DESIGN & ANALYSIS of ALGORITHMS

unit-3

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

DECREASE & CONQUER

- reduce problem to smaller instance
- solve smaller instance
- extend solution to the solution of problem
- no 2 recursion trees like in divide and conquer

- bottom-up: iterative
- top-down: recursive

- inductive approach / incremental approach

Variations

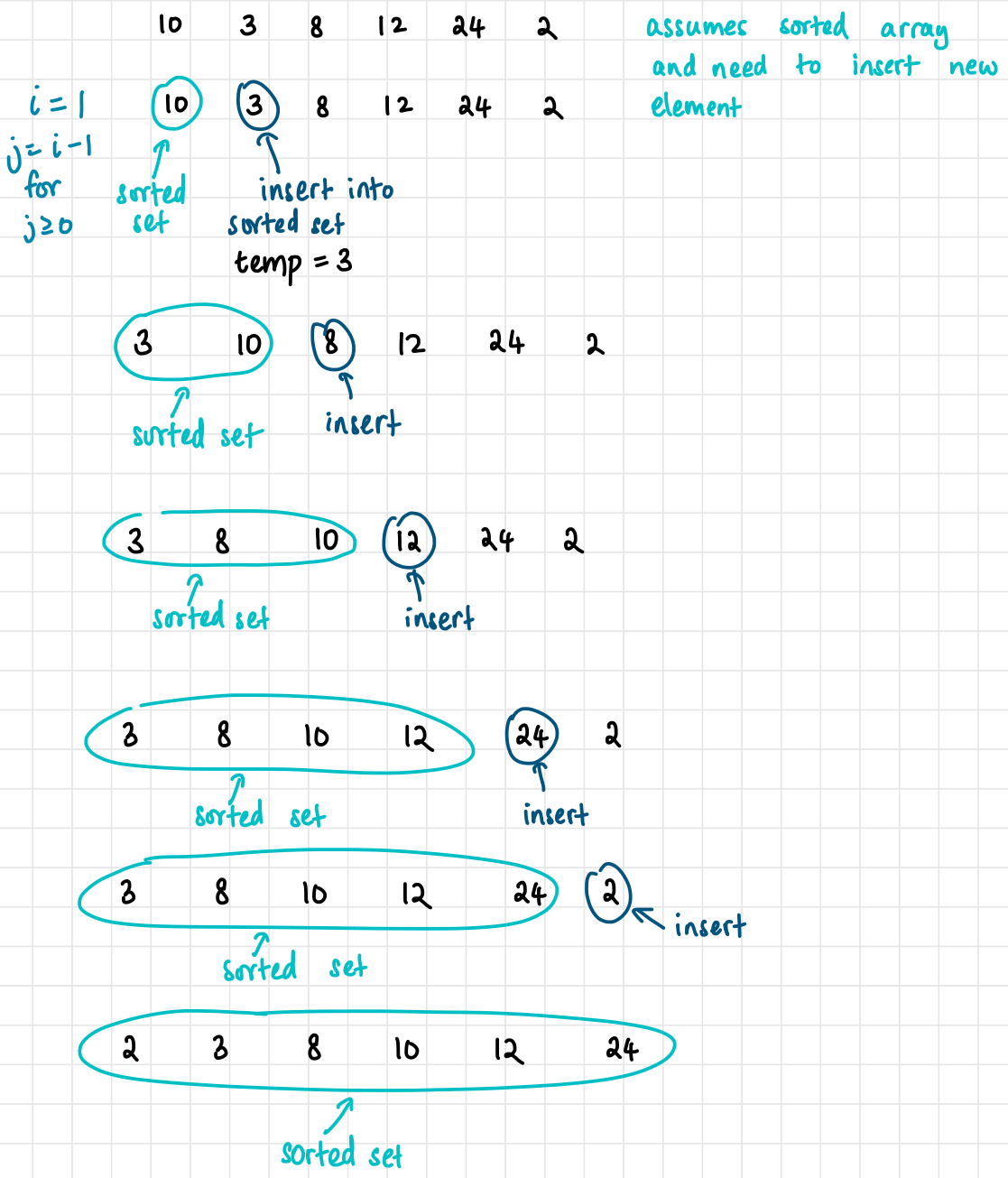
1) Decrease by constant
   eg: factorial — reduced by 1
   exponentiation — $a^n = a^{n-1} \times a$

2) Decreased by constant factor
   eg: exponentiation — $a^n = (a^{n/2})^2$

3) Variable size decrease
   eg: Euclid's gcd algorithm
   $gcd(m,n) = gcd(n, m \bmod n)$

INSERTION SORT

- decrease and conquer



IMPLEMENTATION IN C

```c
void insertion_sort(int *A, int n) {
    int insert, i, j;
    for (i = 1; i < n; ++i) {
        insert = A[i];
        for (j = i - 1; j >= 0 && A[j] > insert; --j) {
            // Shift right
            A[j + 1] = A[j];
        }
        A[j + 1] = insert;
    }
}
```

for sorted / near sorted elements: ~0 ms

Execution time:

10^4 elements : ~60 ms ↘ ×100

10^5 elements: ~6000 ms ↘ ×100

10^6 elements: ~ 600 s ← 10 mins

for merge sort:

10^6 elements: ~200 ms

Time complexity

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$$

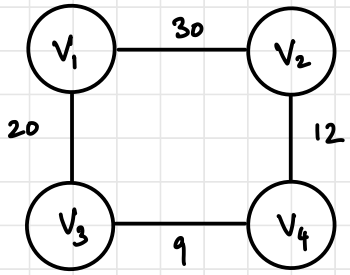
$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$$

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

check out shell sort

graph

$$G = \{V, E\}$$

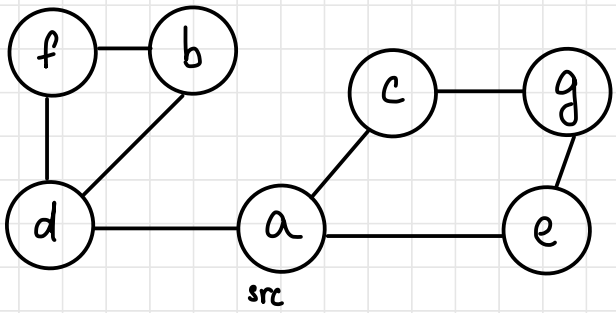


traversals — DFS, BFS

- While traversing, ordered tree created

traversals

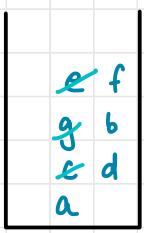
DFS

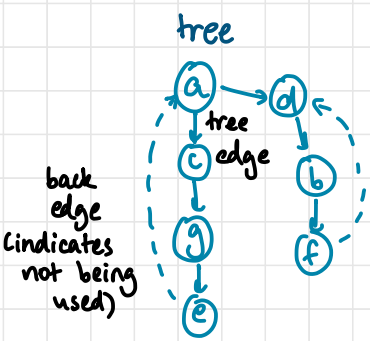


(alphabetically - convention)

	a	b	c	d	e	f	g
visited	1	0	0	0	0	0	0
	1	0	1	0	0	0	0
	1	0	1	0	0	0	1
	1	0	1	0	1	0	1
	1	0	1	1	1	0	1
	1	1	1	1	1	0	1
	1	1	1	1	1	1	1

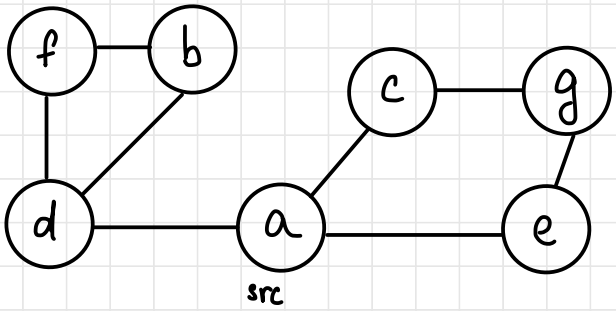
stack





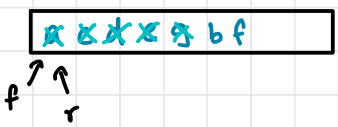
- Efficiency - adj matrices : $\Theta(|V|^2)$
- Efficiency - adj lists : $\Theta(|V|+|E|)$

BFS



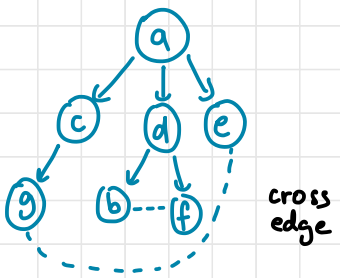
While queue not empty

Queue



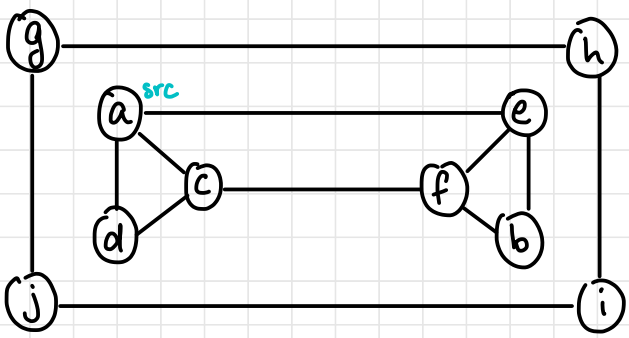
	a	b	c	d	e	f	g
visited	1	0	0	0	0	0	0
	1	0	1	1	1	0	0
	1	0	1	1	1	0	1
	1	1	1	1	1	1	1

tree

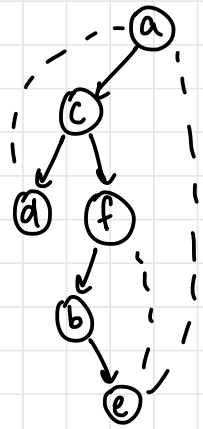


- Efficiency - adj matrices : $\Theta(|V|^2)$
- Efficiency - adj lists : $\Theta(|V|+|E|)$

disconnected graph

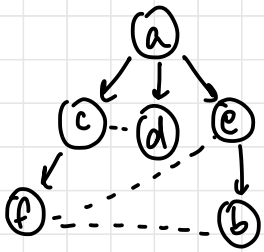


DFS from a



a c d f b e

BFS from a



a c d e f b

solution

- restart DFS from an unvisited node
- will create forest

SPARSE MATRIX

- adj list preferred

BFS Algorithm

ALGORITHM *BFS*(G)

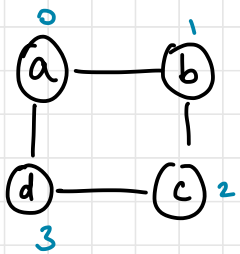
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable count
count ← count + 1;  mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```

Comparison

	DFS	BFS
Data structure	a stack (uses stack, pop, push)	a queue (queue)
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

IMPLEMENTATION IN C



	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

```c
void dfs(int source, int adj[][MAX], int *visited, int n) {
    visited[source] = 1;
    printf("%d ", source);

    for (int i = 0; i < n; ++i) {
        if (adj[source][i] && !visited[i]) {
            dfs(i, adj, visited, n);
        }
    }
}
```

```c
void bfs(int adj[][MAX], int *visited, int n, int *queue, int f, int r) {
    int v;

    while (!empty(queue, f, r)) {
        v = dequeue(queue, &f, &r);
        printf("%d ", v);

        for (int i = 0; i < n; ++i) {
            if(adj[v][i] && !visited[i]) {
                enqueue(queue, i, &f, &r);
                visited[i] = 1;
            }
        }
    }
    printf("\n");
}
```

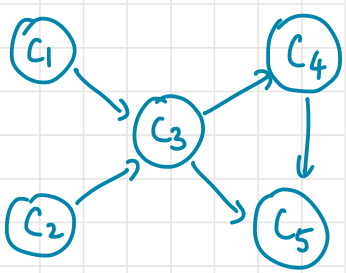
TOPOLOGICAL SORTING

- Graphs: edge's start vertex before end vertex
- Directed graphs / digraph
- Acyclic graphs (only) have solutions

Q: Topological sorting - problem.

Consider a set of five different courses {C1, C2, C3, C4, C5} that must be completed in order to fetch a degree.

Prerequisites to be met represented in graph. Only one course per term. What is the order in which the courses must be taken?



	1	2	3	4	5
1	0	0	1	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	0	0	0	0	1
5	0	0	0	0	0

Algorithm 1

- Perform DFS
- Order in which they become dead-ends (popped out of stack)
- Reversing order yields solution

Solution 1

- start with C_1
- order in which they pop:

  C_5 , C_4, C_3, C_1
- start with C_2 , pop

C_2 $C_1 \rightarrow C_3 \rightarrow C_4 \rightarrow C_5$

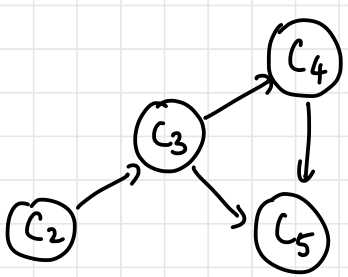
Algorithm 2

- Identify source vertex (indegree = 0)
- Delete source vertex and outgoing edges

} decrease and conquer

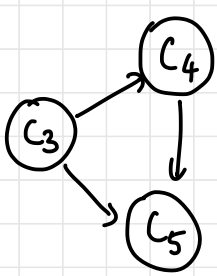
Solution 2

1. Delete C_1



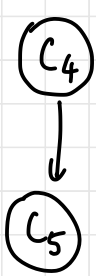
order: C_1

2. Delete C_2



order: $C_1 \rightarrow C_2$

3. Delete C_3



order: $C_1 \rightarrow C_2 \rightarrow C_3$

4. Delete C_4

order: $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$

5. Delete C_5

order: $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow C_5$

CODE IMPLEMENTATION IN C

Algorithm 1

- stack[] is an array indicating if stack[i] is in the stack, not the actual stack itself

```c
void dfs(int v, int graph[][MAX], int n, int visited[], int topoorder[], int stack[], int *index) {
    int i;
    visited[v] = 1;
    stack[v] = 1;

    for (i = 0; i < n; i++) {
        if (graph[v][i] && !visited[i]) {
            dfs(i, graph, n, visited, topoorder, stack, index);
            stack[i] = 0;

            topoorder[*index] = i;
            (*index)++;

        }
        else if (graph[v][i] && visited[i] && stack[i]) {
            printf("\nGraph has a cycle and hence there is no solution");
            exit(1);
        }
    }
}
```

in main

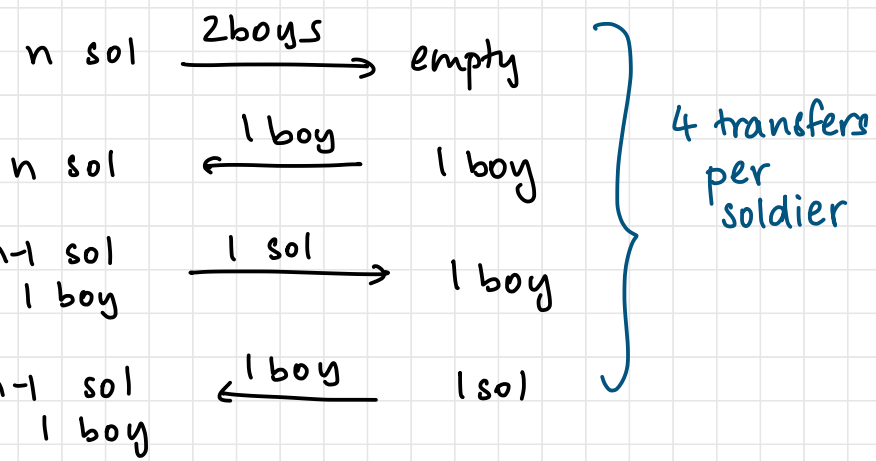
```
for (i = 0; i < n; i++) {
    if (visited[i] == 0) {
        dfs(i, graph, n, visited, topoorder, stack, &index);
        stack[i] = 0;
        topoorder[index] = i;
        index++;
    }
}
printf("\nTopological order:\n");
for (i = n - 1; i >= 0; i--) {
    printf("%d ", topoorder[i]);
}
```

Q: Problem: transfer from shore 1 to shore 2

n soldiers 2 boys 1 boat 2 boys or 1 soldier

No. of transfers required = ?

To transfer 1 soldier



$\therefore$ $4n$ for n soldiers

$+ 1$ for 2 boys

$\therefore$ total $= 4n+1$ trips

Q: Is given graph bipartite ?

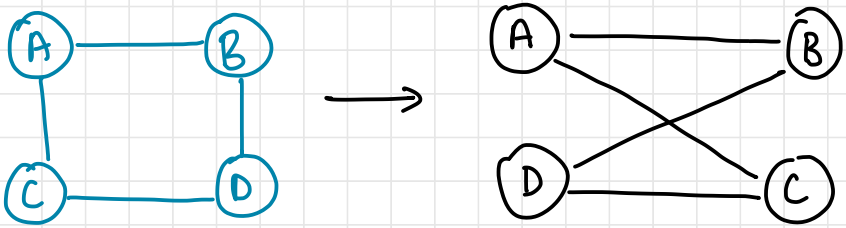
OR

Is graph 2-colourable

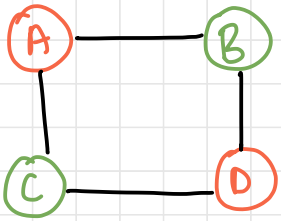
OR

Can graph be divided into 2 disjoint subsets X & Y such that every node in X has an edge to Y

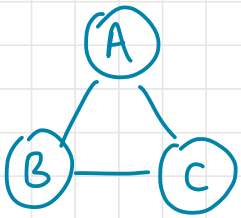
(i)



· No 2 adjacent vertices have the same colour



(ii)



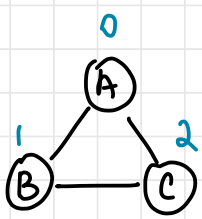
not possible

(iii) Graph (V, E) → use 2-colour theorem

- Use DFS
- Source vertex: colour 1
  Adjacent vertex: colour 2
  Next vertex: colour 1
  ⋮

  no adjacent vertex should have same colour

- eg:



dfs(A) → A: green
dfs(B) → B: red
dfs(C) → C: adjacent (A) is already green (same colour)

∴ fails

IMPLEMENTATION IN C

```c
int is_bipartite(int source, int adj[][MAX], int *visited, int n, int *colors, int cur_color) {

    visited[source] = 1;
    colors[source] = cur_color;

    cur_color = !cur_color;

    int res = 1;

    for (int i = 0; i < n; ++i) {
        if (adj[source][i]) {
            if (!visited[i]) {
                res = is_bipartite(i, adj, visited, n, colors, cur_color);
            }
            else {
                if (colors[i] == colors[source]) {
                    return 0;
                }
            }
        }
    }
    return res;
}
```

Generation of Permutations

- Given a set S = {A, B, C}, how many permutations?
- 3! = n! where n = size of set
- Produce all n! permutations
- Best : n!

Johnson-Trotter Algorithm

topcoder generating permutations

{1}

{1, 2}

{1, 2, 3}

1

21 12

321 231 213 312 132 123

introduce 3 as RM element

adjacent swaps

{1, 2, 3} 321 231 213 123 132 312

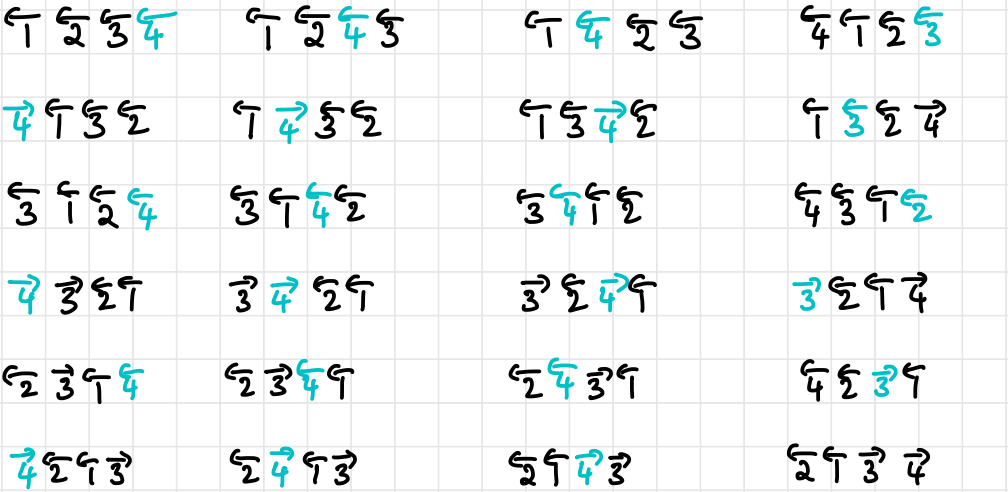
alternate

Algorithm

- Put the n^{th} element in all positions
- Makes it look like it is 'moving'

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list



CODE IMPLEMENTATION IN C

- Store array elements in a[] with index = position
- Store directions in dir[] where index = element
- Function called from main – printpermutations(n)

Constants & Helper Functions

```c
// Constants
#define MAX 10
#define RtoL 0
#define LtoR 1

int fact(int n) {
    int res = 1;
    for (int i = 1; i <= n; i++) {
        res = res * i;
    }
    return res;
}

void swap(int *ele1, int *ele2) {
    int temp;
    temp = *ele1;
    *ele1 = *ele2;
    *ele2 = temp;
}
```

Print all Permutations

```
void printpermutations(int n) {
    int perm[MAX], dir[MAX];

    // Initialise with all pointing RtoL, order 1...n
    for (int i = 0; i < n; i++) {
        perm[i] = i + 1;
        dir[i] = RtoL;
    }

    printf("\nPermutations:\n");
    // First permutation
    for (int i = 0; i < n; i++) {
        printf("%d ", perm[i]);
    }
    printf("\n");

    int factn = fact(n);

    // Print subsequent permutations
    for (int i = 1; i < factn; i++) {
        // Gets next permutation
        getonepermutation(perm, dir, n);
    }
}
```

Helper function- position of element

```
// Gets the position of the mobile element (indexed at 1)
int searchperm(int perm[], int n, int mobile) {
    for (int i = 0; i < n; i++)
        if (perm[i] == mobile) {
            return i + 1;
        }
    return -1;
}
```

Get Next Permutation

```c
// Get next permutation
void getonepermutation(int perm[], int dir[], int n) {
    int mobile = getmobile(perm, dir, n);
    // Gets the position of the mobile element (indexed at 1)
    int pos = searchperm(perm, n, mobile);

    // If pointing left, swap with left element
    if (dir[perm[pos - 1] - 1] == RtoL) {
        swap(&perm[pos - 1], &perm[pos - 2]);
    }

    // If pointing right, swap with right element
    else if (dir[perm[pos - 1] - 1] == LtoR) {
        swap(&perm[pos], &perm[pos - 1]);
    }

    // Check entire array for elements greater than
    // the one being moved and flip direction
    for (int i = 0; i < n; i++) {
        if (perm[i] > mobile) {
            if (dir[perm[i] - 1] == RtoL) {
                dir[perm[i] - 1] = LtoR;
            }
            else if (dir[perm[i] - 1] == LtoR) {
                dir[perm[i] - 1] = RtoL;
            }
        }
    }
    // Print the permutation
    for (int i = 0; i < n; i++) {
        printf("%d ", perm[i]);
    }
    printf("\n");
}
```

```java
// Gets largest mobile element
int getmobile(int a[], int dir[], int n) {
    int mprev = 0, mobile = 0;
    for (int i = 0; i < n; i++) {
        // Largest moving element from right to Left
        if (dir[a[i] - 1] == RtoL && i != 0) {
            if (a[i] > a[i - 1] && a[i] > mprev) {
                mobile = a[i];
                mprev = mobile;
            }
        }
        // Largest moving element from left to right
        if (dir[a[i] - 1] == LtoR && i != n - 1) {
            if (a[i] > a[i + 1] && a[i] > mprev) {
                mobile = a[i];
                mprev = mobile;
            }
        }
    }
    if (mobile == 0 && mprev == 0)
        return 0;
    else
        return mobile;
}
```

LEXICOGRAPHIC ORDER – GENERATE PERMUTATIONS

1, 2, 3, 4 ⟶ 1, 2, 4, 3 ⟶ 1, 3, 2, 4 ⟶ ...

ALGORITHM *LexicographicPermute(n)*

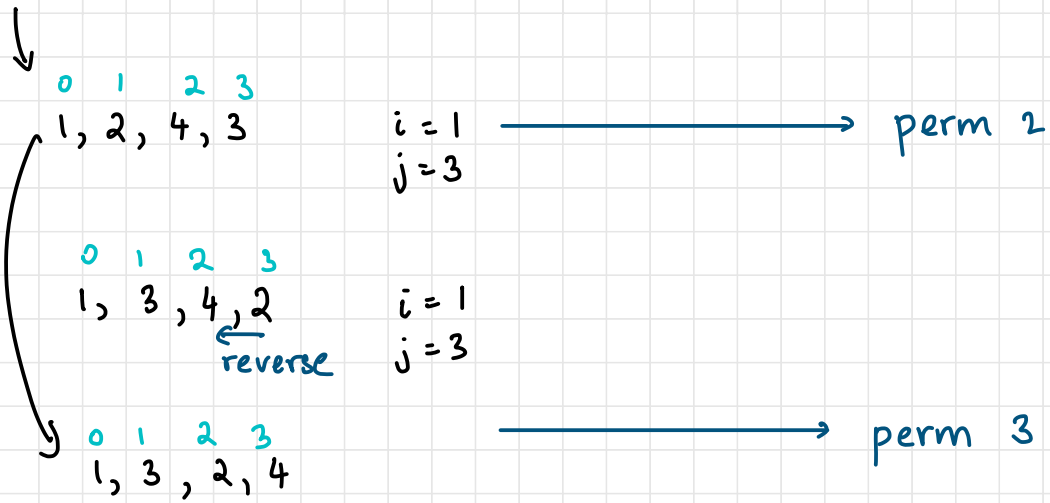
//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
  let i be its largest index such that $a_i < a_{i+1}$ // $a_{i+1} > a_{i+2} > \cdots > a_n$
  find the largest index j such that $a_i < a_j$ // $j \geq i+1$ since $a_i < a_{i+1}$
  swap a_i with a_j // $a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
  reverse the order of the elements from a_{i+1} to a_n inclusive
  add the new permutation to the list

0 1 2 3

1, 2, 3, 4

i = 2
j = 3

⟶ perm 1



IMPLEMENTATION IN C

```c
void lex(int *a, int n) {
    int i, j, finished = 0, temp;
    quicksort(a, 0, n - 1);
    display(a, n);
    while (!finished) {
        // Find pair of increasing elements
        for (i = n - 2; i >= 0; --i) {
            if (a[i] < a[i + 1]) {
                break;
            }
        }

        // If i = -1, all are in decreasing order
        if (i == -1) {
            finished = 1;
        }

        else {
            for (j = n - 1; j > i; --j) {
                if (a[j] > a[i]) {
                    // Swap a[i] and a[j]
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                    break;
                }
            }
            quicksort(a, i + 1, n - 1);
            display(a, n);
        }
    }
}
```

OUTPUT

```
→  3-5 Permutations Lexicographic ./lex
Enter n: 4
Enter elements: 1 2 3 4
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
```

Generation of Subsets

- use decrease & conquer
- use generated subsets of (n-1) elements and use it to generate for n elements
- $\emptyset$ —— $\emptyset$ —— 2^0 el
  $\{1\}$ —— $\emptyset$, $\{1\}$ —— 2^1 el
  (w/o 1, w 1)
  $\{1,2\}$ —— $\emptyset$, $\{1\}$, $\{2\}$, $\{1,2\}$ — 2^2 el
  (w/o 2, w 2)
  ⋮
  .

- $T(n) = 2 \cdot T(n-1)$ let $T(0) = 1$
  $= 2^2 \cdot T(n-2)$
  $= 2^i \cdot T(n-i)$

  $T(n) = 2^n \cdot 1 \Rightarrow O(2^n)$

- Binary string of size $n \longrightarrow 2^n$ numbers
- One-one correspondence between binary strings and subsets
- eg: binary string of size 2:

00	01	10	11
$\emptyset$	{1}	{2}	{1,2}

- enumerate all numbers: enumerate all subsets

KNAPSACK problem

Find the most valuable subset of elements to fit into a knapsack of a given capacity

Find all subsets and find optimal subset

Squashed Order

Any subset involving a_j can be listed only after all subsets involving $a_1, a_2 \dots a_{j-1}$ are listed

MSB turns 1 only after all LSB's enumerated

GRAY CODES

000 001 011 010 110 111 101 100

also in squashed order

n=1

0
1

n=2

0 0
0 1
1 1
1 0

reflect

n=3

0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0

reflect

binary reflected gray code (Frank Gray)

Decrease by Constant Factor

FAKE COIN PROBLEM

Given n coins where one is fake (lighter in weight) where all others are of the same weight

Algorithm

1. If even, divide into 2 even piles; if odd then leave one coin out
2. Weigh 2 piles; lighter pile contains fake coin
3. If equal, leftover coin is fake
4. Repeat until found

$W(n) = W(n/2) + 1$

$W(n) = \log_2 n$

Improvement

Divide into 3 piles

$W(n) = \log_3 n \longrightarrow$ slight improvement

Russian Peasant multiplication

- Multiply m and n recursively

$$n \times m = (n/2) \times 2m \qquad n \text{ even}$$

$$n \times m = (n-1)/2 \times 2m + m \qquad n \text{ odd}$$

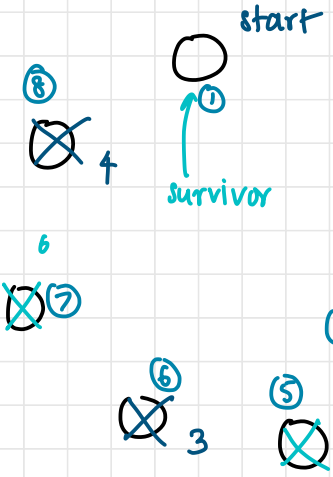
Q: 50 x 65

n		m		
50	x	65		
= 25	x	130		
= 12	x	260	+	130
= 6	x	520	+	130
= 3	x	1040	+	130
= 1	x	2080	+	130
= 3250				

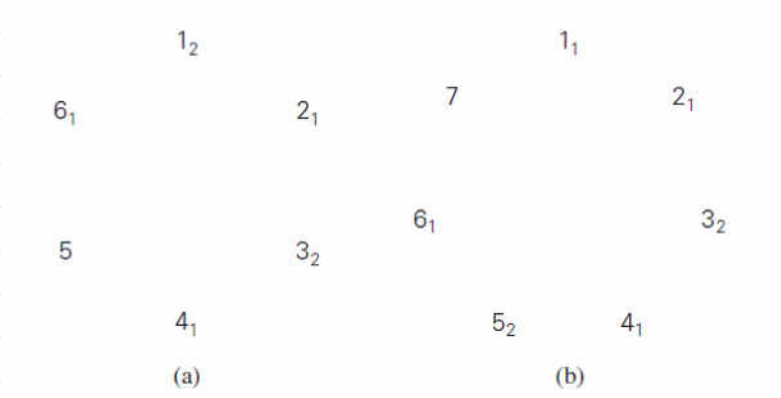
```
int russian_peasant(int m, int n) {
    int sum = 0;
    if (m != 0 || n != 0) {
        while (n != 1) {
            if (n % 2 != 0) {
                sum = sum + m;
            }
            n = n / 2;
            m = m + m;
        }
        sum = sum + m;
    }
    return sum;
}
```

Fast Fourier & Inverse multiply $O(n \log n)$

JOSEPHUS PROBLEM



- People standing in a circle
- Start at one spot, each eliminates immediate neighbour



(a) Josephus
n = 6

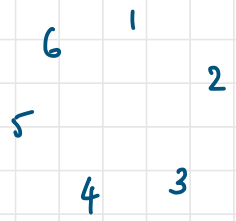
J(6) = 5

(b) Josephus
n = 7

J(7) = 7

(a) Even

- every round, 1/2 eliminated
- 1 lives for another round
- eg: $n = 6$



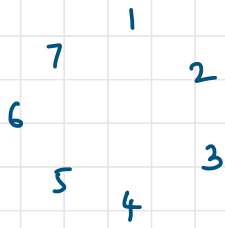
$n = 6$ $(2k)$	1	2	3	4	5	6
		↓				
$n = 3$ (k)	1		3		5	
	1		2		3	

mapping: initial position of survivor (for $2x$ people)

$= 2 *$ new position $- 1$

$$J(2k) = 2J(k) - 1$$

(b) Odd



$3 = 2k+1$

$k = 1 = 3/2$

$2 * j(k/2)+1$

start = 1

n=7 (2k+1)	1	2	3	4	5	6	7
n = 3 (k)			3 1		5 2		7 3

Survivors' old position = 2 * new position + 1

$J(2k+1) = 2J(k)+1$

Efficient Algorithm

- 1-bit cyclic left shift of Binary Number
- $J(6) = \overline{5}$

  $J(110) = 101$

- $J(8) = J(1000) = 0001$
- Interestingly, result is always odd (1 MSB always becomes LSB)

IMPLEMENTATION IN C

```c
int josephus(int n) {
    if (n == 1)
        return 1;
    else if (n % 2) {
        /* Odd number */
        return (2 * josephus(n / 2)) + 1;
    }
    else {
        /* Even number */
        return (2 * josephus(n / 2)) - 1;
    }
}
```

EFFICIENT ALGORITHM IN C

```c
int josephus_eff(int n) {
    int temp = n, size;

    // size = highest set bit in number
    for (int i = 0; i < sizeof(int) * 8; ++i) {
        if ((temp >> i) & 1) {
            size = i + 1;
        }
    }
    return ((n << 1) & (__UINT32_MAX__ >> (sizeof(int) * 8 - size)) | 1);
}
```

circular left shift

eg: 8-bit n = 6 , sizeof(n) = 8

n = 0 0 0 0 0 1 1 0

n<<1 = 0 0 0 0 1 1 0 0

MAX >>5 = 0 0 0 0 0 1 1 1

1 = 0 0 0 0 0 0 0 1

TRANSFORM & CONQUER

- Problem transformed into simpler form
- Three major variations

1) Instance Simplification

- simpler instance of problem
- duplicate elements: presort and then compare adjacent elements

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise

sort the array A] $O(n \log n)$

for $i \leftarrow 0$ **to** $n-2$ **do**
if $A[i] = A[i+1]$ **return false**
return true] $O(n)$

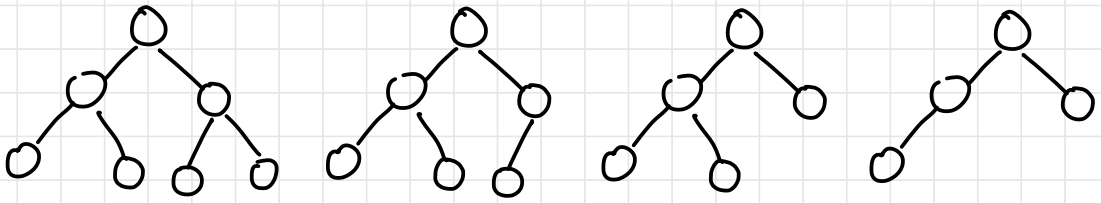
$$T(n) \in O(n \log n) + O(n) = O(n \log n)$$

2) Representation Change

3) Problem Reduction

heap

- Operations
  - findmax O(1)
  - insert O(log n)
- Implemented using binary trees
- Full binary tree: all nodes have 0 or 2 children
- Complete binary tree: $2^n - 1$ nodes where all leaf nodes are on the same level - strict definition, not for all values of array length
- Complete binary tree: all leaves need not be at same level; can be at prev level



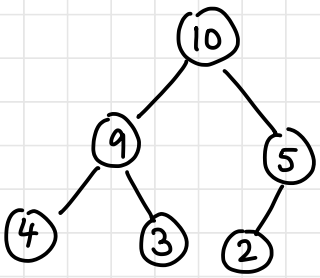
n=7 n=6 n=5 n=4

- Not more than one node has only one child node

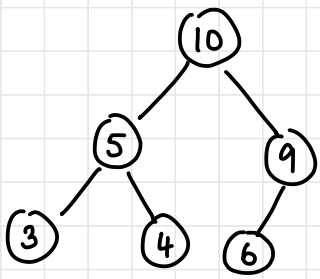
- Partially ordered DS: number stored in a node must be greater than the numbers stored in its children (recursive property) — max heap
- Elements in a heap are typically unique (keys)

Example heaps

(1)

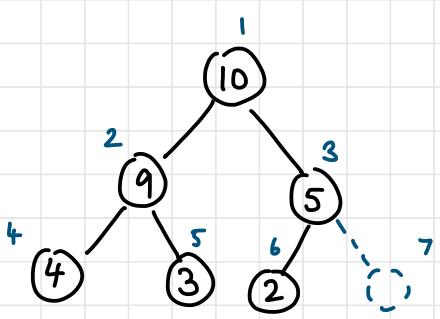


(2)

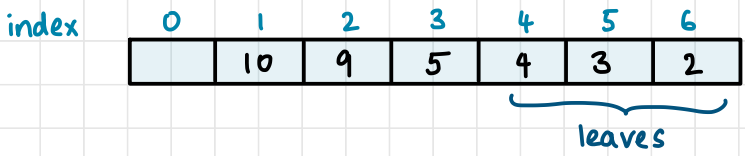


Representation Using Array

- Breadth-first traversal of tree / level traversal
- Root node placed at location 1



- Array representation



- Even length of heap (2k): one less than a full binary tree where first k indices are non-leaf nodes and the next k indices are leaf nodes

- Odd length of heap (2k+1): full binary tree where first k indices are non-leaf nodes and the next k+1 indices are leaf nodes

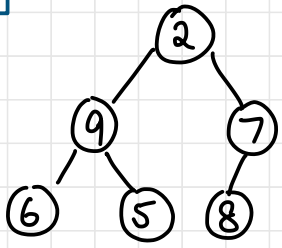
- Generally: leaf nodes = $\left\lceil \frac{n}{2} \right\rceil$ and non-leaf nodes = $\left\lfloor \frac{n}{2} \right\rfloor$

- index of node: i

- index of left child: $2i$

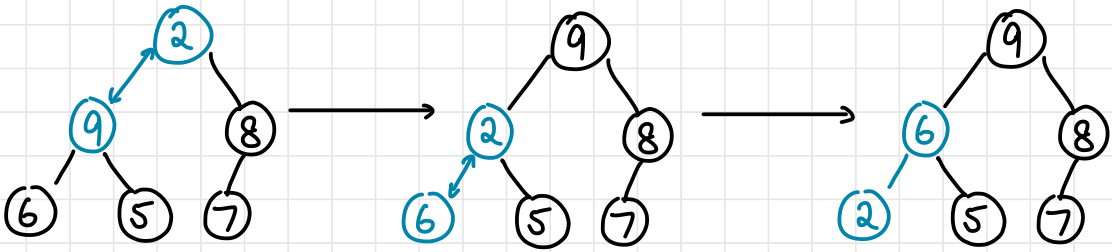
- index of right child: $2i + 1$

Construct Heap - Bottom Up



- All sub-trees must be heaps (bottom-up)
- All leaf nodes are already heaps (by definition)
- Therefore, start with last non-leaf node
- Start with 7 as root; it is not a heap; swap root with greater child

- Check previous node, i.e 9 as root; it is a heap
- Check 2 as root; it is not a heap; replace with bigger child



- Go to swapped child and check if heap and keep swapping until heap is formed

ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;  v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];  k ← j
    H[k] ← v
```

index = k, value = v

left child

(n > j means j+1 exists)

right child greater

until subtree forms heap

root of subtree is greater child

Let root = old index of greater child

time complexity

2 comps: one b/w children and one w parent

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{j=0}^{2^i} 2(h-i)$$

i: levels

h = height of tree

h = $\lfloor \log_2 n \rfloor$

$$C_{worst}(n) = \sum_{i=0}^{h-1} 2^i \cdot 2(h-i) \qquad n = 2^h$$

$$= \sum_{i=0}^{h-1} 2^{i+1}(h-i)$$

$$= \sum_{i=0}^{h-1} h \cdot 2^{i+1} - \sum_{i=0}^{h-1} i\,2^{i+1}$$

USEFUL SUMMATION

$$\sum_{i=1}^{n} i\,2^i = 1\cdot 2 + 2\cdot 4 + 3\cdot 8 + \ldots + n2^n = (n-1)2^{n+1} + 2$$

$$= \sum_{i=0}^{h-1} h \cdot 2^{i+1} - \sum_{i=1}^{h-1} i\,2^{i+1}$$

$$= 2h \sum_{i=0}^{h-1} 2^i - 2 \sum_{i=0}^{h-1} i2^i$$

$$= 2h\,(2^h - 1) - 2\left[(h-2)2^h + 2\right]$$

$$= 2h\,(2^h) - 2h - h \cdot 2^{h+1} + 2^{h+2} - 4$$

$$= \cancel{h \cdot 2^{h+1}} - \cancel{h \cdot 2^{h+1}} - 2h + 2^{h+2} - 4$$

$$= -4 - 2h + 2^{h+2}$$

$$= -4 - 2\log_2 n + 2n$$

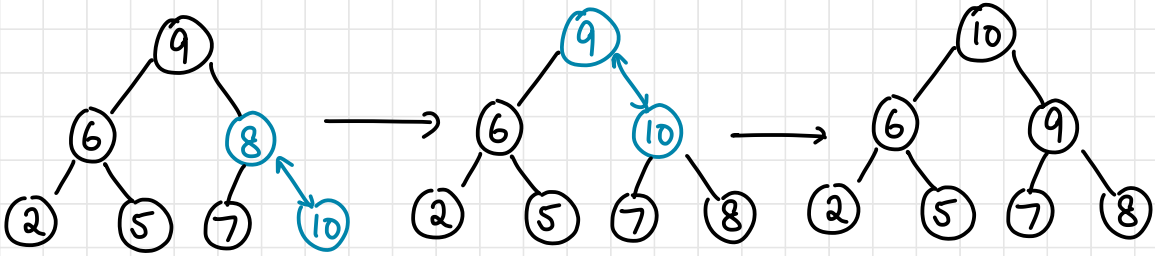
$\in \Theta(n)$

IMPLEMENTATION IN C

```c
void heapify(int heap[], int n) {
    int i, j, k, v;
    int flag;   /* heapify still happening */
    for (i = n / 2; i >= 1; i--) {
        k = i;
        v = heap[k];
        flag = 0;
        while (flag == 0 && 2 * k <= n) {
            j = 2 * k;      /* 2*k -> k's left child */
            if (j < n) {    /* Right child exists */
                if (heap[j] < heap[j + 1]) {
                    /* Right child bigger */
                    j = j + 1;
                }
            }
            if (v > heap[j]) {
                flag = 1;
            }
            else {
                /* Move large child up */
                heap[k] = heap[j];
                /* Large child = new key index */
                k = j;
            }
        }
        /* Original parent stored back */
        heap[k] = v;
    }
}
```

Insert an Element

- Add element to end of array
- Percolate up to right place



top-down approach

- Insert a new key K into a heap by attaching a new node with key K in it after the last leaf of the existing heap.
- Sift K up to its appropriate place in the new heap as follows.
  - Compare K with its parent's key: if the latter is greater than or equal to K, stop (the structure is a heap);
  - otherwise, swap these two keys and compare K with its new parent.
- This swapping continues until K is not greater than its last parent or it reaches the root
- Insertion operation cannot require more key comparisons than the heap's height.
- The time efficiency of insertion is in $O(\log n)$.

- $\Theta(\log n)$

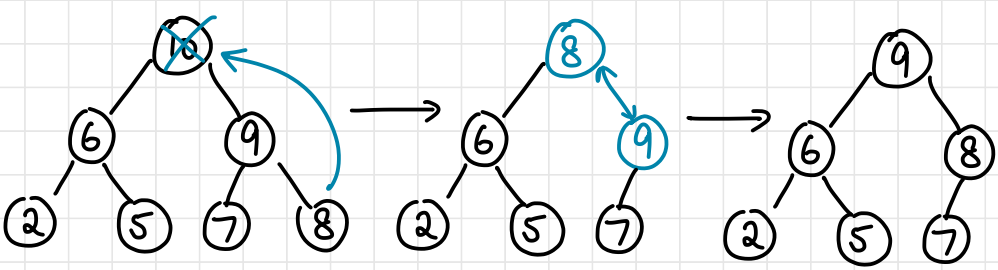
IMPLEMENTATION IN C

```c
void top_heap(int *h, int n) {
    int i, j, k, key;
    for (k = 1; k <= n; k++) {
        i = k;
        key = h[i];
        j = (i) / 2; // get the parent

        while ((i > 1) && (key > h[j])) {
            h[i] = h[j]; // move the parent down
            i = j;
            j = (i) / 2; // find the new parent
        }
        h[i] = key; // insert key
    }
}
```

Delete Max

- Remove root, construct new heap



- $\Theta(\log n)$

heap sort

Algorithm
1. Construct heap for array
2. Apply delete-root n-1 times

$\Theta(n \log n)$

- Merge, quick sort preferred

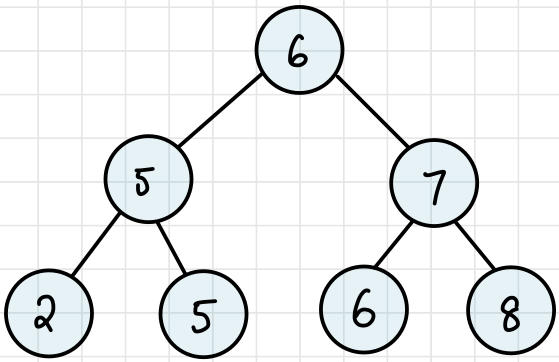
IMPLEMENTATION IN C

```c
void heapsort(int *h, int n) {
    int i, t;
    // Create initial heap;
    top_heap(h, n);
    for (i = n; i > 0; i--) {
        // Swap the first and the last element
        t = h[0];
        h[0] = h[i];
        h[i] = t;
        // Recreate a heap for remaining set of elements
        heapify(h, i - 1);
    }
}
```

can use adjust also

BINARY SEARCH TREES

- Cormen textbook ; $\log_2 n = \lg n$; pg 286 (303)



binary search tree Property

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

SEARCH

Algorithm

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2       return x
3  if k < x.key
4       return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

if height of tree = h , $T \in \Theta(h)$

INSERT

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == NIL
10      T.root = z          // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

if height of tree = h , $T \in \Theta(h)$

DELETION

Three cases - delete node z from tree

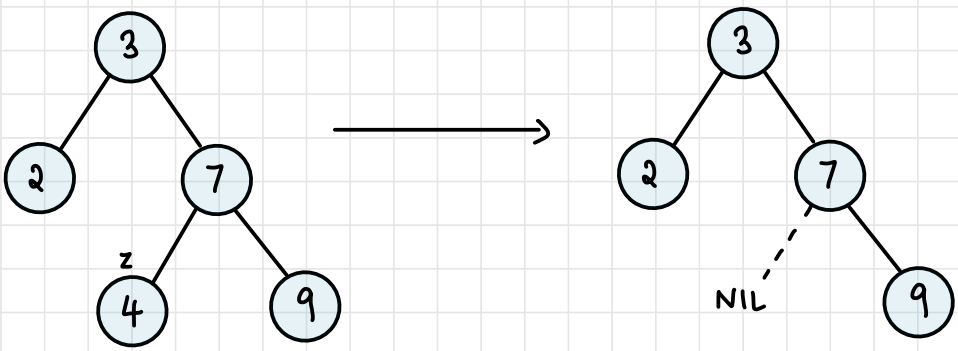
1. If z has no children, simply remove z and parent should replace z with NIL
2. If z has just one child, elevate that child to take z's position and parent replaces z by z's child
3. If z has two children, find inorder successor y of z, and have y take z's place in that tree. The rest of z's original right subtree becomes y's right subtree and the rest of z's left subtree becomes y's left subtree.

(i) If y is z's right child, replace z by y, leaving y's right child alone

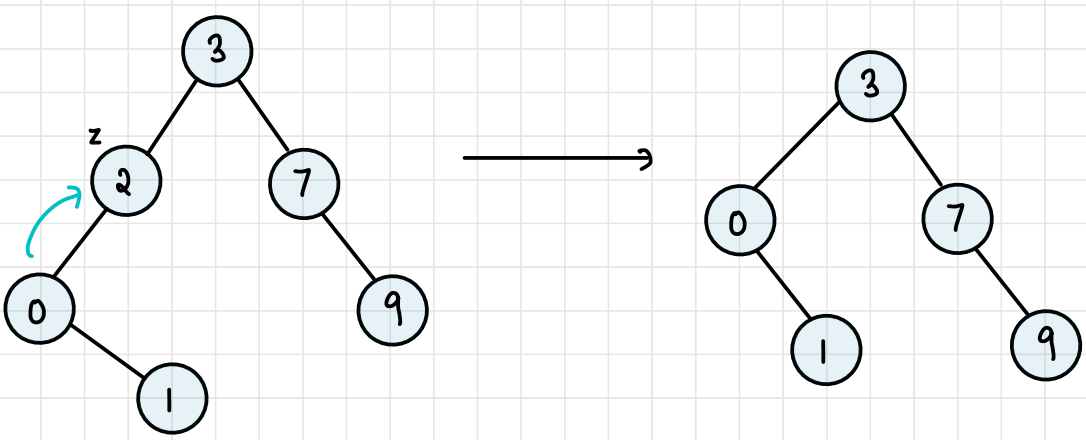
(ii) If y is in z's right subtree but is not the direct right child of z, replace y by its right child and then repace z by y

Example:

1. z has no children

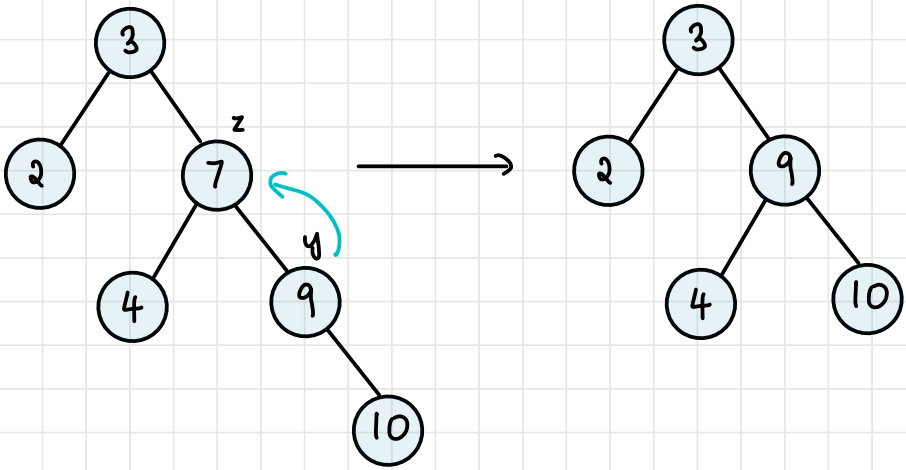


2. z has one child

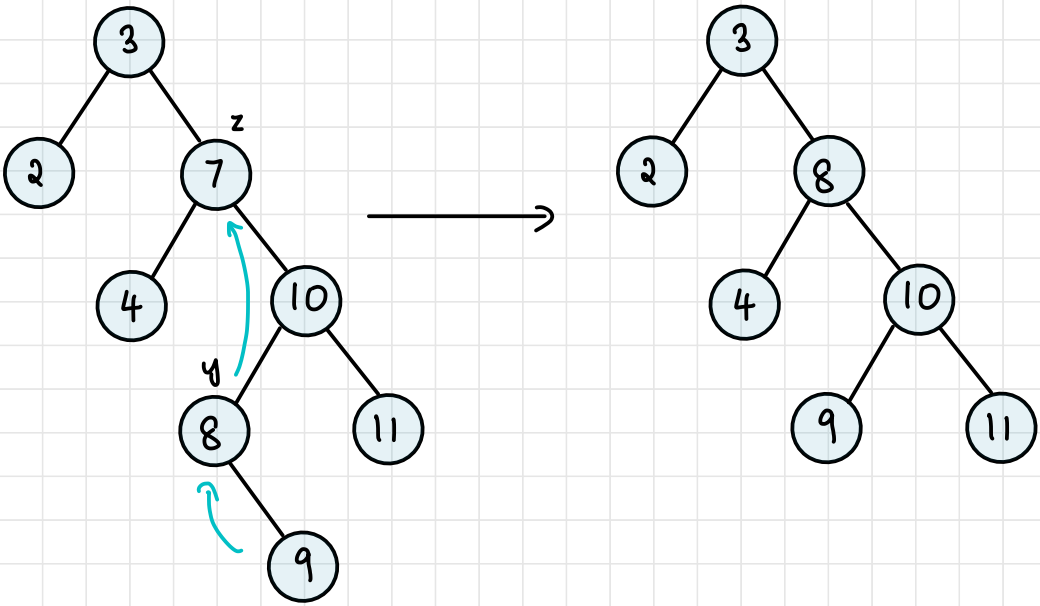


3. z has 2 children

(i) y is z's right child



(ii) y is not z's right child



Time Complexity

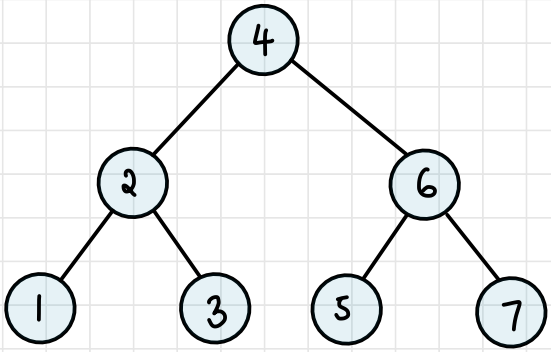
- Inorder successor: $\Theta(h)$
- $T_{delete} \in \Theta(h)$

BINARY TREE

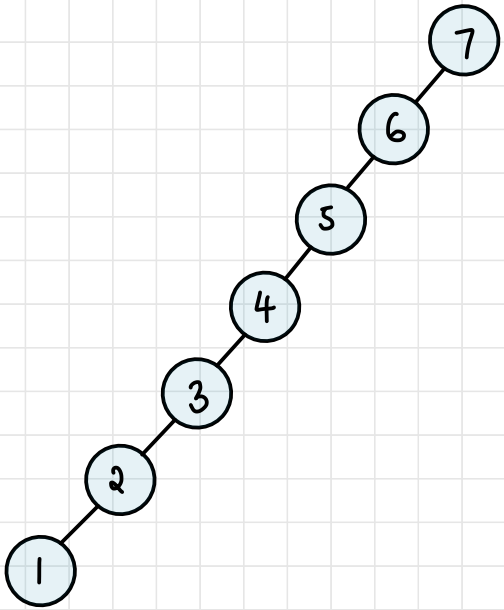
- find —— $O(h)$
- insert —— $O(h)$
- delete —— $O(h)$

Order of Insertion Matters

(1) Best-case: 4, 2, 6, 1, 3, 5, 7

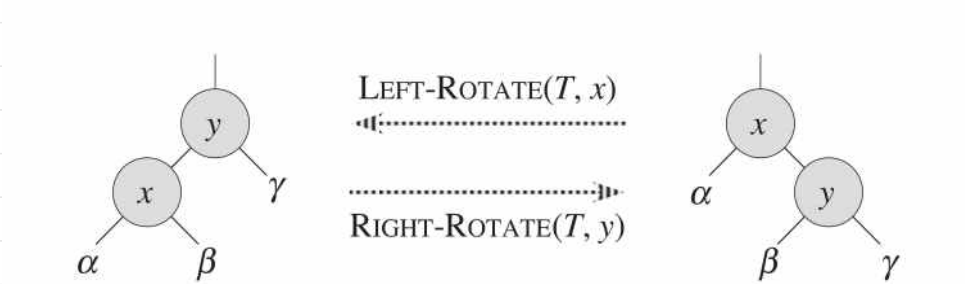


(2) Worst-case: 7, 6, 5, 4, 3, 2, 1



AVL TREES

Rotations



Maintenance and rotations take a long time

RED BLACK TREES

- Not strict complete BST
- Instead of height $\approx \log_2 n$, allow for $2 \log_2 n$ levels
- Each node has extra bit representing colour of the node (red or black)

PROPERTIES

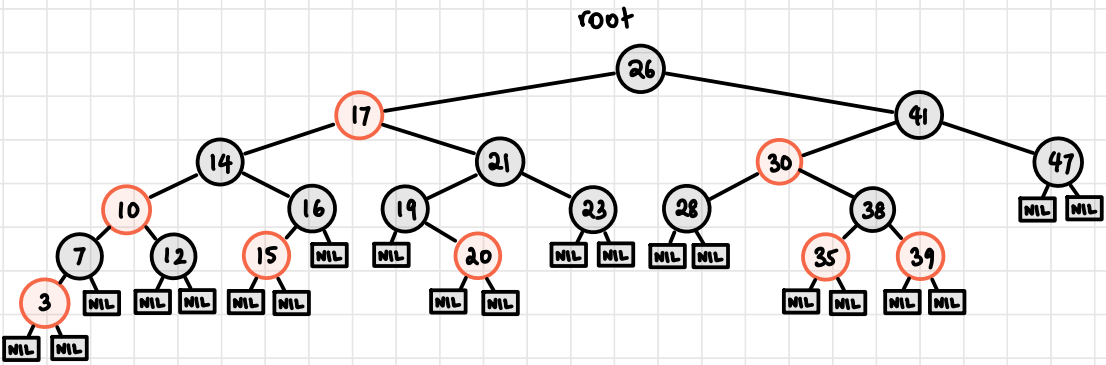
1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, both its children are black
5. For each node, all simple paths from the node to descendant leaves contains the same number of black nodes

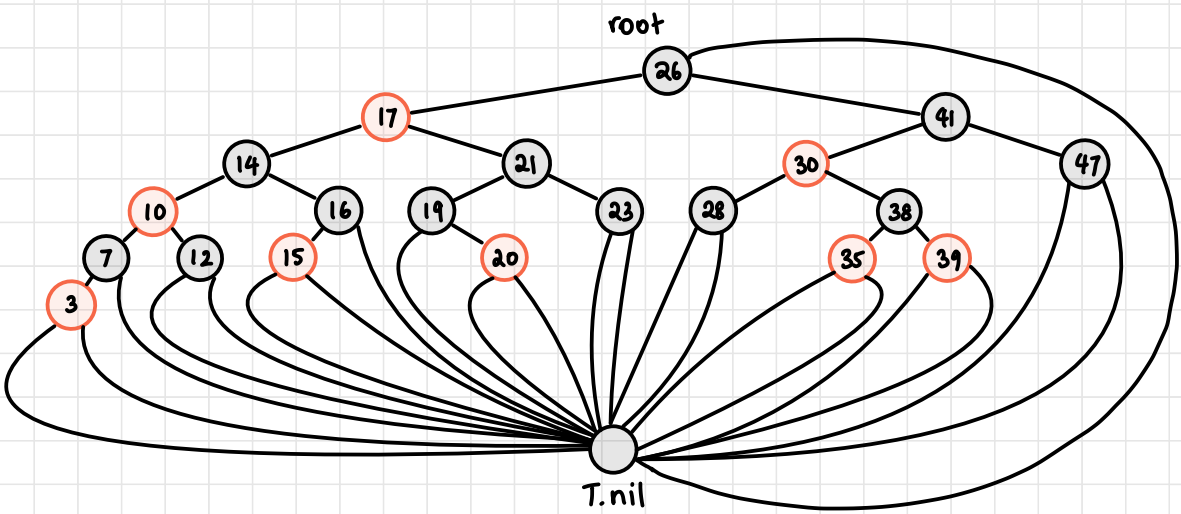
Lemma

if a RB tree has n internal nodes, height is at most $2\log_2(n+1)$

Note

- Logically, need NIL nodes for every leaf node
- Implementing, all leaves point to same NIL
- Parent of root also NIL





OPERATIONS

- find —— O(log n)
- insert ——

FIND | SEARCH

- same as BST
- findmax, findmin, inorder successor all same as BST

insertion

ALGORITHM

same as BST

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4      y = x
 5      if z.key < x.key
 6          x = x.left
 7      else x = x.right
 8  z.p = y
 9  if y == T.nil
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

colour inserted node red

fixup function (only if its parent is red)

FIXUP

```
RB-INSERT-FIXUP(T, z)
1   while z.p.color == RED
2       if z.p == z.p.p.left
3           y = z.p.p.right
4           if y.color == RED
5               z.p.color = BLACK
6               y.color = BLACK
7               z.p.p.color = RED
8               z = z.p.p
9           else if z == z.p.right
10                  z = z.p
11                  LEFT-ROTATE(T, z)
12              z.p.color = BLACK
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

while its parent is red

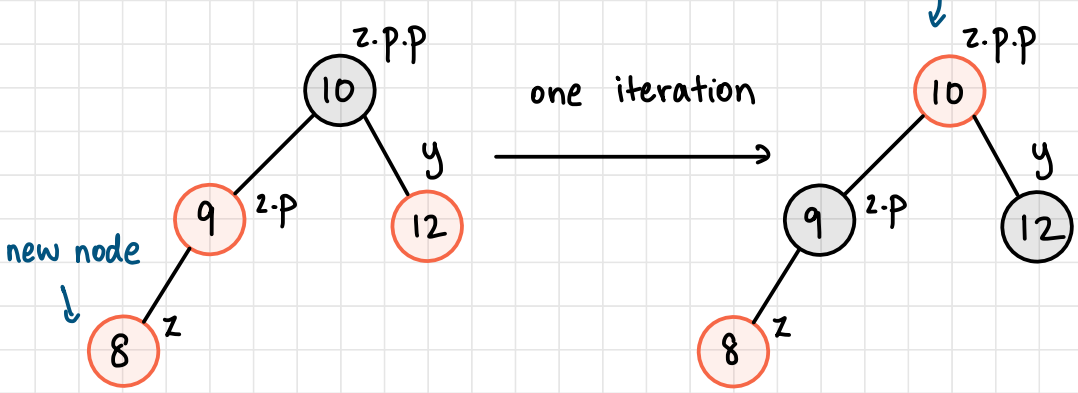
if z.p is left child of its parent

Case 1

Case 2

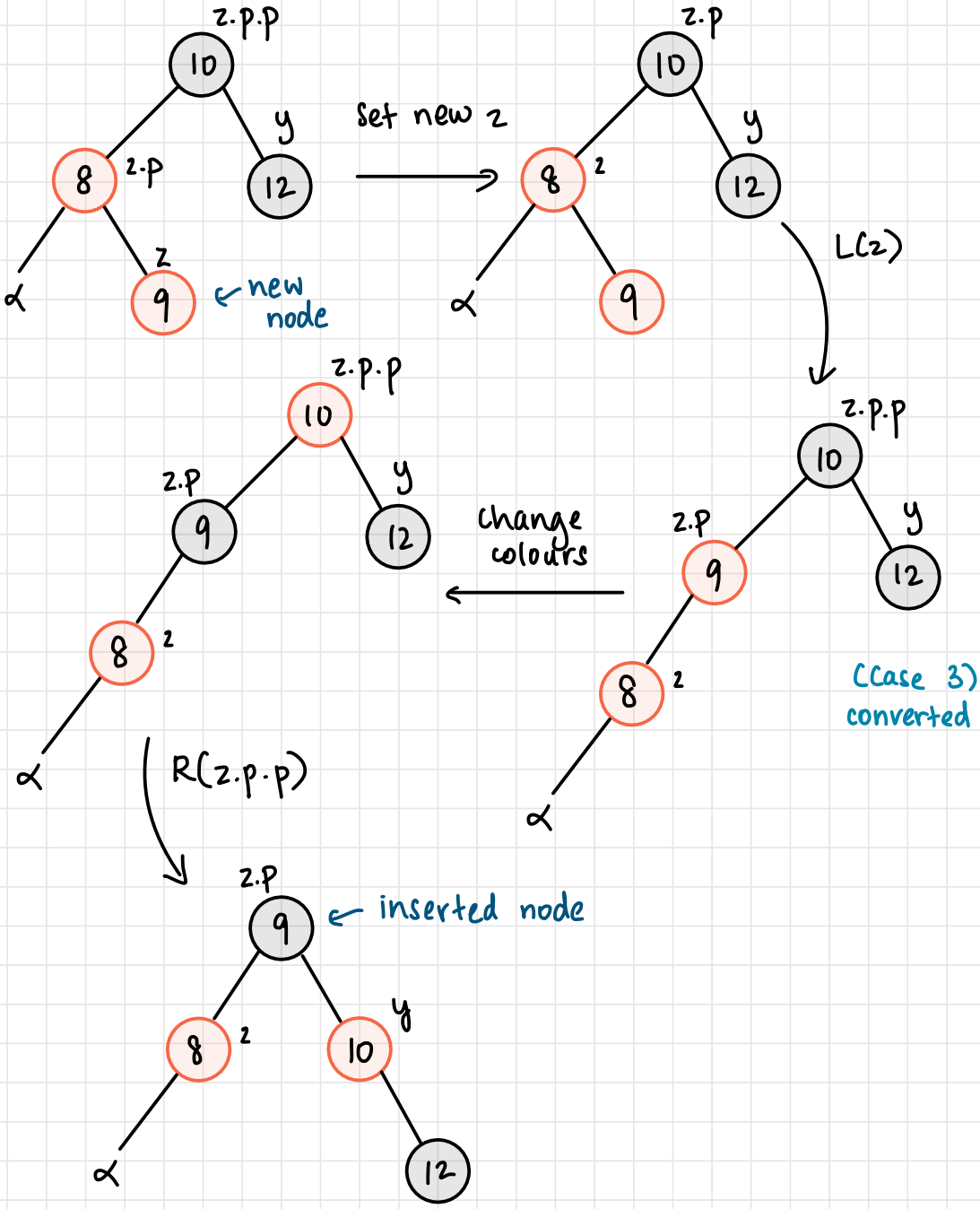
overwrite root node colour

Case 1 - uncle of z is red



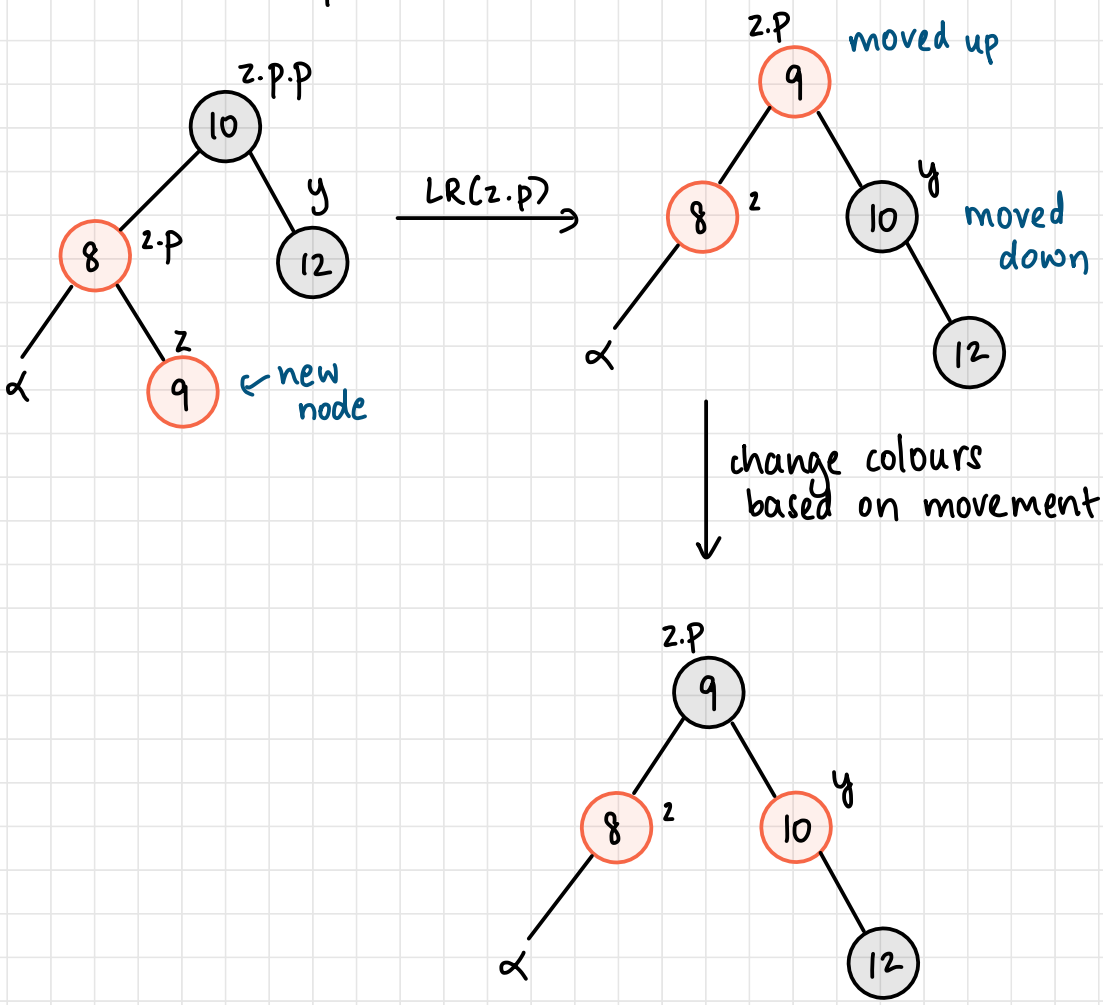
new z = z.p.p (iterate upwards)

Case 2


z.p.p
10
y
8
z.p
12
z
α
9
new node
set new z
z.p
10
y
8
z
12
α
9
L(z)
z.p.p
10
z.p
9
y
12
8
z
α
change colours
z.p.p
10
z.p
9
y
12
8
z
α
(Case 3) converted
R(z.p.p)
z.p
9
inserted node
8
z
y
10
α
12

Compact Case 2

- Note: if something goes down, turns red and if something goes up, turns black
- LR rotate (z.p)



deletion

- Black height gets affected if black node deleted

Deletion in BST

1) Leaf node - simple
2) Single child - connect child to grandparent
3) Two children - replace with inorder successor, move right child of inorder successor to its parent (leaf or single child)

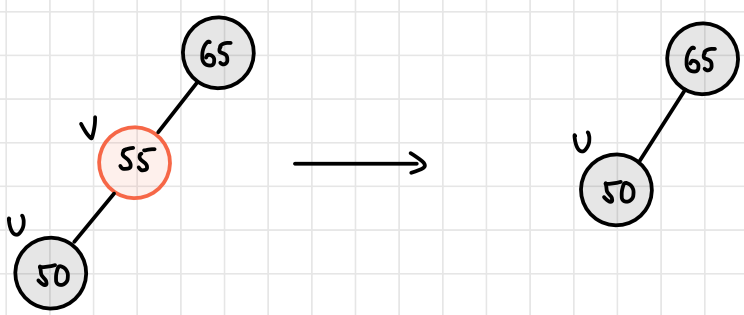
Red-Black Tree Deletion

Case 1

- one of them is red
- delete v

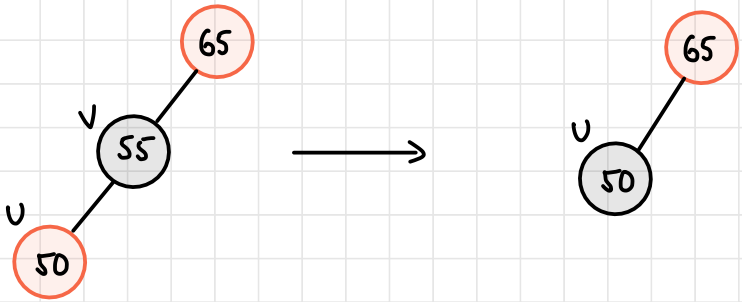
(a) v is red, u is black

- simply delete v



(b) v is black, u is red

- delete v and make u black

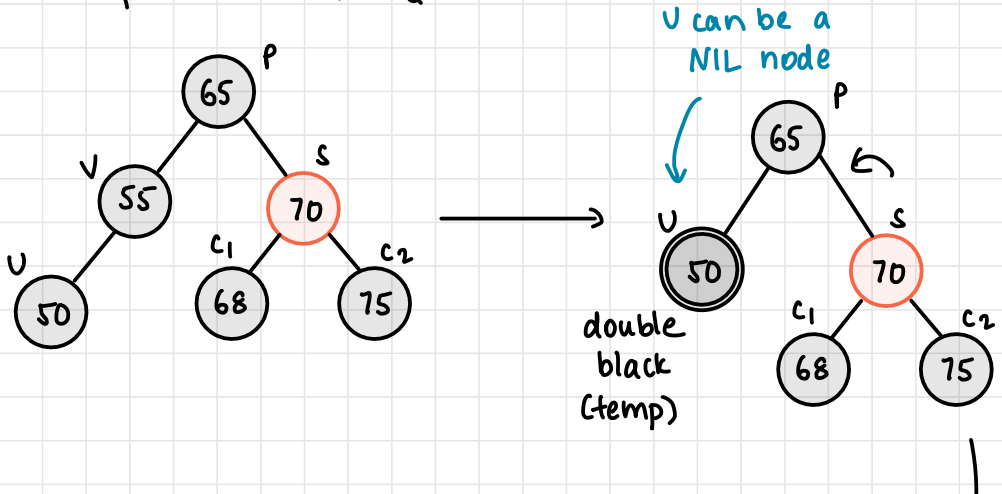


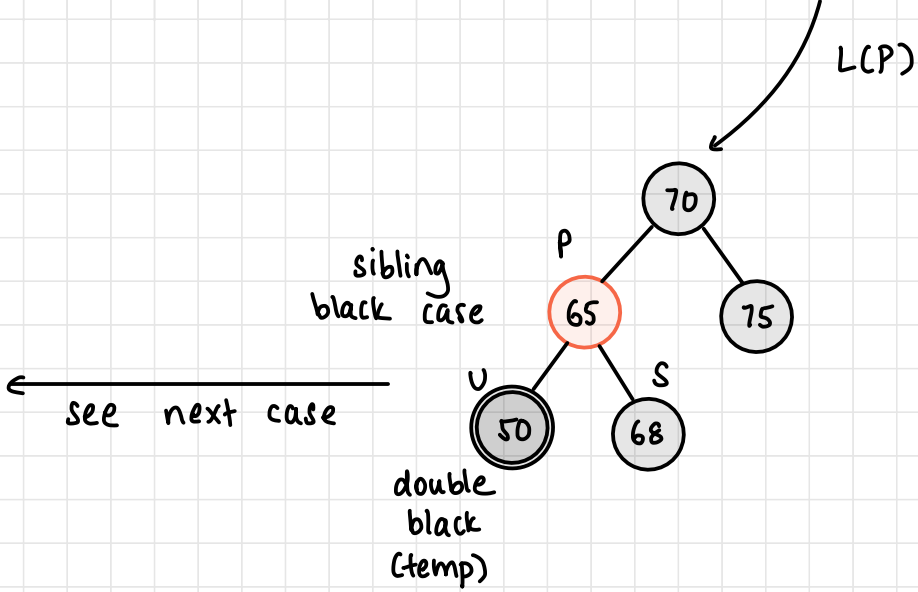
Case 2

- both u and v are black
- delete v
- one black lost in height

(a) Sibling is red

- Right case (S is right child) – L(P)
- Left case – R(P)
- Swap colours of P & S

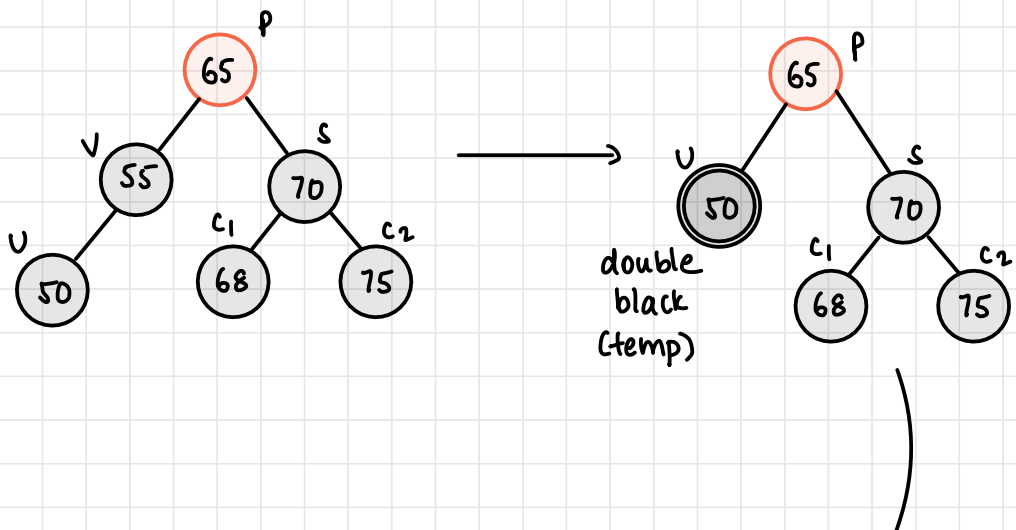


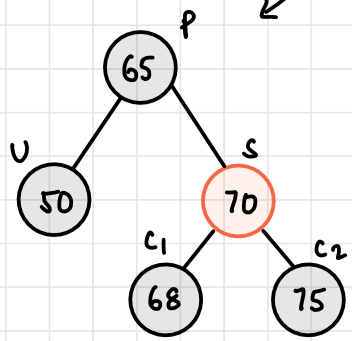


(b) Sibling is black, c_1 & c_2 black

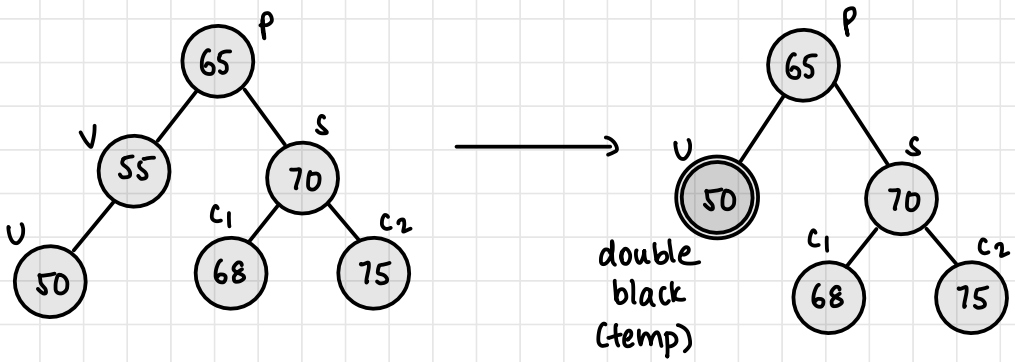
- change colour of S to red
- double black → single black
- if parent red, change to black
- if parent black, make double black and recurse

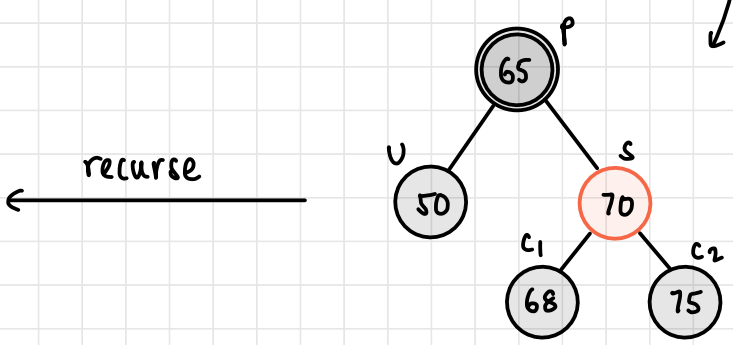
(i) Parent red





(ii) Parent black

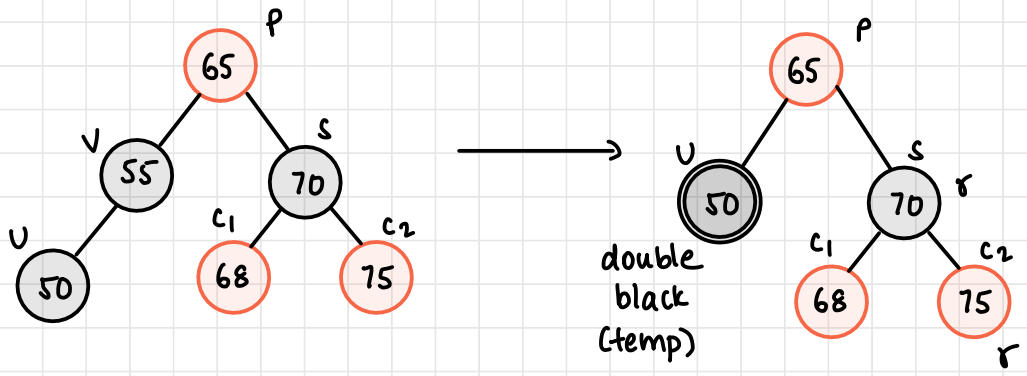


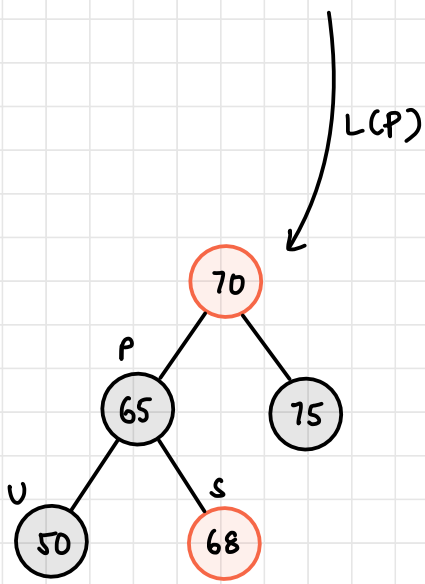


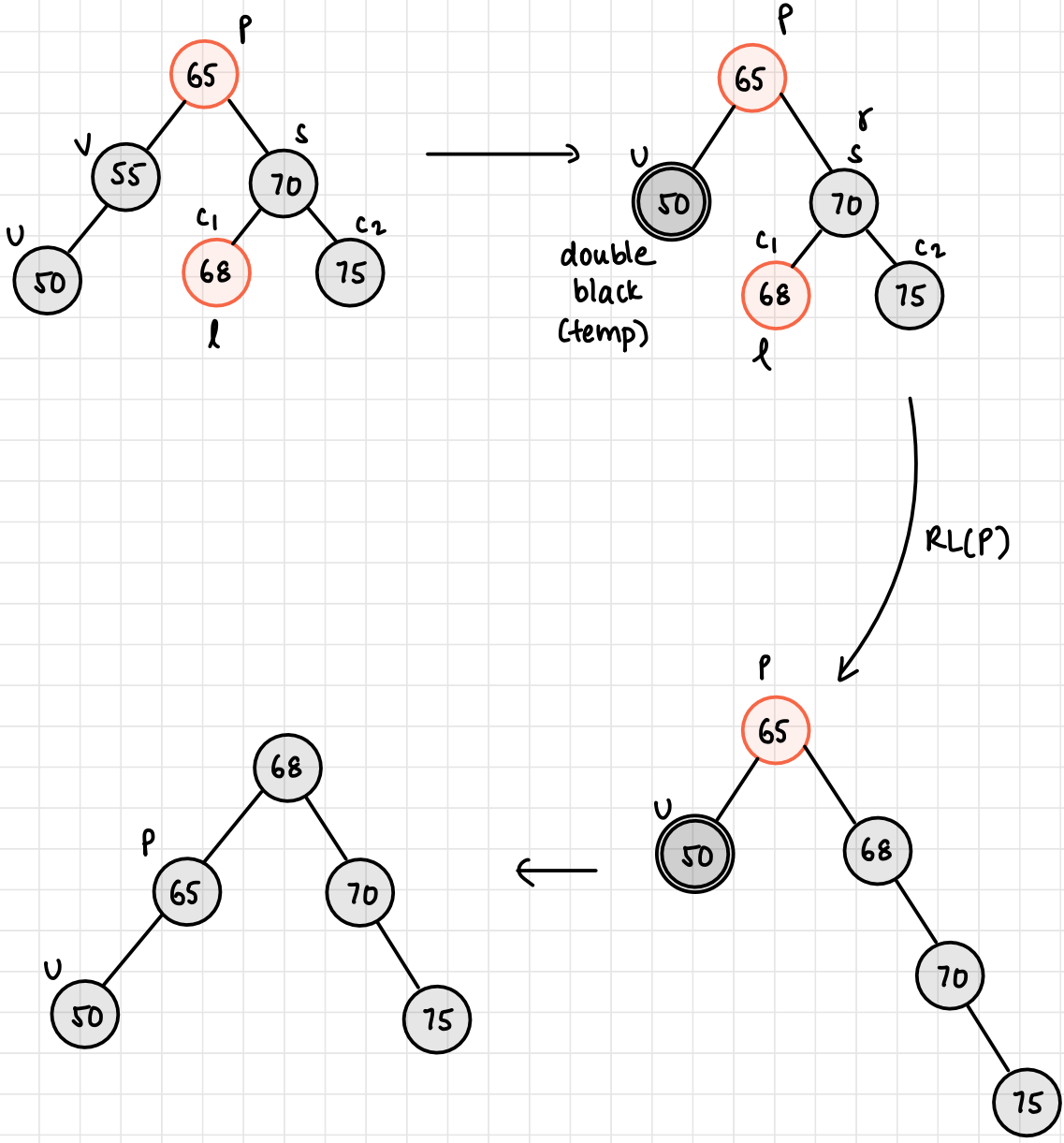
- If root becomes DB, make B

(c) Sibling is black, one or two children are red

- Mark the red child of S as r or l (r - right, l - left)
- AVL rotation conditions on P, change red child to black
- rr, lr, rl, ll
  - rr → L
  - lr → LR
  - rl → RL
  - ll → R



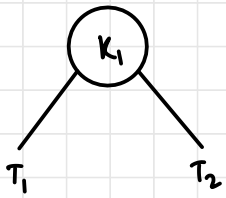



P
65
V
55
S
70
U
50
C1
68
C2
75
l
P
65
U
50
r
S
70
C1
68
C2
75
l
double
black
(temp)
RL(P)
P
65
U
50
68
70
75
68
P
65
70
U
50
75

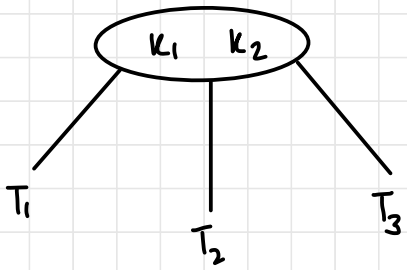
2-3 Trees

- B-tree of order 3
- Multiway balanced search tree
- Node can have 1 key (2 children) or 2 keys (3 children)
- All elements added to leaf
- All leaves at same level
- B-tree of order m ⇒ m-1 keys

2-node



3-node

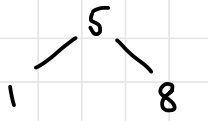


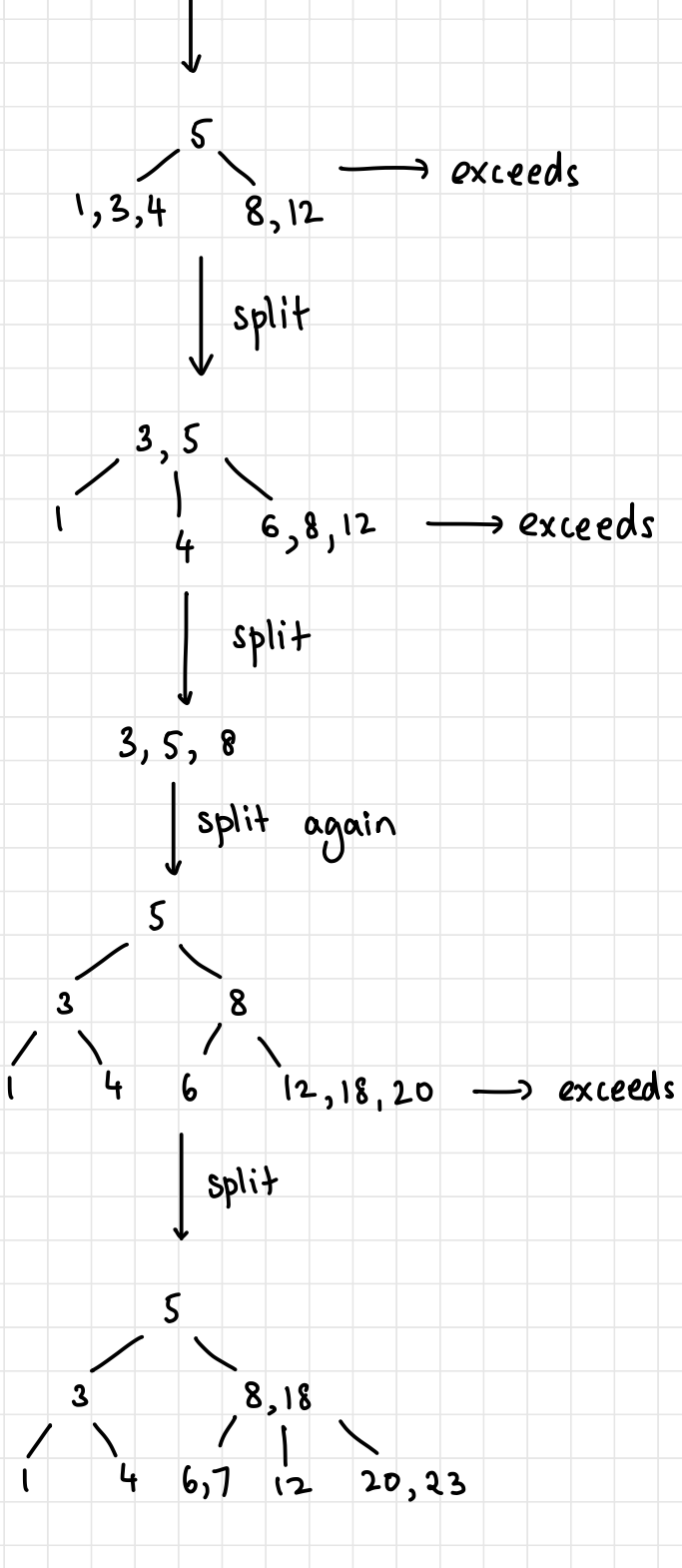
Q: Convert to 2-3 tree

8, 5, 1, 3, 12, 4, 6, 18, 20, 23, 7

1, 5, 8 ⟶ exceeds

↓ Split




5
1,3,4
8,12
exceeds
split
3, 5
1
4
6,8,12
exceeds
split
3, 5, 8
split again
5
3
8
1
4
6
12,18,20
exceeds
split
5
3
8,18
1
4
6,7
12
20,23

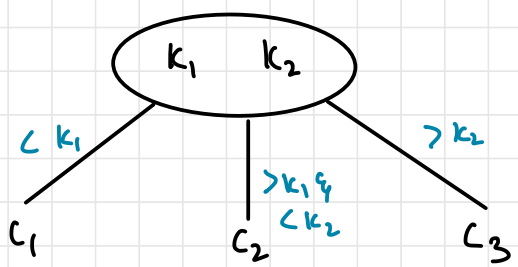
B-Trees

B-Tree of Order m

1) All leaves on same level

2) All internal nodes except the root have at most (m) non-empty children and at least $\lceil m/2 \rceil$ non-empty children, at most (m-1) keys and at least $\lceil m/2 - 1 \rceil$ keys (non-root)

3) Number of keys in each internal node is one less than the number of non-empty children and partitioning is based on search tree concept



4) Root max : m children and min : 2 children / 0 children

Deletion in B-Trees

- Deletion can be internal node or leaf node

1) Non-leaf / internal node

   - its immediate predecessor / successor will be in a leaf
   - promote immediate predecessor / successor to position of deleted node

2) Leaf node

(i) Case 1 – leaf contains keys > min no. of keys

- simply delete the key

(ii) Case 2 – leaf contains min no. of keys

- first look at two adjacent leaves (immediate) and are children of same parent
- if one of them has more than min, move key to parent and move parent to deletion position
- if adjacent leaf has only minimum number of entries, then two leaves and the median entry from parent are combined as new leaf which will contain no more than the maximum no. of entries
- If this step leaves the parent node with few entries, the process propagates upwards

- refer data structures, unit 4 – B trees