A standard trie for S is an ordered tree T with following properties.

① Each node of T, except the root, is labelled with character of Σ.

day / date:

(2) The children of an internal node of T have distinct labels.

(3) T has S leaves each associated with a string of S, such that the concatenation of labels of nodes on a path from root to leaf v of T yields the string of S associated with v.

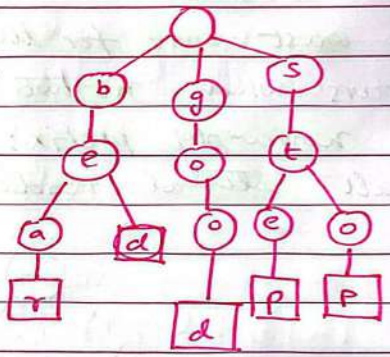
EXAMPLE

Input = {bed, bear, stip, stop, good}

Σ : {a, b, d, e, g, o, p, r, s, t}

$|\Sigma| = 10$

$n = 19$ (sum of lngth of strings)



Total strings = 5

Max lngth of string = 4

Height = 4

Total internal nodes = 10

Total leaves = 5

Total Nodes = 15 + 1 = 16

g p b s day / date:

PROPERTIES :-

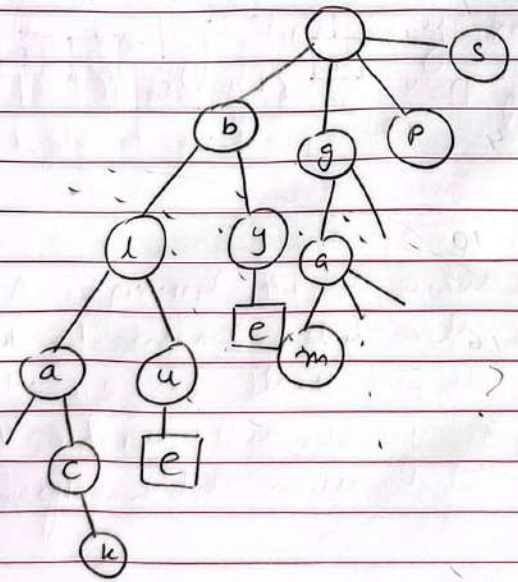
(1) the height of T is equal to the length of largest string in S

(2) every internal node of T has at most $|\Sigma|$ children.

(3) T has s leaves

(4) The no of nodes of T is at most $n+1$

?? The worst case for the number of nodes of trie occurs when no two strings share a common nonempty prefix; that is except the root all internal nodes have one child.



Time Complexity of DELETION $\Rightarrow O(2m)$ day / date:

FINDING A STRING. $\Rightarrow O(m)$

→ given a search string x of length x
→ start from root take the path based on characters in x.
→ if path terminates at a leaf node, the string x exists in the trie
→ if path terminates at an internal node, the string x does not exist in the trie.

DELETING A KEY

→ during delete operation we delete the key in botton up maner using recursion. The following are possible conditions when deleting a key from trie.
→ key may not be ~~true~~ there in trie. Delete operation should not modify trie.
→ key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all nodes.
→ key is prefix of another long key in trie. Unmark the leaf node.

→ key present in trie having atleast one other key as prefix key. Delete node from end of key until first leaf node of longest prefix key.

SEARCH O(m)

→ assumes no string is a prefix of another string.

→ running time of search of string of length m is $O(m \cdot |\Sigma|)$

→ we visit m+1 nodes at most

→ we spend at most $|\Sigma|$ time at each node.

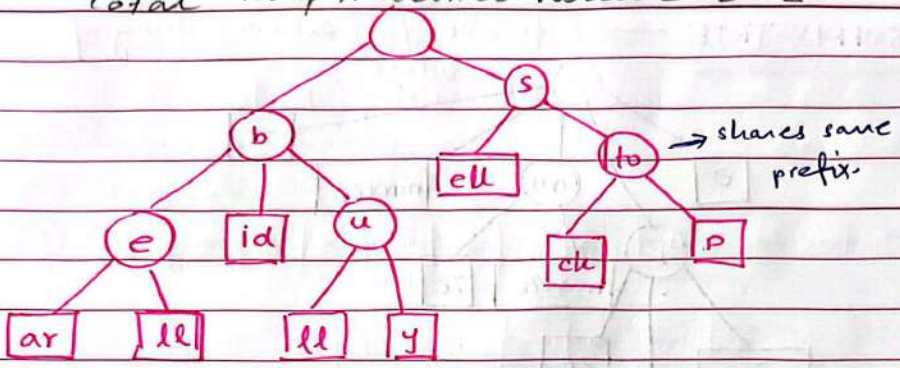
→ the running time may be reduced to $O(\log(|\Sigma|)) \approx O(1)$ by mapping character to hash table at each node.

COMPRESSED TRIE

A compressed trie is similar to standard trie but it ensures that each internal node in the trie has atleast two children.

It enforces this rule by compressing chain of single-child nodes into individual edges.

No of Leaf Node = Total no of words = L
Total No of internal Nodes = L - 1

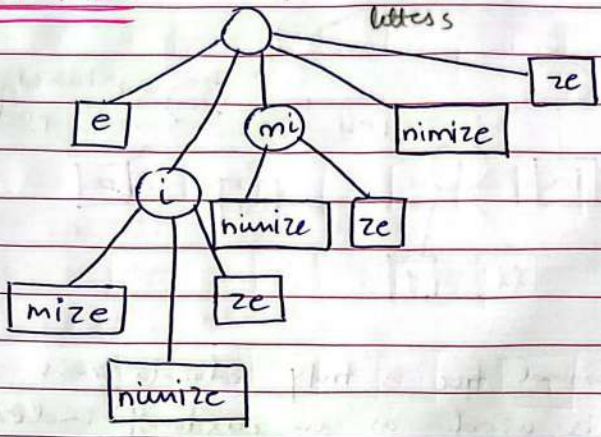


→ a compressed trie is truly advantageous only when it is used as an auxiliary index structure over collection of strings already stored in primary structure and is not required to actually store all characters of strings in collection.

	0	1	2	3	4
S[0] =	s	e	e		
S[1] =	b	e	a	r	
S[2] =	s	e	l	l	
S[3] =	s	t	o	c	k
S[4] =	b	u	l	l	
S[5] =	b	u	y		
S[6] =	b	i	d		
S[7] =	h	e	a	r	
S[8] =	b	e	l	l	

day / date:

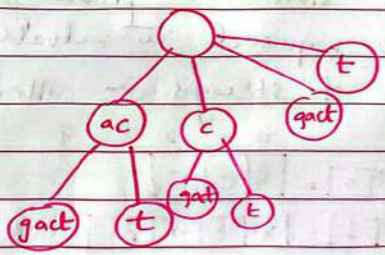
SUFFIX TRIE → preferable for continuous string of letters



→ a suffix trie is a compressed trie for all the suffixes of the text.

e.g. Text : acgact

Suffixes : acgact
cgact
gact
act
ct
t



day / date:

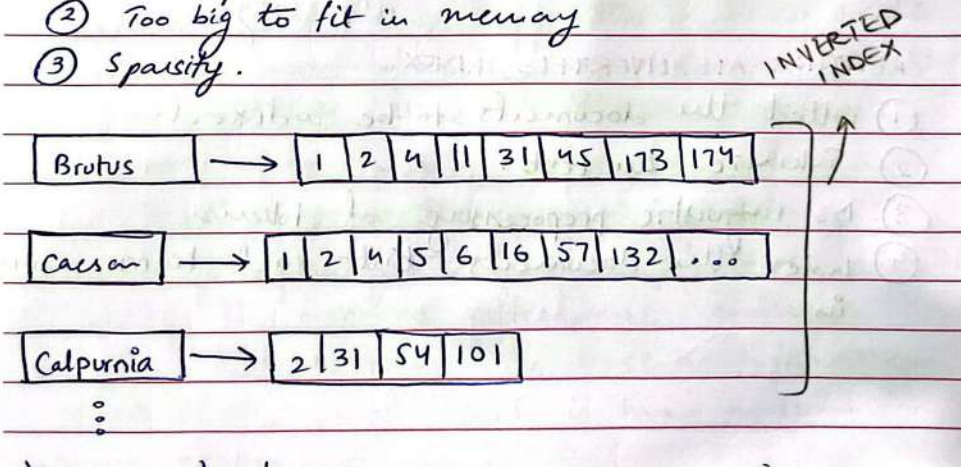
INVERTED INDEX

(1) The sun is shining
(2) The weather is sweet
(3) The sun is shining and the weather is sweet

{'the': 5, 'shining': 2, 'weather': 6, 'sun': 3, 'is': 1, 'sweet': 4, 'and': 0}

DRAWBACKS

(1) size of matrix
(2) Too big to fit in memory
(3) Sparsity.

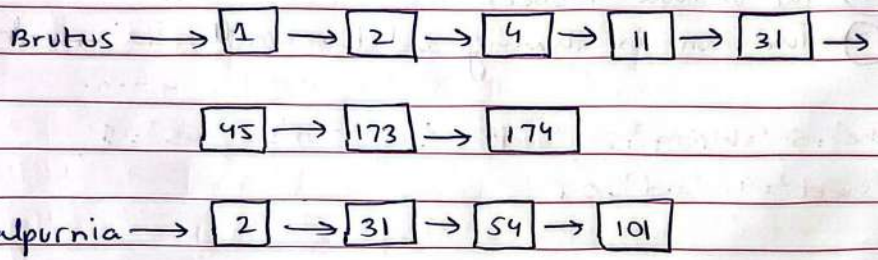


*if the list lyths are x and y then the merge takes O(x+y)

day / date:

PROCESSING QUERIES.

→ retrieve documents containg 'Bruts AND Calpurnia'



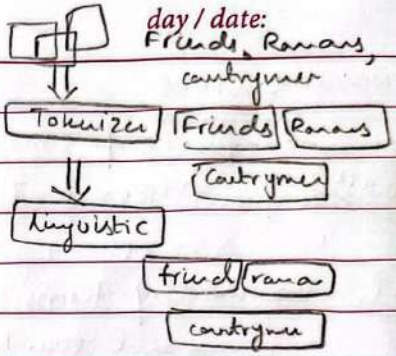
CREATING AN INVERTED INDEX

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Do linguistic prepocessing of tokens.
4. Index the documents that each term occurs in.

the, a, to, of

PREPROCESSING

→ removing stop words
→ removing hyphens
→ abbreviations
→ maintaing synonyms.
→ stemming
→ lemmatization



TEXT PREPROCESSING

authaize, authaizations

→ STEMMING usually refess to rude heuristic procesn that chops off the ends of wards in the hope of achieving this goal correctly most of the time.

→ LEMMATIZATION usually refess to doing things properly with the use of vocabulary and maphological analysis of wads, normally aiming to remove inflectional endings only and to return the base a dictionary form of a ward which is knew as lema.

→ NORMALIZATION

→ map text and query term to same form
→ you want U.S.A and USA to match.

TF-IDF

day / date:

→ vector representation doesn't consider the ordering of words in a document

→ how many times a word occur?

→ John is quicker than Mary and Mary is quicker than John have same vectors.

TERM FREQUENCY

→ The term frequency $tf_{t,d}$ of term t in document d is defined as the number of times that t occurs in d.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

LOG FREQUENCY WEIGHTING

→ the log frequency weighting of term t in d is

$$w_{t,d} = \begin{cases} 1 + \log_{10} tf_{t,d} & \text{, if } tf_{t,d} > 0 \\ 0 & \text{, otherwise} \end{cases}$$

→ score for a document-query pair: sum over terms t in both q and d:

$$Score = \sum_{t \in q \cap d} (1 + \log tf_{t,d})$$

→ the score is 0 if none of query terms is present in document.

day / date:

DOCUMENT FREQUENCY

→ Rare terms are more informative than frequent terms
  → recall stop words. the, to, if
→ Consider a term in the query that is rare in the collection?
→ frequent terms are less informative than rare terms / weighted less
→ In the extreme case take a word like 'the' that occurs in EVERY document.
→ Terms that occur in only a few documents are weighted more.

idf weight

→ df_t is the document frequency of t: the number of documents that contain t
→ df_t is the inverse measure of the informativeness of t.
→ $df_t \leq N$

$$idf_i = \log_{10}\left(N / df_t\right)$$

N → no of sentences
df_t → no of sentences containing word.

→ we use $\log(N/df_t)$ to "dampen" the effect of idf

calculates how common a word is across documents.

day / date:

TF+IDF (term frequency-inverse document frequency) is a statistical measure that evaluates how relevant a word is to document in a collection of documents.

$$tf\text{-}idf_{i,j} = tf_{i,j} \times idf_i$$

$$tf\text{-}idf_{i,j} = (1 + \log tf_{t,d}) \times \log_{10}(N / df_t)$$

↑ hyphen not a "minus" sign.

→ increases with the number of occurrences within a document

→ increases with rarity of term in collection.

* TF-IDF is highest when t occurs many times within small no of documents.
* lower when the term occurs fewer times in document or occurs in many documents
* lowest when term occurs in virtually all documents.