澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

CISC2006
Algorithm Design and Analysis

Xiaowei Wu

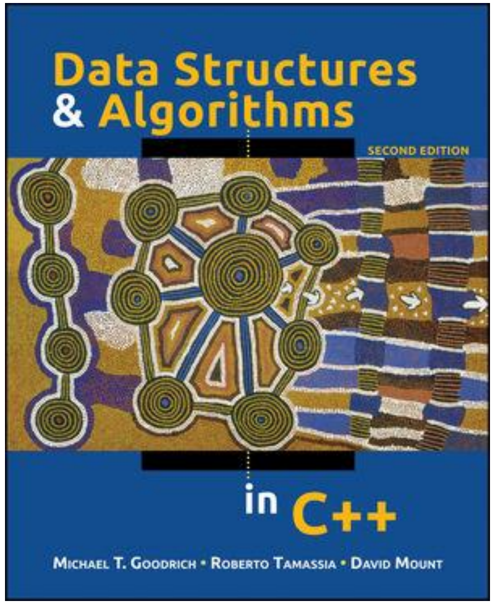
SKL of IOTSC, Department of CIS

University of Macau

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Materials

- **Textbook:**
  - Data Structures and Algorithms in C++ (2^{nd} Edition)
  - Data Structures and Algorithms in Java (6^{th} Edition)
  - by M.T. Goodrich, R. Tamassia, and M.H. Goldwasser
- Materials modified from
  - Algorithms and Data Structures 2
    by Monika Henzinger (University of Vienna)
  - Discrete Mathematics
    by T-H. Hubert Chan (University of Hong Kong)
  - Design and Analysis of Algorithms (Advanced Class)
    by Tak-Wah Lam (University of Hong Kong)



Topics (Part-1)

- Algorithm Basics (CISC2003)
  - Big-Oh, Array, Stacks, Queues, Linked Lists, Trees …
- Priority Queues and Binary Search Trees
  - Priority Queue Abstract Data Type and Implementation
  - Heap, Heapsort
  - Binary Search Trees: ADT, AVL Trees
- Sorting:
  - Merge-Sort, Quick-Sort
  - Complexity Analysis and Lower Bounds

Topics (Part-2)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Algorithm Design and Analysis Techniques
  - Recursion, Divide-and-Conquer, Greedy
  - Dynamic Programming

- Graphs:
  - Definition and Representation
  - Graph Traversals: BFS, DFS
  - Shortest Path (Dijkstra's Algorithm)
  - Minimum Spanning Tree (Kruskal's and Prim's Algorithm)

Algorithm Basics

Algorithm

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Problems**: input -> output
- **Algorithms**: a sequence of instructions that can produce the desired output from any given input.
- **Correctness**: always compute the desired output.
- **Time complexity/efficiency**:
  - how many steps: a function of the input size (usually denoted by $T(n)$, where n is the input size).
  - asymptotic behavior: the trend (i.e., when the input is large); big O notation. E.g., merge sort $O(n \log n)$ time; bubble sort $O(n^2)$ time.

Terminologies

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Problem vs. Instance**
  - E.g., sorting is a problem; array $(3,4,1,2)$ is an instance
  - E.g., MST is a problem; a graph $G(V,E)$ is an instance

- **Algorithm vs. Solution**
  - An algorithm is for solving a problem
  - A solution is to an instance

Big O notation

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- $T(n)$ be a function of n
- Big O notation: $T(n) = O(f(n))$ if

  there exist fixed positive constants c and N such that,

  for all $n \geq N$, $T(n) \leq c \cdot f(n)$.

Example:

- $T(n) = 2n^2 + 50n = O(n^2)$
- $T(n) = 5n \log n + 10\sqrt{n} = O(n \log n)$

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Big O notation

- $T(n)$ be a function of n
- Big O notation: $T(n) = O\big(f(n)\big)$ if

  there exist fixed positive constants c and N such that,

  for all $n \geq N$, $T(n) \leq c \cdot f(n)$.

Example:

- $T(n) = 2n^2 + 50n = O(n^3)$
- $T(n) = 5n\log n + 10\sqrt{n} = O(n^2)$

Big O notation

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- $T(n)$ be a function of n
- Big O notation: $T(n) = O\big(f(n)\big)$ if

  there exist fixed positive constants c and N such that,

  for all $n \geq N$, $T(n) \leq c \cdot f(n)$.

- Big O notation reflects the asymptotic behavior:

  the complexity trend when the input is large.
- Big O only guarantees an **upper bound.**

Big Ω notation

- $T(n)$ be a function of n
- Big Ω notation: $T(n) = \Omega\big(f(n)\big)$ if

  there exist fixed positive constants c and N such that,

  for all $n \geq N$, $T(n) \geq c \cdot f(n)$.

Example:

- $T(n) = 2n^2 + 50n = \Omega(n^2)$
- $T(n) = 5n \log n + 10\sqrt{n} = \Omega(n)$

Big Ω notation

- $T(n)$ be a function of n
- Big Ω notation: $T(n) = \Omega(f(n))$ if

  there exist fixed positive constants c and N such that,

  for all $n \geq N$, $T(n) \geq c \cdot f(n)$.

- Big Ω only guarantees a **lower bound.**

Big Θ notation

- $T(n)$ be a function of n
- **Big Θ notation:** $T(n) = \Theta(f(n))$ if

  $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

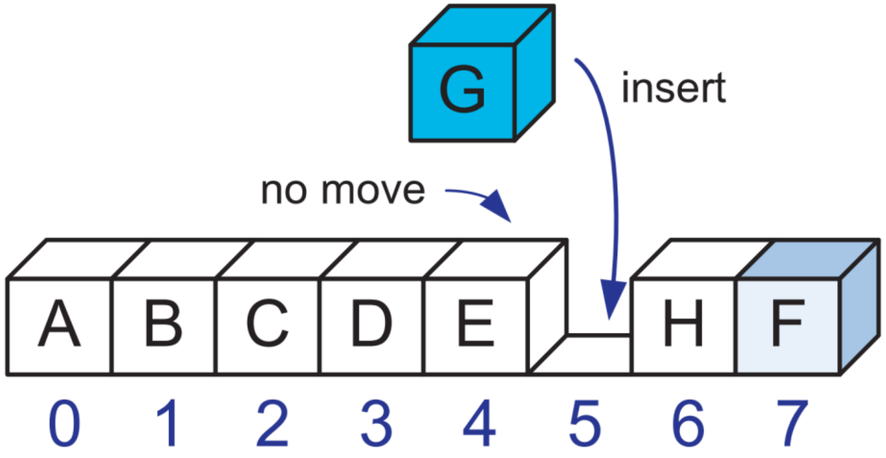
Example:

- $T(n) = 2n^2 + 50n = \Theta(n^2)$
- $T(n) = 5n\log n + 10\sqrt{n} = \Theta(n\log n)$
- $T(n) = n^3 + 10n\log^5 n = \Theta(n^3)$

Array

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

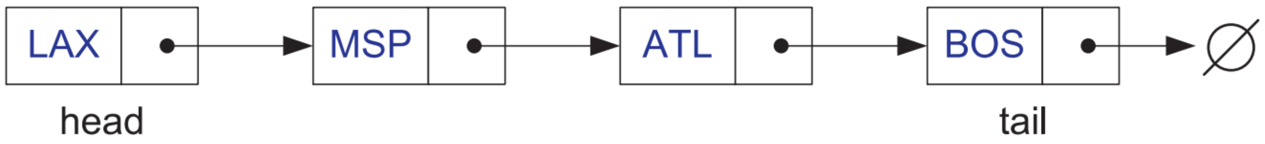
- An array data structure, or simply an array, is a data structure consisting of a collection of elements, each identified by at least one array index or key.
- Modify the "i-th element" in $O(1)$ time



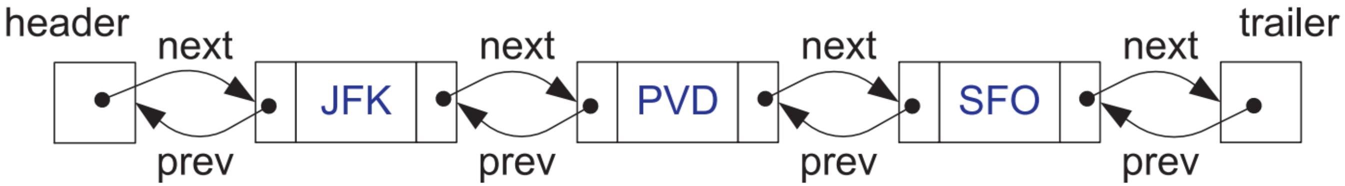
Linked List

- A linked list is a collection of nodes that together form a linear ordering.
- Modify the “i-th element” in $O(i)$ time
- Insert an element at a given position in $O(1)$ time
- Merging two linked lists in $O(1)$ time

- Singly linked list



- Doubly linked list



Stack

- A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.

- Stack Abstract Data Type (ADT):
  - push(e): insert element e
  - top(): return top element
  - pop(): return and remove top element
  - size(): return size of the stack
- Can be implemented by array/linked-list

Queue

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- A queue is a container of objects that are inserted and removed according to the first-in-first-out (FIFO) principle.

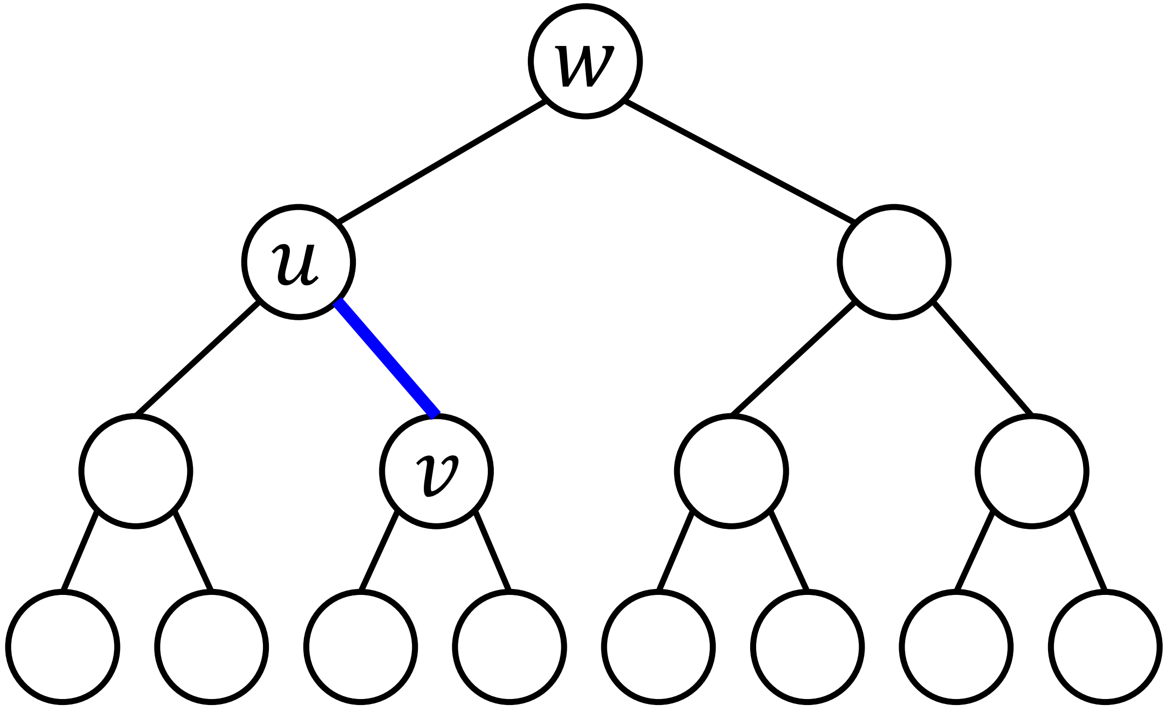
- Queue ADT:
    - enqueue(e): insert element e
    - front(): return front element
    - dequeue(): remove front element
    - size(): return size of the queue
- Can be implemented by array/linked-list

Trees

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Most important nonlinear data structures
- Nodes: objects
- Edges: relations

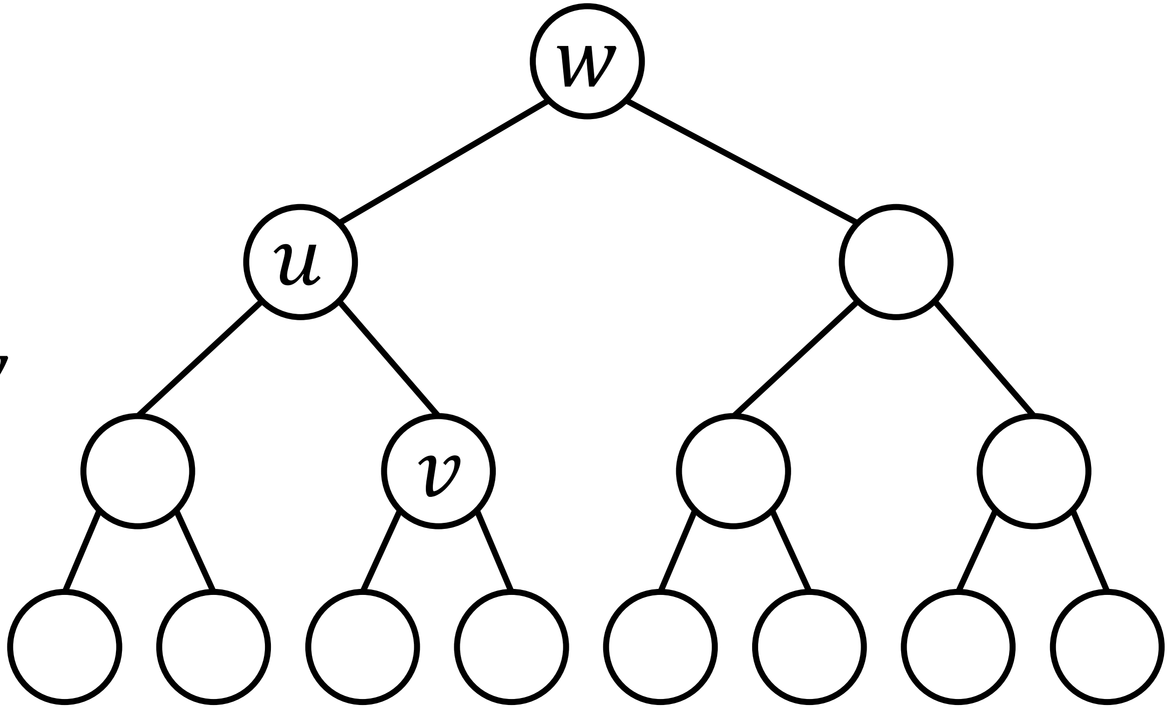
- Node u is parent of node v
- Node v is child of node u



Trees

- Most important nonlinear data structures
- Nodes: objects
- Edges: relations

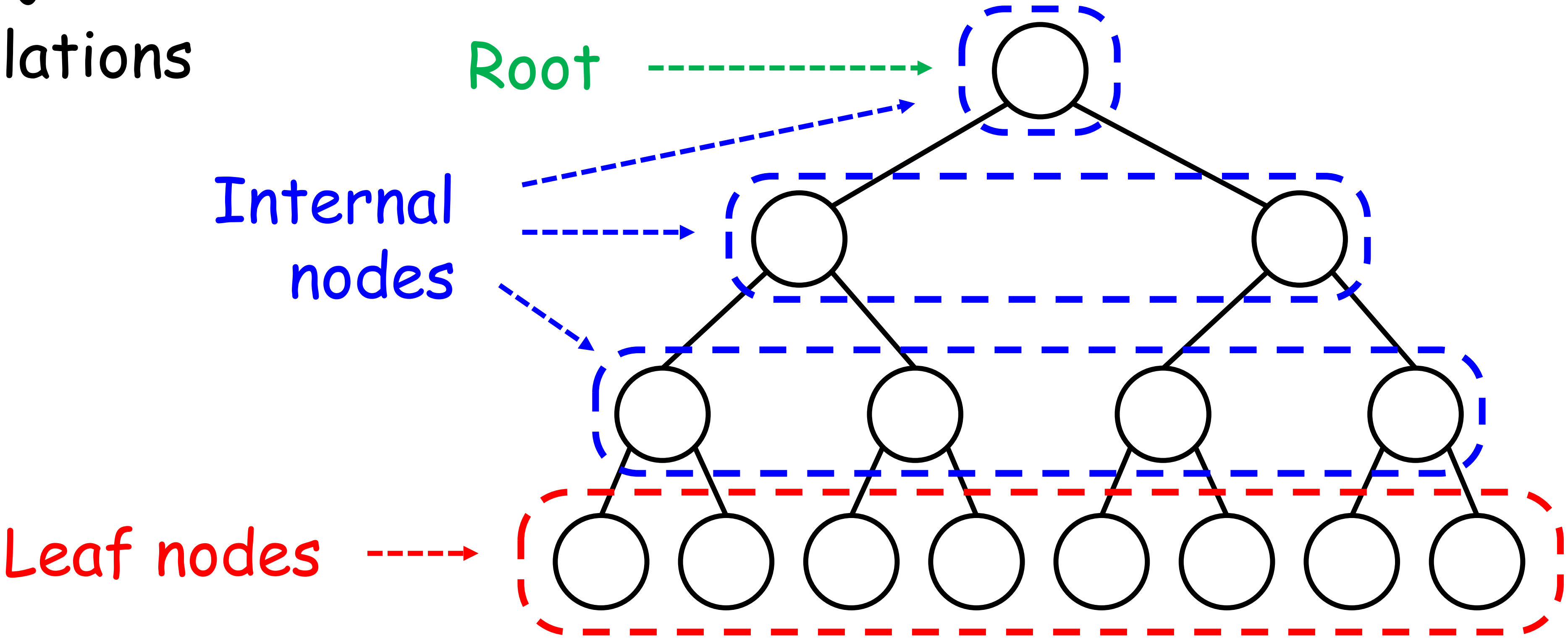
- Node w is an ancestor of v
- Node v is a descendent of w



Trees

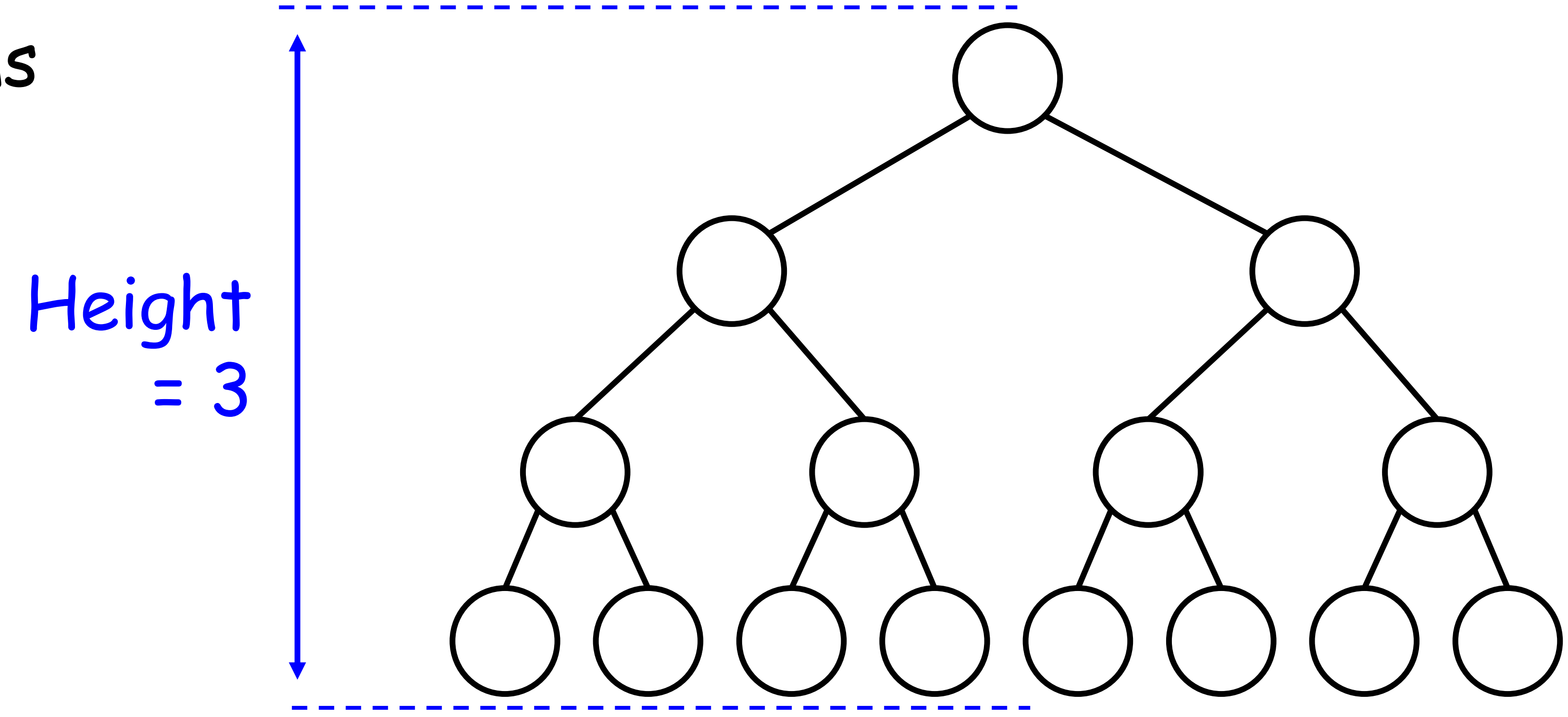
澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Most important nonlinear data structures
- Nodes: objects
- Edges: relations



Trees

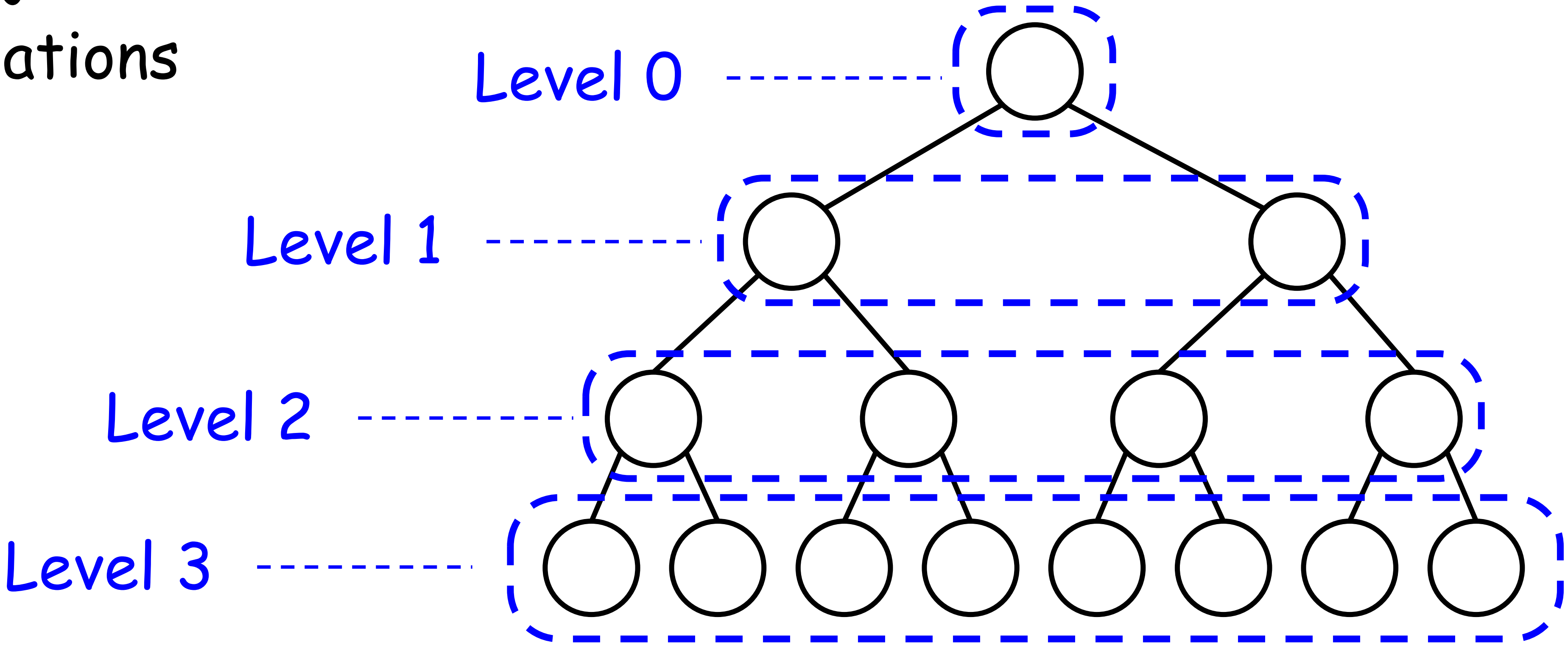
- Most important nonlinear data structures
- Nodes: objects
- Edges: relations



澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Trees

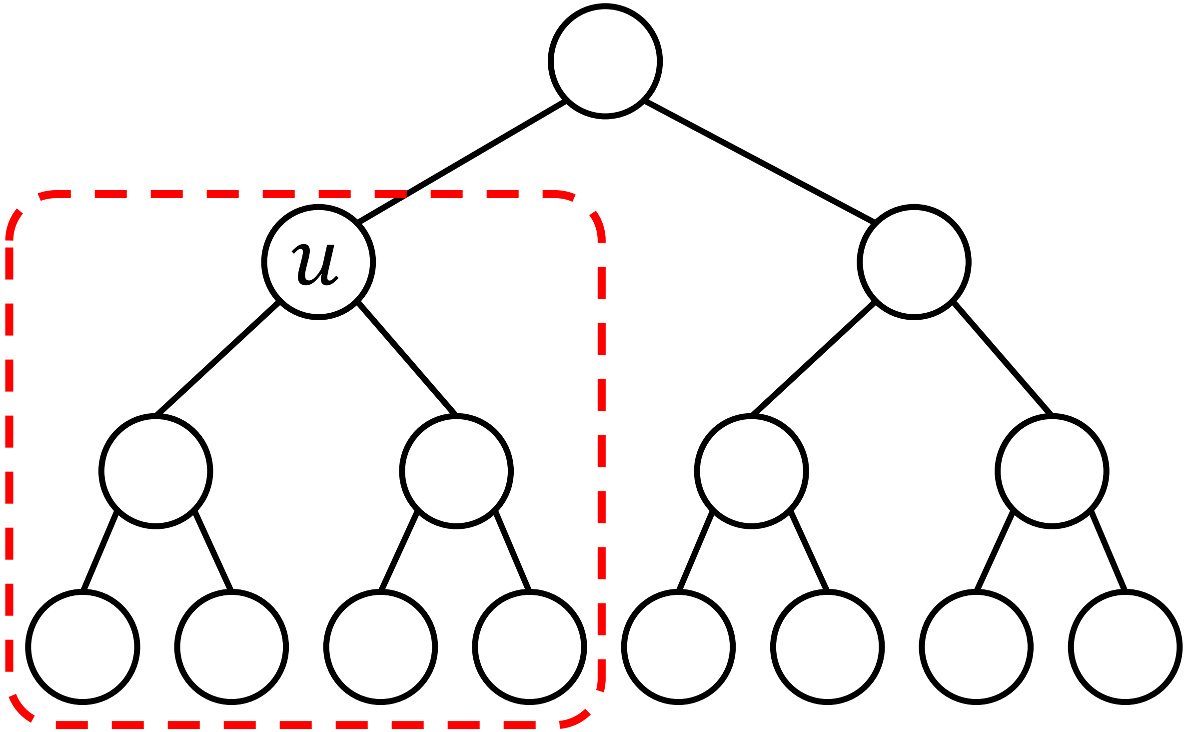
- Most important nonlinear data structures
- Nodes: objects
- Edges: relations



Trees

- Most important nonlinear data structures
- Nodes: objects
- Edges: relations

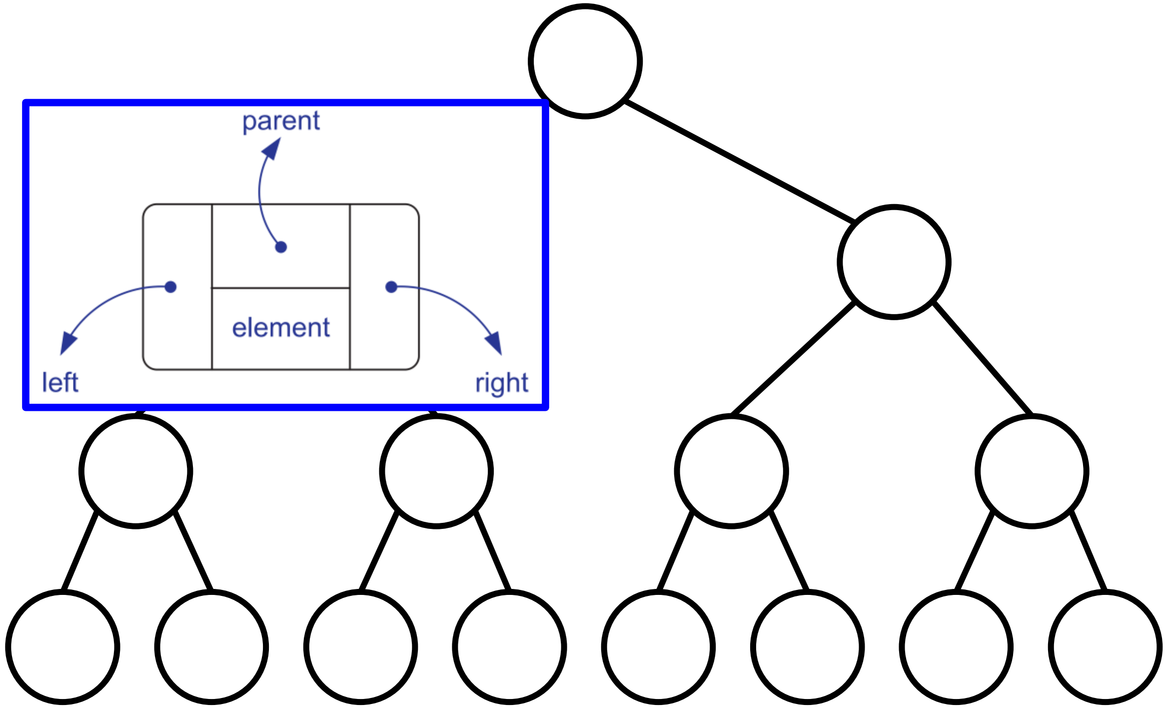
Subtree rooted at node u



Trees

- Most important nonlinear data structures

- Binary Tree ADT
    - size()
    - root()
    - p.parent()
    - p.left()
    - p.right()



Binary Search

- Given a sorted array,

1	3	6	7	9	11	12	16	19	23	24	26	28

> 9

- Task: search for a key value 9

Binary Search

- Given a sorted array,

1	3	6	7	9	11	12	16	19	23	24	26	28

< 20

- Task: search for a key value 20

Binary Search

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Given a sorted array,

1	3	6	7	9	11	12	16	19	23	24	26	28

Output: Does not exist!

< 20

- Task: search for a key value 20
- Complexity: $O(\log n)$ for sorted array of length n

澳 門 大 學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Recursion

- Given a sorted array,

1	3	6	7	9	11	12	16	19	23	24	26	28

Search(L)

> 9

- Task: search for a key value 9

Recursion

- Given a sorted array,

1	3	6	7	9	11	12	16	19	23	24	26	28

< 20

Search(R)

- Task: search for a key value 20
- Complexity: $T(n) = T(n/2) + O(1) = O(\log n)$

Recursion

- During the execution of a function, a call to itself (with a smaller input) is made.

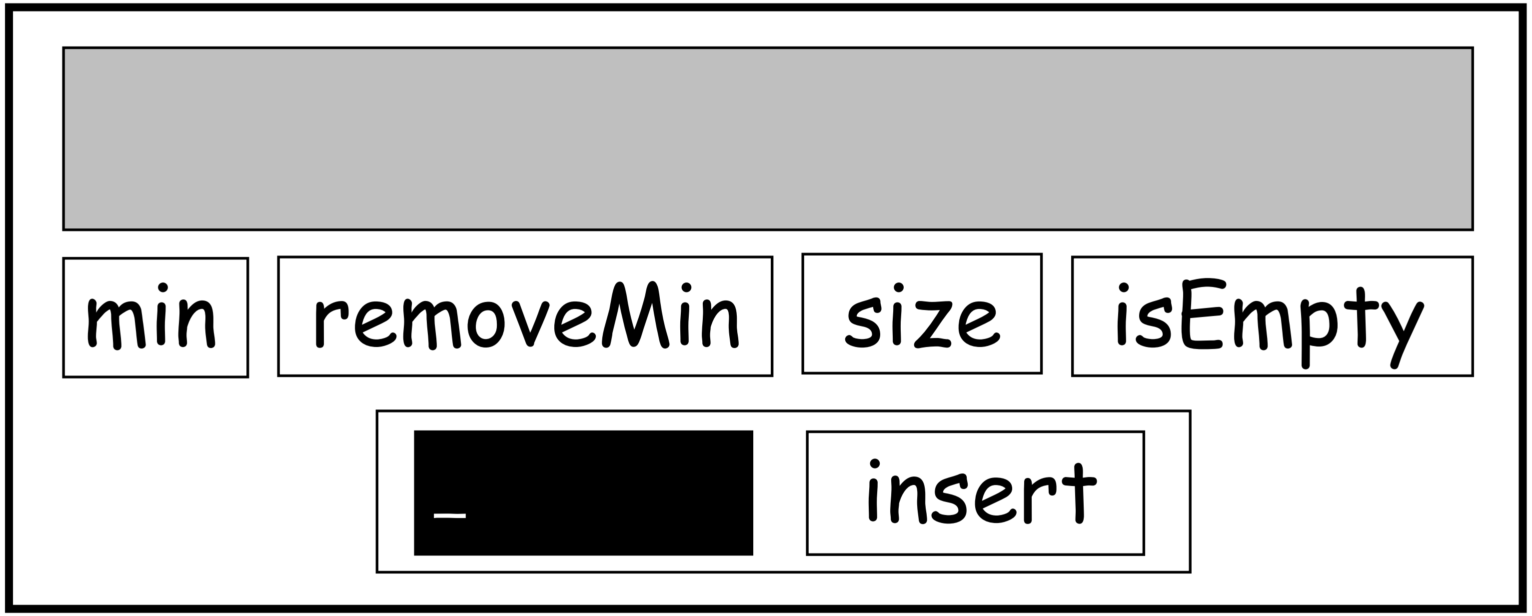
- Don't forget about…
  - Base case and boundary cases
  - Avoid infinite loops
  - Transforming solutions for the subproblems to the original problem

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Priority Queue

Priority Queue

- An abstract data type storing a set of prioritized elements
- Supports arbitrary element insertion
- Supports removal of the elements of highest priority



Priority Queue

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Priority Queue ADT (for priority queue P):
  - insert(e): insert element e to P
  - min(): return the minimum element of P
  - removeMin(): remove the minimum element of P
  - size(): return the number of elements in P
  - isEmpty(): return True if P is empty
- Extensions: general elements, e.g., e = (key, value, ...)
  - e.g., for storing multi-dimensional data
  - Need to define the comparator (to compare elements)

Sorting with Priority Queue

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Given a sequence of numbers $A = (a_1, \dots, a_n)$, sort the numbers into $B = (b_1, \dots, b_n)$ such that $b_1 \le b_2 \le \cdots \le b_n$
- Sorting_with_Priority_Queue:
  - initialize an empty priority queue P
  - for $i = 1,2, \dots, n$:
    - $P.\mathrm{insert}(a_i)$
  - for $i = 1,2, \dots, n$:
    - $b_i \leftarrow P.\mathrm{min}()$
    - $P.\mathrm{removeMin}()$

Sorting with Priority Queue

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Given a sequence of numbers $A = (a_1, \dots, a_n)$, sort the numbers into $B = (b_1, \dots, b_n)$ such that $b_1 \le b_2 \le \cdots \le b_n$
- Sorting_with_Priority_Queue:
  - suppose $P.\mathrm{insert}(e)$ takes T_i time
  - suppose $P.\mathrm{min}()$ takes T_m time
  - suppose $P.\mathrm{removeMin}()$ takes T_r time

  - Complexity: $n \cdot (T_i + T_m + T_r)$

Implementation

- How do we store the elements in P ?
  - Unsorted list
  - Sorted list
- How would it affect the performance?
- Example:
  - inserting elements (7,4,8,2,5,3,9) one-by-one
  - output and remove minimums one-by-one

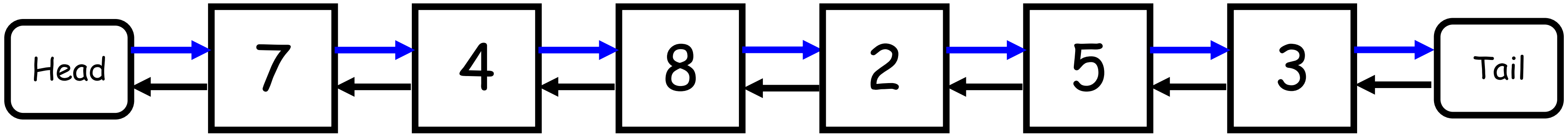
Implementation

- **Unsorted list:**
- insertion is easy
- output is difficult

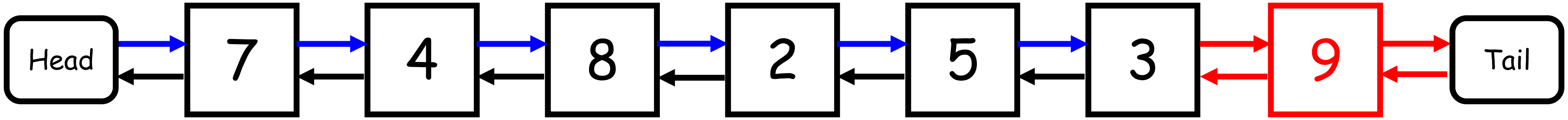
		List L	Priority Queue P
Input		(7,4,8,2,5,3,9)	()
Phase 1	(a)	(4,8,2,5,3,9)	(7)
	(b)	(8,2,5,3,9)	(7,4)
	⋮	⋮	⋮
	(g)	()	(7,4,8,2,5,3,9)
Phase 2	(a)	(2)	(7,4,8,5,3,9)
	(b)	(2,3)	(7,4,8,5,9)
	(c)	(2,3,4)	(7,8,5,9)
	(d)	(2,3,4,5)	(7,8,9)
	(e)	(2,3,4,5,7)	(8,9)
	(f)	(2,3,4,5,7,8)	(9)
	(g)	(2,3,4,5,7,8,9)	()

Implementation (linked list)

- **Unsorted list:**

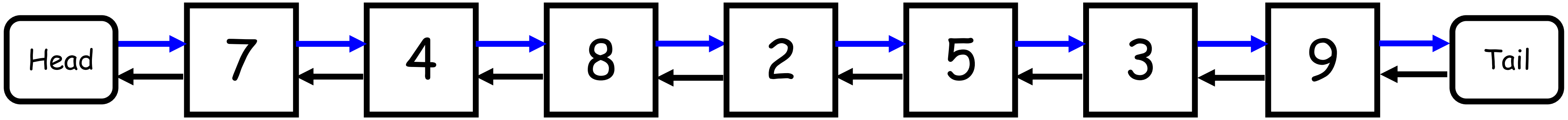


- insert(9) is easy ($O(1)$ time)

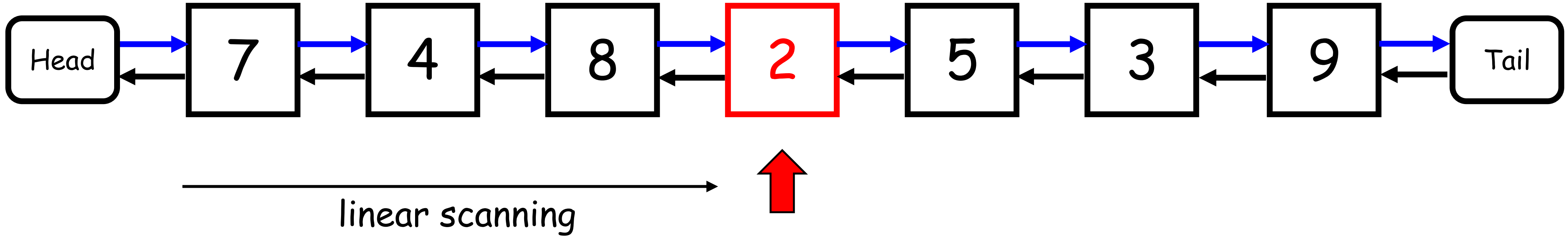


Implementation (linked list)

- **Unsorted list:**



- min() and removeMin() is difficult ($O(n)$ time)



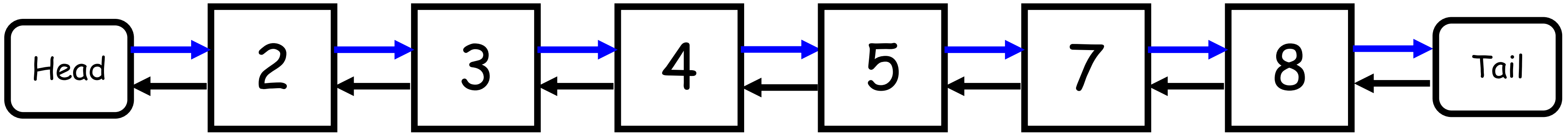
Implementation

- **Sorted list:**
- insertion is difficult
- output is easy

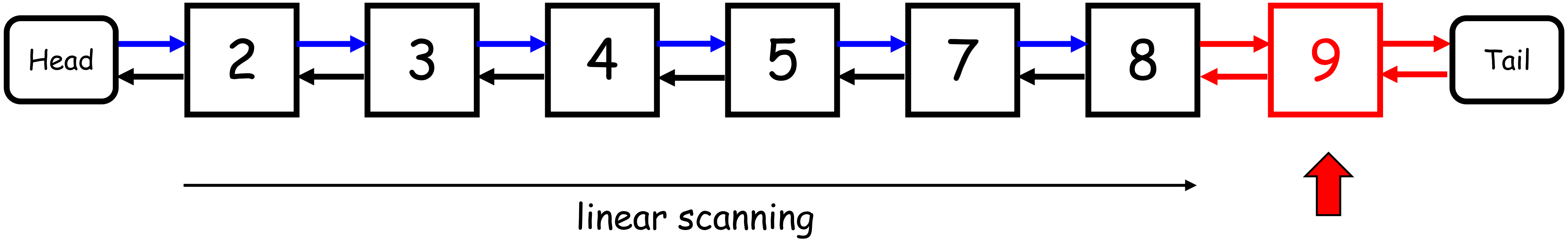
		List L	*Priority Queue P*
Input		(7,4,8,2,5,3,9)	()
Phase 1	(a)	(4,8,2,5,3,9)	(7)
	(b)	(8,2,5,3,9)	(4,7)
	(c)	(2,5,3,9)	(4,7,8)
	(d)	(5,3,9)	(2,4,7,8)
	(e)	(3,9)	(2,4,5,7,8)
	(f)	(9)	(2,3,4,5,7,8)
	(g)	()	(2,3,4,5,7,8,9)
Phase 2	(a)	(2)	(3,4,5,7,8,9)
	(b)	(2,3)	(4,5,7,8,9)
	⋮	⋮	⋮
	(g)	(2,3,4,5,7,8,9)	()

Implementation (linked list)

- **Sorted list:**

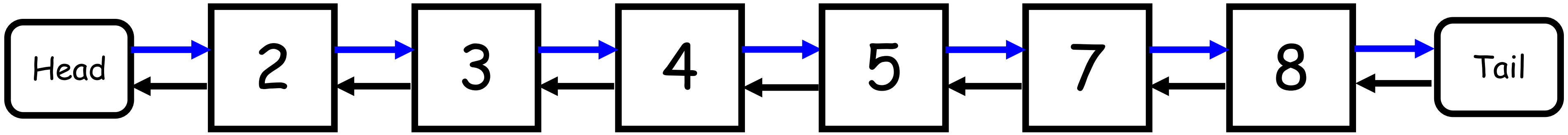


- insert(9) is difficult ($O(n)$ time)

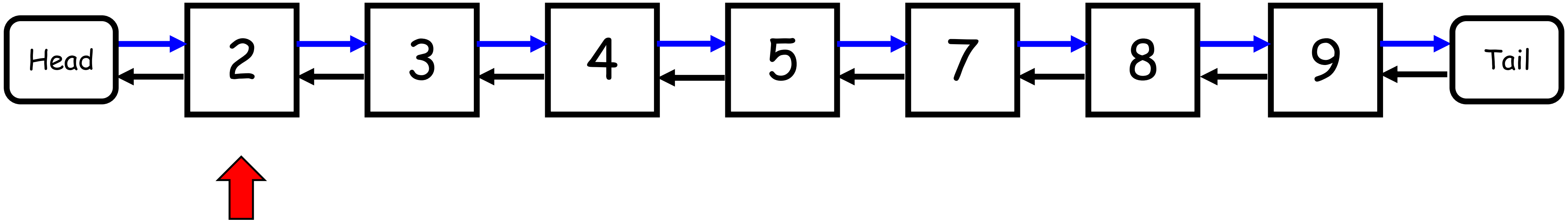


Implementation (linked list)

- **Sorted list:**



- min() and removeMin() is easy ($O(1)$ time)



Implementation (linked list)

- Complexity:

Operation	*Unsorted List*	*Sorted List*
size, empty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min, removeMin	$O(n)$	$O(1)$

- Sorting complexity: $O(n^2)$ for both implementation
  - Sorting requires inserting and removing n elements

Heaps

Heap

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- An efficient implementation of priority queue
- Store the n elements of priority queue in a tree structure
- Supports
  - size(), min() in $O(1)$ time
  - removeMin(), insert(e) in $O(\log n)$ time
- Heap-sort: $O(n \log n)$ time

Heap

- Data Structure: Complete binary tree T in which each node corresponds to one element of the priority queue.
  - Height of the tree: h
  - All levels $i \in \{0,1,\ldots,h-1\}$ are full (has 2^i nodes)
  - Nodes at level h fill this level from left to right.

- Heap-Order Property: for every node $v \neq$ root, the key associated with $v \geq$ key associated with v's parent.
  - Root = overall minimum (Min-Heap)

Heap

Lemma. Heap T storing n elements has height $h = \lfloor \log n \rfloor$.

Proof. Given a heap storing n elements and has height h:

- #elements at level i: 2^i (for all $0 \le i \le h-1$)
- #elements at level h: $n - \left(1 + 2 + \cdots 2^{h-1}\right) \in [1, 2^h]$

$$\Leftrightarrow \quad 1 \le n - \left(2^h - 1\right) \le 2^h$$

$$\Leftrightarrow \quad 2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$$

$$\Leftrightarrow \quad h \le \log n < h + 1 \quad \Leftrightarrow \quad h = \lfloor \log n \rfloor \qquad \blacksquare$$

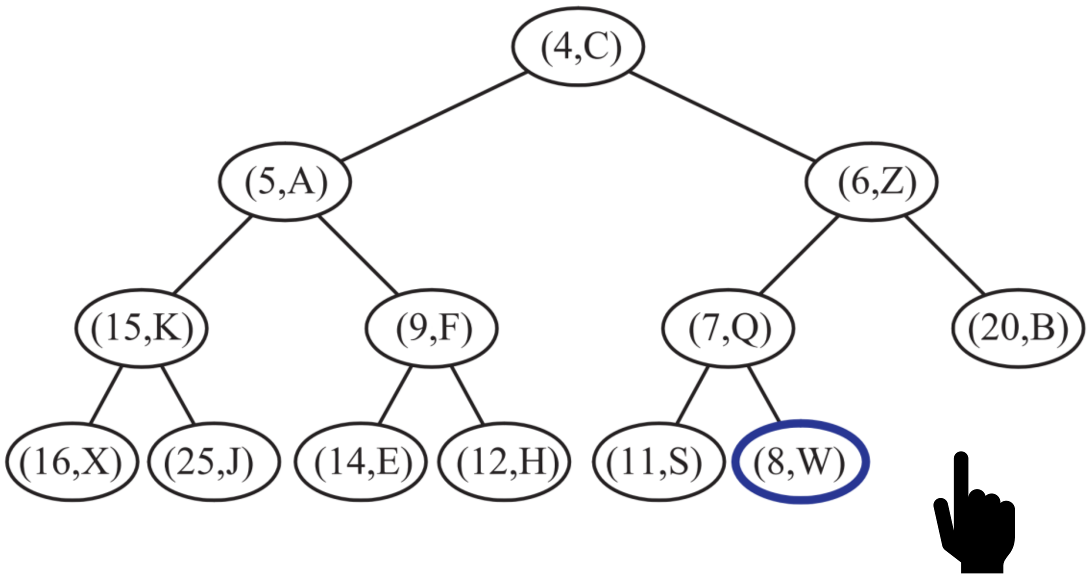
Heap

Operations of Heap (with height $= h$)

- size(): $O(1)$ time
- min(): $O(1)$ time
- insert(e): $O(h)$ time
    - ❖maintain complete binary tree and heap-order property
- removeMin(): $O(h)$ time
    - ❖maintain complete binary tree and heap-order property

Insert(e)

Maintenance of complete binary tree



Insert(e)

- Create a new node u to the right of the last node
  - If the last level is full, create a new level
- Set u.key $\leftarrow$ e.key and u.value $\leftarrow$ e.value
- **while**(u $\neq$ root and u.key $<$ u.parent.key)
  - swap keys and values of u and u.parent
  - u $\leftarrow$ u.parent

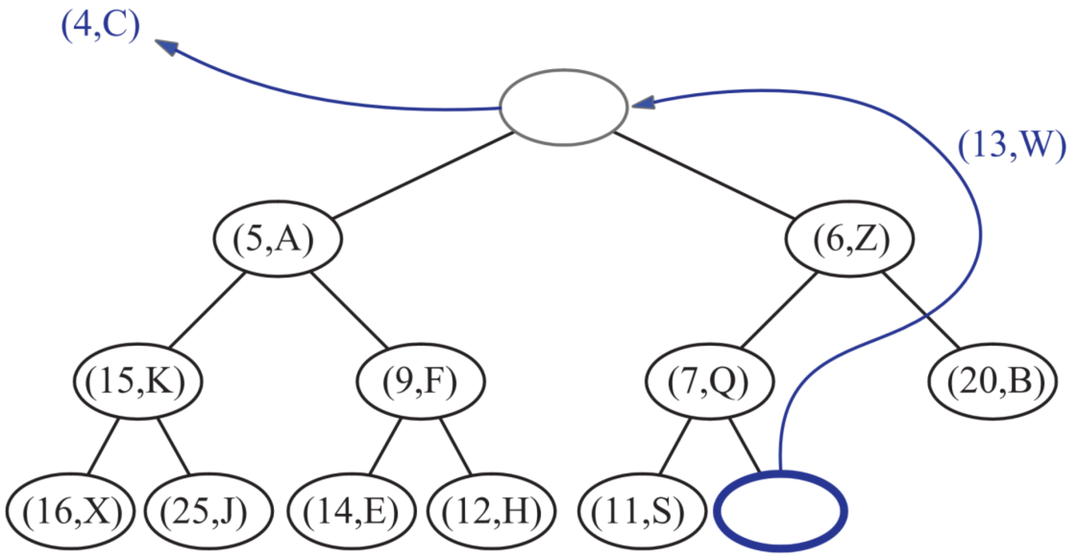
Up-heap Bubbling

Observation: at most h swaps are necessary

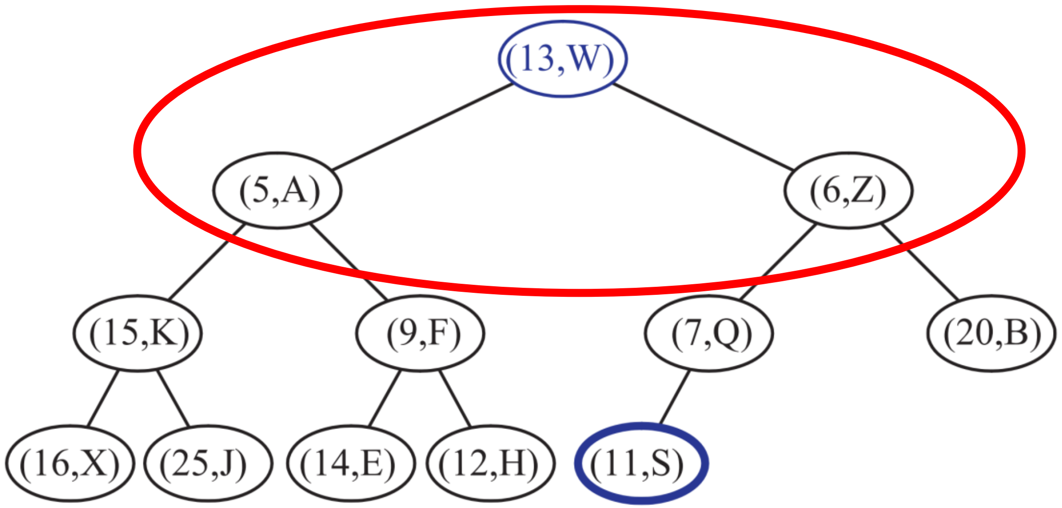
Lemma: insertion of elements can be implemented in $O(h)$ time

RemoveMin()

Maintenance of complete binary tree



move the last node to the root



need some fix to maintain the heap-order property !

RemoveMin()

- Let u point to the last node
- Set root.key ← u.key and root.value ← u.value
- Remove the last node (u points to) and let u point to the root
- **while**(u has child v such that v.key < u.key)
  - swap keys and values of u and child v of u with smaller value
  - u ← v

Down-heap Bubbling

Observation: at most h swaps are necessary

Lemma: removal of min can be implemented in $O(h)$ time

Heap-sort

Sort a given an array $A = (a_1, a_2, \ldots, a_n)$ using heap

- create an empty heap and array $B = (b_1, b_2, \ldots, b_n)$
- **for** $i = 1,2, \ldots, n$
  - insert(a_i)
- **for** $i = 1,2, \ldots, n$
  - $b_i \leftarrow$ min()
  - removeMin()
- **output** B

Complexity: $O(n \log n)$

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Binary Search Tree

Binary Search Tree

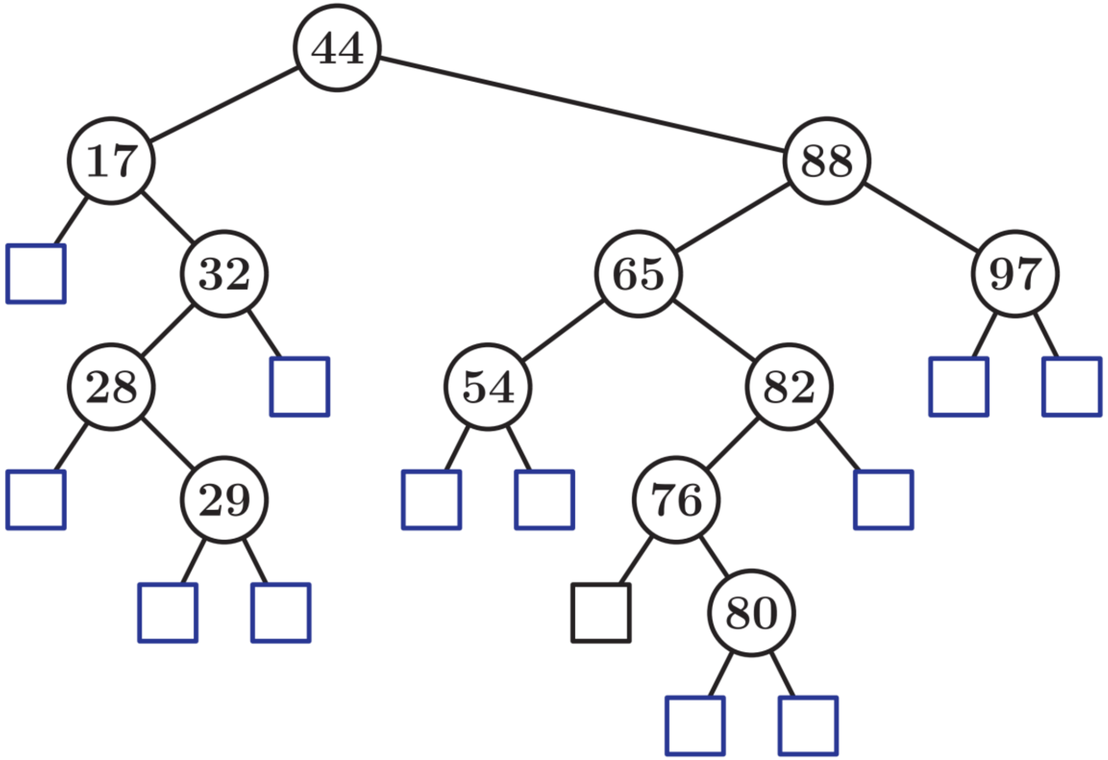
澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Binary tree that supports efficient searching of elements
- Binary Search Tree (BST) supports
  - insert(e) insert an element, e.g., e = (key, value)
  - find(key) find element e with e.key = key
  - remove(key) remove the element e with e.key = key
  - remove(p) remove the element e point p points to

- **Application**: searching in a dictionary.

Binary Search Tree

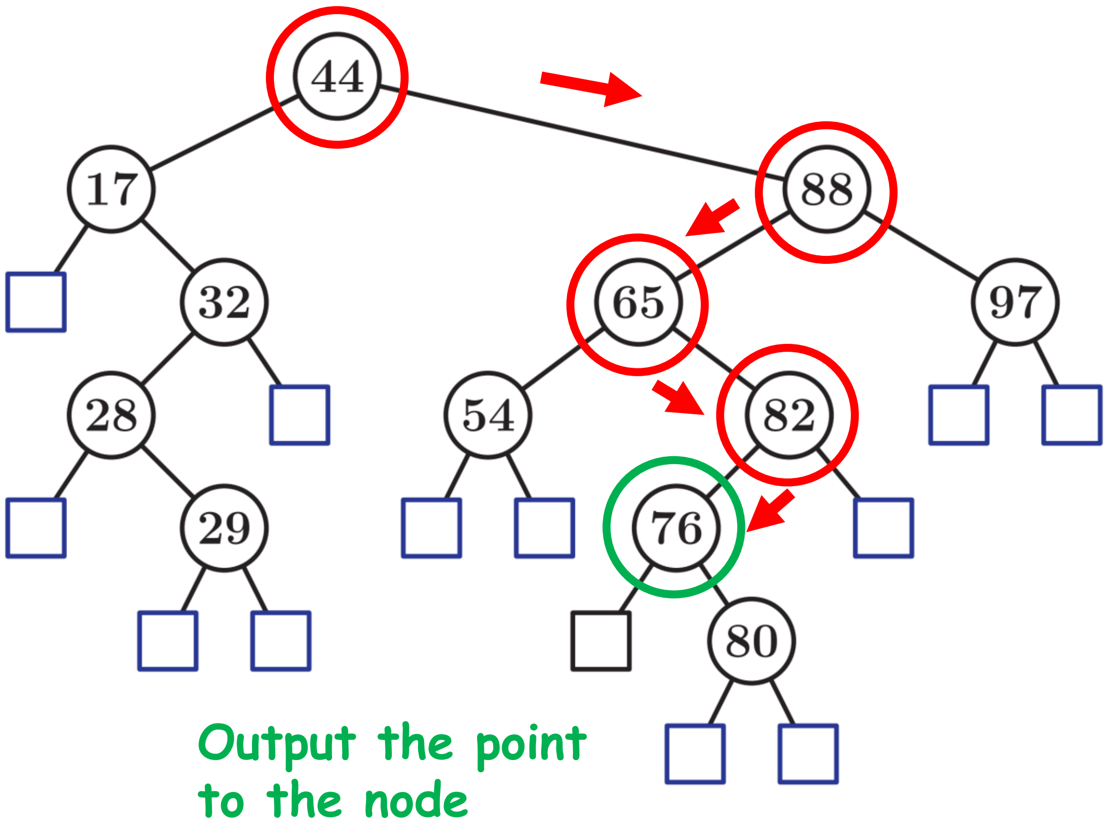
- A binary tree such that each internal node v stores an entry with key = k such that
  - Keys on left subtree are $\leq k$
  - Keys on right subtree are $\geq k$



Binary Search Tree

- Useful when searching target:
  - Go left if target < $u.key$
  - Go right if target > $u.key$
  - Fail if target $\neq u.key$ and u is a left node.

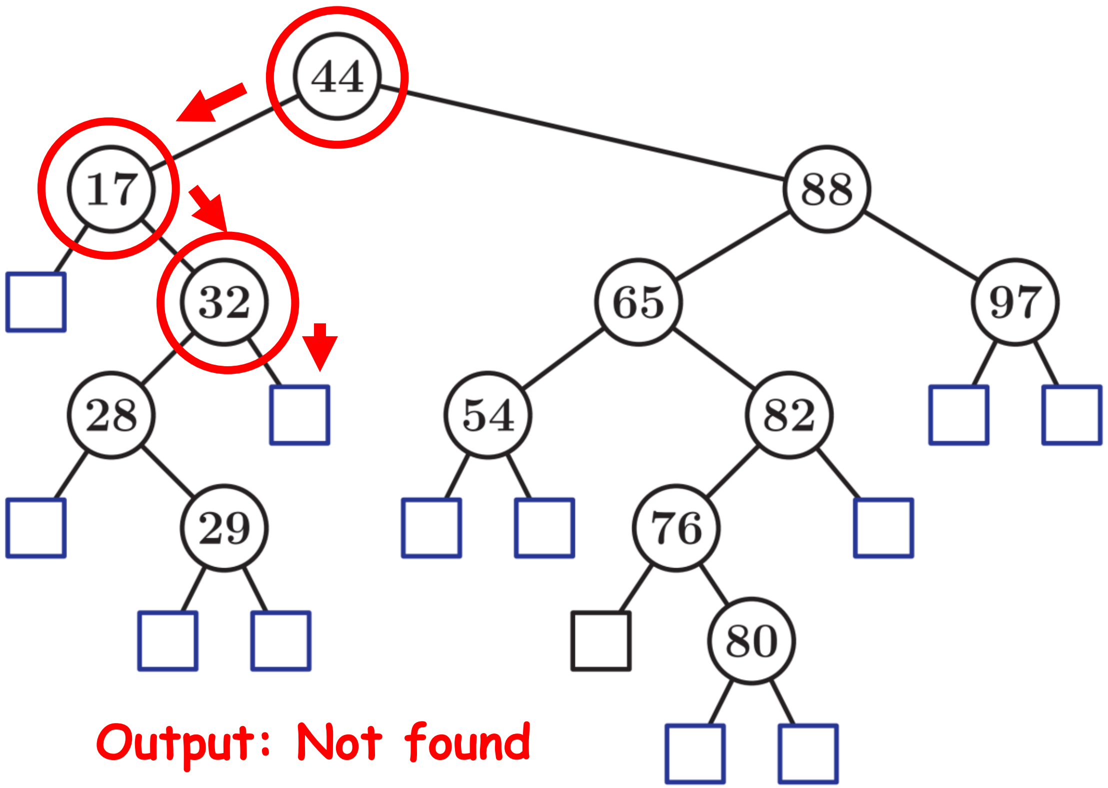
- Example: searching for 76



Binary Search Tree

- Useful when searching target:
  - Go left if target < $u.key$
  - Go right if target > $u.key$
  - Fail if target $\neq u.key$ and u is a left node.

- Example: searching for 40



Binary Search Tree

Function to search for k in the subtree rooted at v

- TreeSearch($k,\ v$)
  - if $v = null$ then
    - Return: v
  - else if $v.key = k$ then
    - Return: v
  - else if $v.key > k$ then
    - Return TreeSearch($k,\ v.left$)
  - else
    - Return TreeSearch($k,\ v.right$)

Complexity: $O(h)$

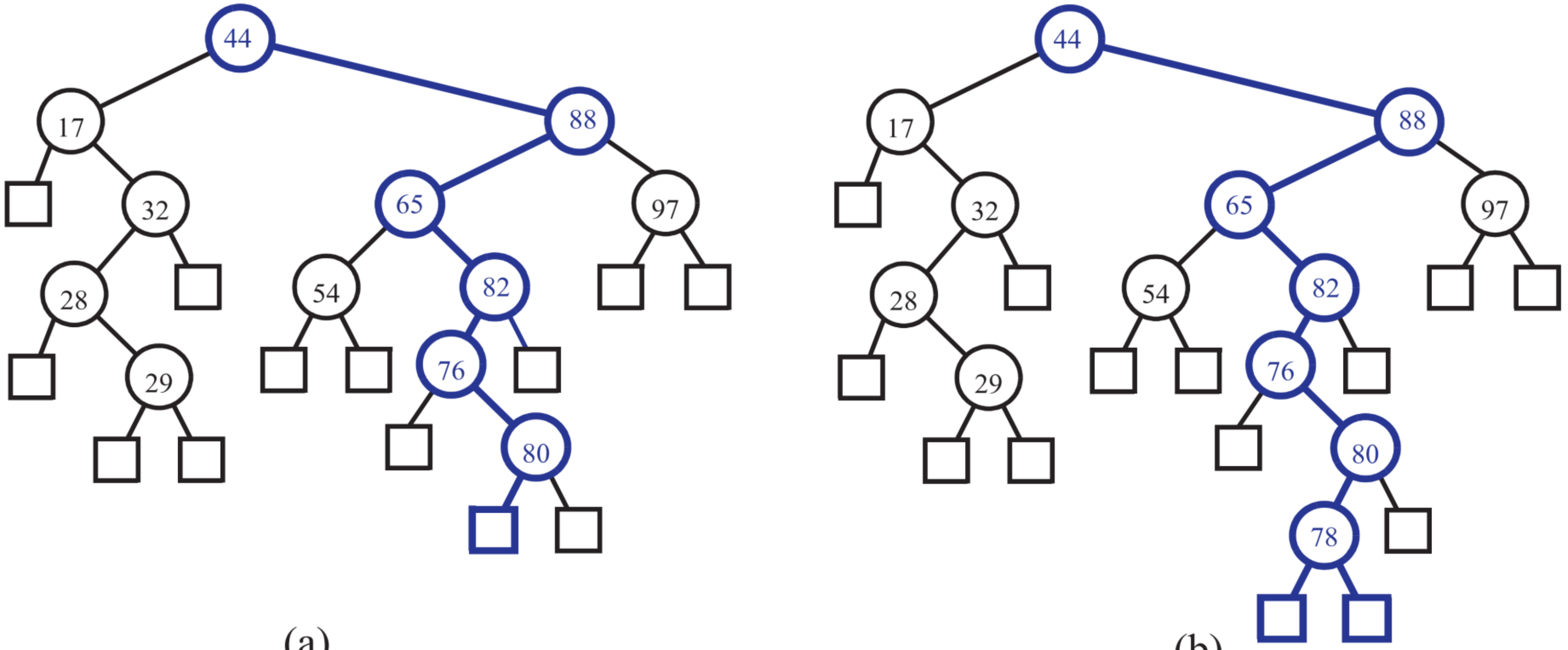
- Every recursion increases the level by one

Extension: Find-All(k, v)

- Output all matched element

Insertions

Insertion of element with key 78



(a) (b)

Insertions

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Function to insert $e = (key, value)$ to the subtree rooted at v

- TreeInsert(e, v)
  - if $v.key > e.key$ then
    - if $v.left = null$ then
      - Create a new node u for e and set $v.left \leftarrow u$
    - else
      - TreeInsert(e, $v.left$)
  - else
    - if $v.right = null$ then
      - Create a new node u for e and set $v.right \leftarrow u$
    - else
      - TreeInsert(e, $v.right$)

- new node u
- $u.key \leftarrow e.key$
- $u.value \leftarrow e.value$
- $u.left \leftarrow null$
- $u.right \leftarrow null$

Insertions

Function to insert $e = (key, value)$ to the subtree rooted at v

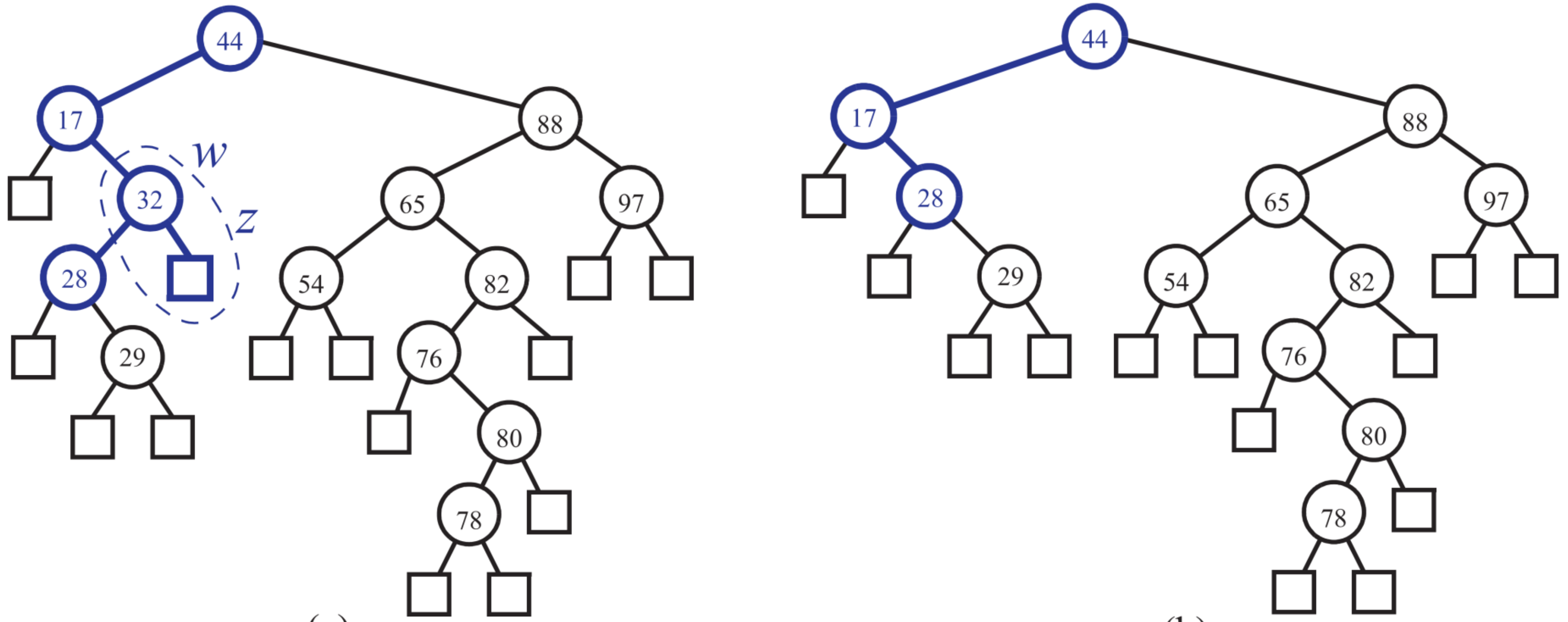
- TreeInsert(e, v)
  - if $v.key > e.key$ then
    - if $v.left = null$ then
      - Create a new node u for e and set $v.left \leftarrow u$
    - else
      - TreeInsert(e, $v.left$)
  - else
    - if $v.right = null$ then
      - Create a new node u for e and set $v.right \leftarrow u$
    - else
      - TreeInsert(e, $v.right$)

Complexity: $O(h)$

- Every recursion increases the level by one

Deletions

Removal of a node w: if w has ≤ 1 child



(a) (b)

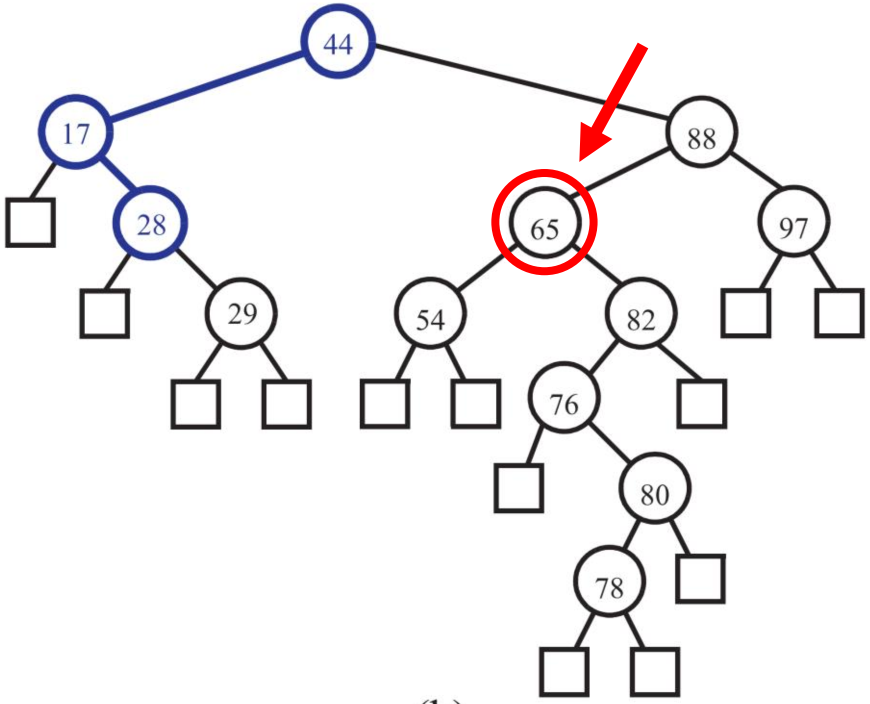
Deletions

Removal of a node w: if w has two children ?

Replace w with:

- maximum element on the left subtree of w, or
- minimum element on the right subtree of w

in this course



(b)

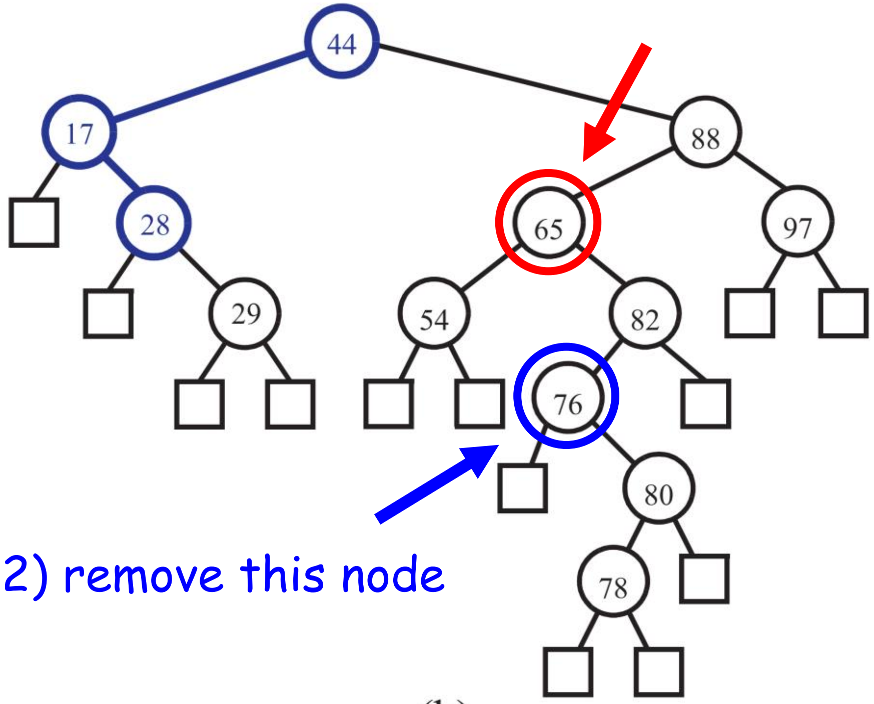
Deletions

Removal of a node w: if w has two children ?

Replace w with:

- maximum element on the left subtree of w, or
- left-most element on the right subtree of w

(1) replace this with node 76

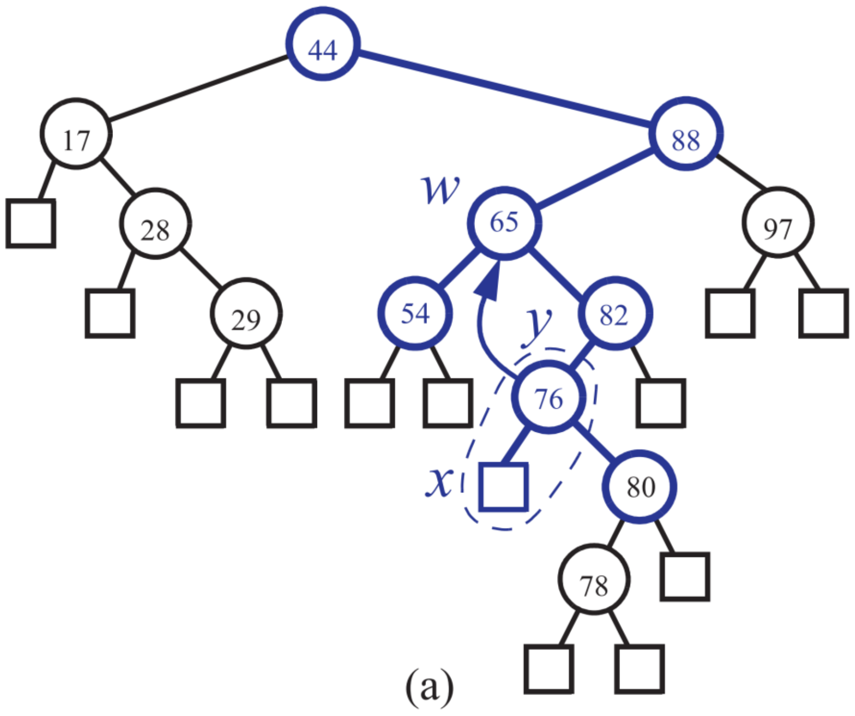


(2) remove this node

(b)

Deletions

Removal of a node w: if w has two children



Deletions

Removal of a node w:

- if w has no child:
  - Update corresponding pointer of parent of w to null
- if w has one child:
  - Replace w by its only child
- if w has two children
  - Replace w by the left-most node u on the right subtree
  - Remove node u (which has at most one child)

Complexity: $O(h)$

Binary Search Tree

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU
澳門大學

- Binary Search Tree (BST)
  - insert(e) $\Rightarrow O(h)$ time
  - find(key) $\Rightarrow O(h)$ time
  - remove(key) $\Rightarrow O(h)$ time
  - remove(p) $\Rightarrow O(h)$ time

- **Observation**: the tree is more efficient when h is small.
  - In worst case, h can be as large as n (example?)
- Question: how to maintain a small height?

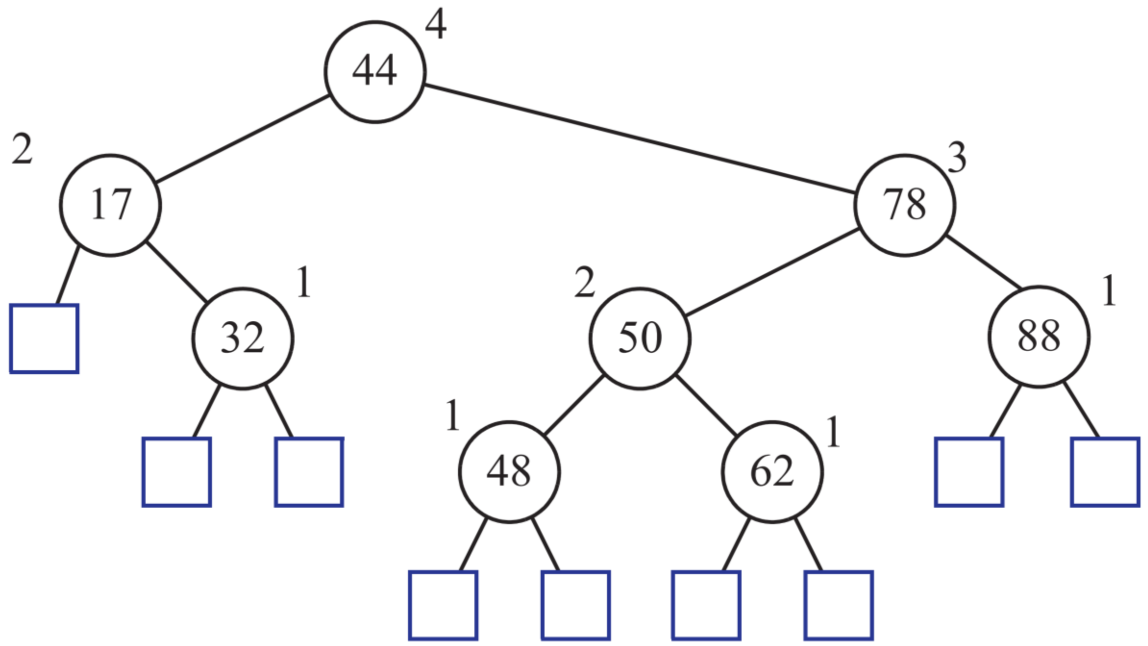
AVL Tree

AVL Tree

- A balanced binary tree
- Height-Balance Property:

  For every internal node v the heights of subtrees rooted at children of v differ by at most 1.

- Named by its inventors: Adel'son-Vel'skii and Landis.



AVL Tree

- **Lemma**. A binary tree with n internal nodes satisfying the height balance property has height $h = O(\log n)$.
- Proof 1.
  - Let $f(h)$ be the minimum number of nodes stored at a height-balanced binary tree of height h, e.g., $f(1) = 2$; $f(2) = 4$.
  - We have $f(h) = 1 + f(h-1) + f(h-2) > 2 \cdot f(h-2) > 2^i \cdot f(h - 2 \cdot i)$.
  - Let $i = \lceil h/2 \rceil - 1$, we have $f(h) > 2^i \cdot f(1) = 2^{i+1} \geq 2^{h/2}$.
  - Every height-balanced binary tree of height h has at least $2^{h/2}$ nodes.
  - Therefore, $n \geq 2^{h/2}$, which implies $h \leq 2 \cdot \log n = O(\log n)$.

AVL Tree

- **Lemma**. A binary tree with n internal nodes satisfying the height balance property has height $h = O(\log n)$.
- Proof 2.
  - Prove that $h \le 2\log n$ for a tree with n internal nodes
    - by Mathematical Induction on n
  - Base case: $n = 1$ and $h = 0$
  - Assume statement true for all $i < n$, and consider $n = 1 + n_l + n_r$
    - n_l (resp. n_r): internal nodes on the left (resp. right) subtree
  - By induction hypothesis: $h_l \le 2\log n_l$ and $h_r \le 2\log n_r$
    - h_l (resp. h_r): height of the left (resp. right) subtree
  - We have

$$h = 1 + \max\{h_l, h_r\} \le 2 + \min\{h_l, h_r\} \le 2 + \left(2\log\frac{n}{2} + 2\right) = 2\log n + 2$$

AVL Tree

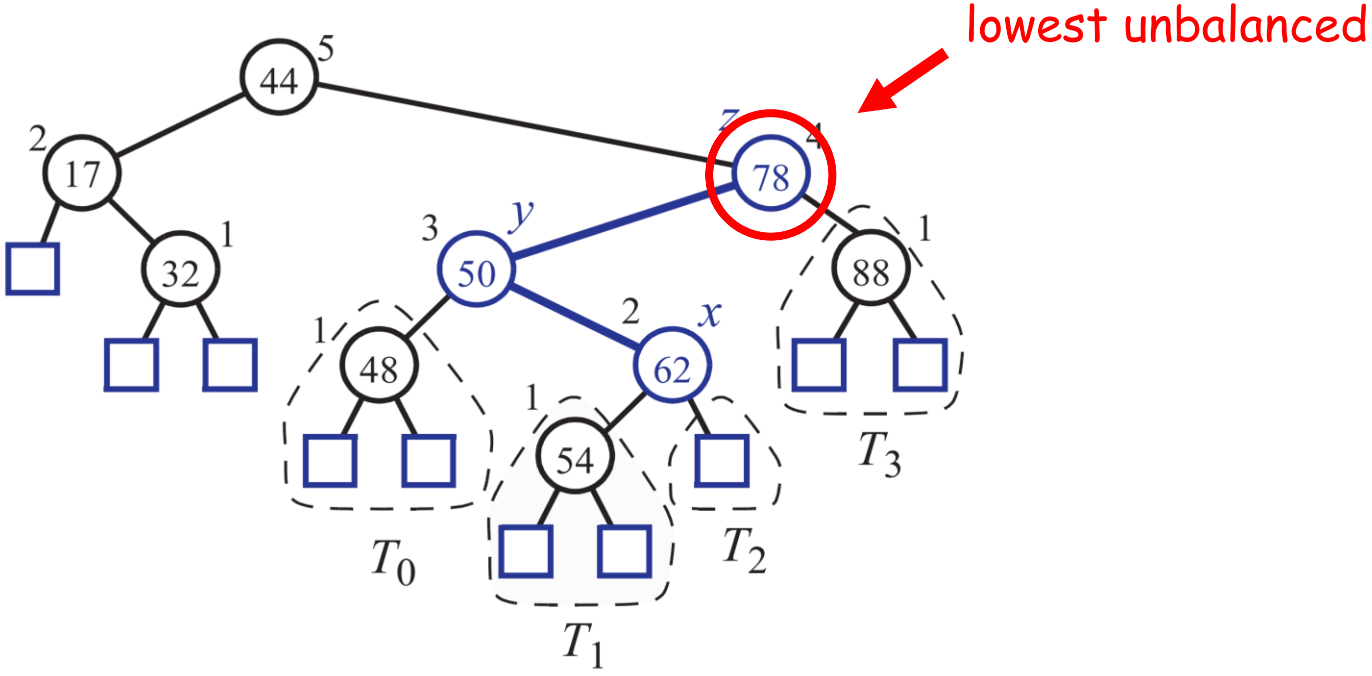
澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Lemma**. A binary tree with n internal nodes satisfying the height balance property has height $h = O(\log n)$.

- How to maintain the property ?
  - Height might get increased by Insert(e)
  - Height might get decreased by Remove(p)
  - Node v is balanced/unbalanced if the difference between subtrees rooted at its children differ by at-most/larger-than 1
  - Need to fix the unbalanced nodes after an update

Insertions

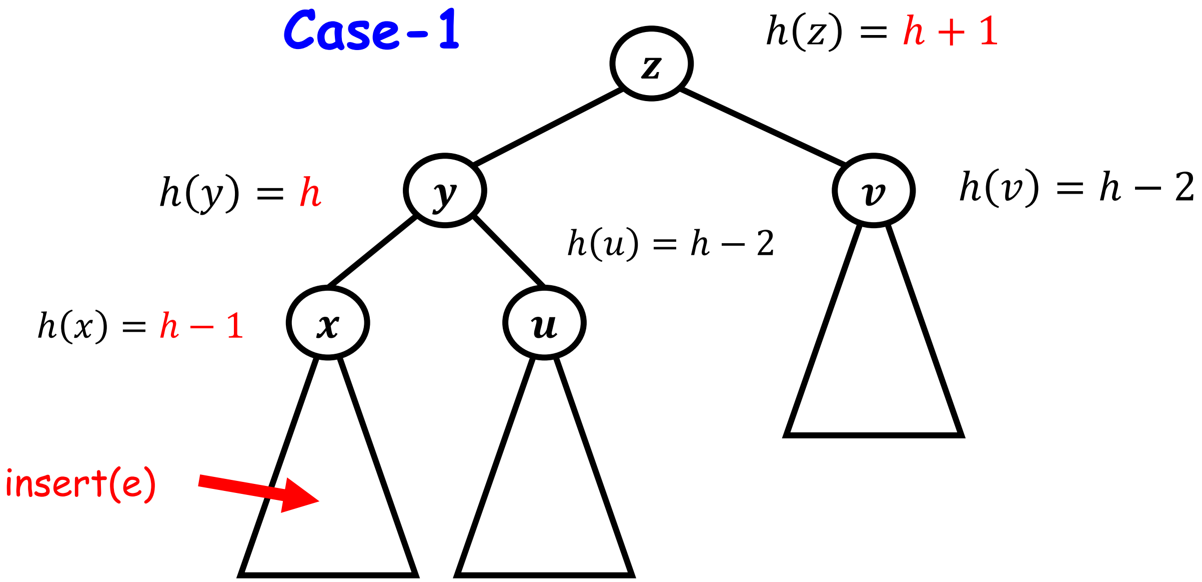
- Insertion of element with key 54



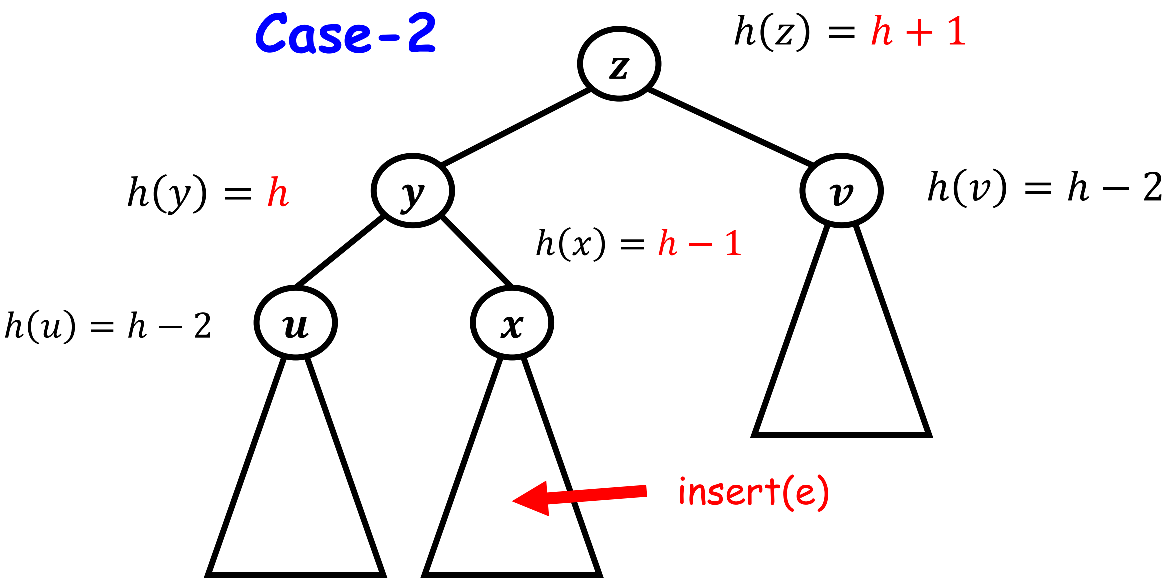
Insertions

- How to re-balance the tree?
- Find sub-trees of height $\leq h - 2$

Case-1

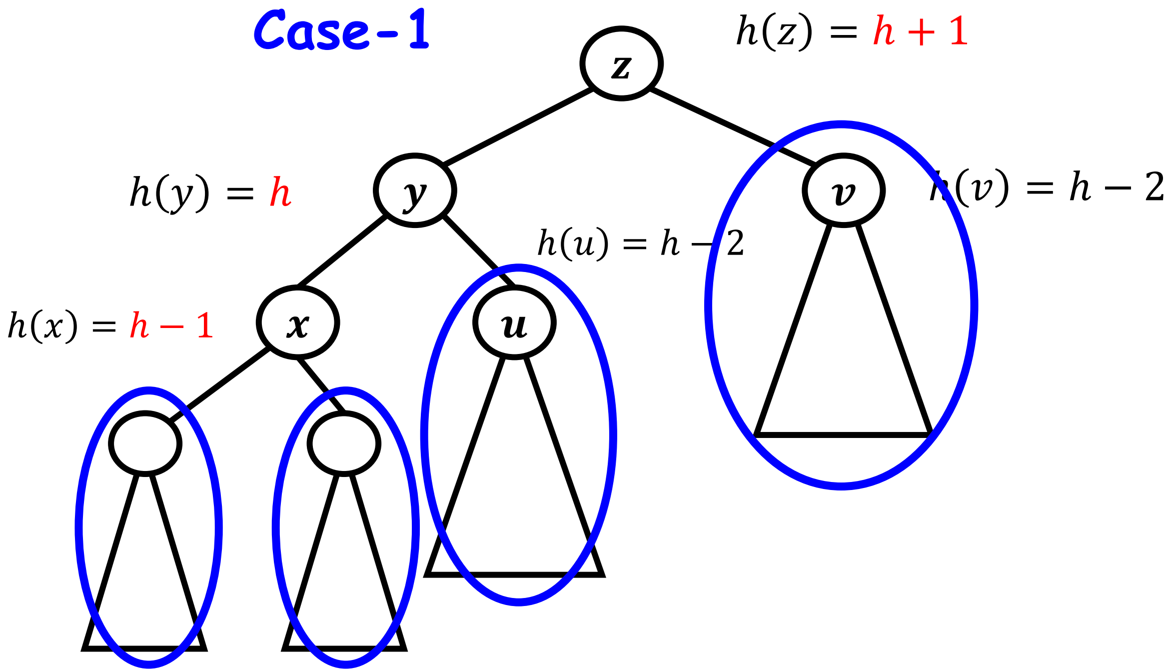


Case-2



Insertions

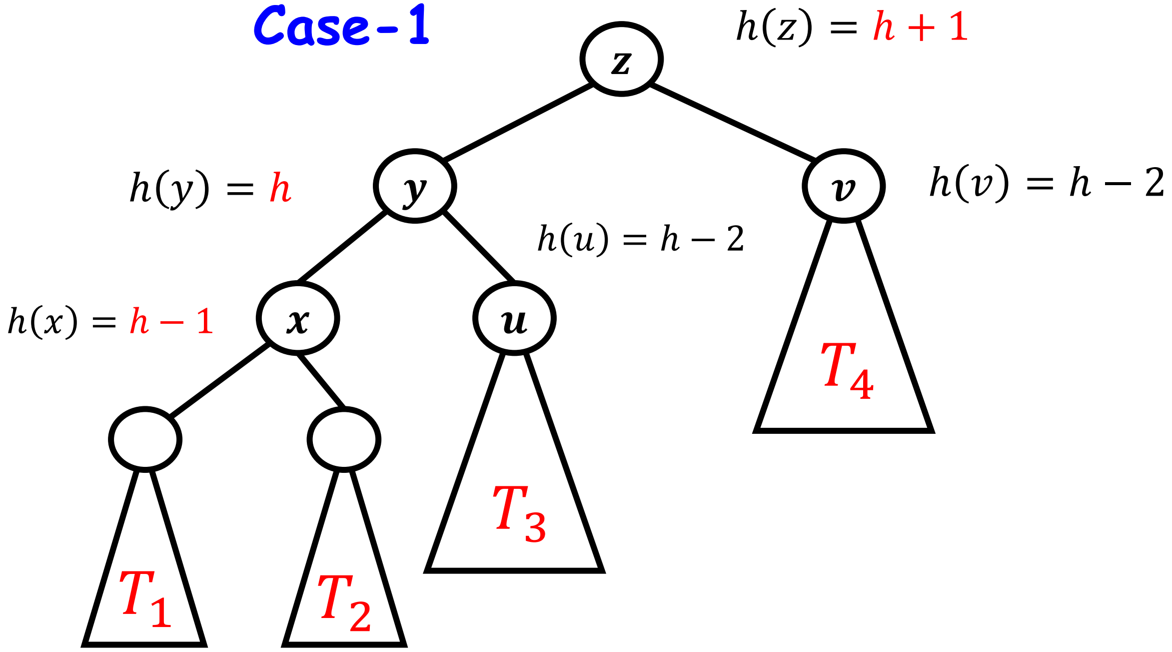
- How to re-balance the tree?
- Find sub-trees of height $\leq h-2$

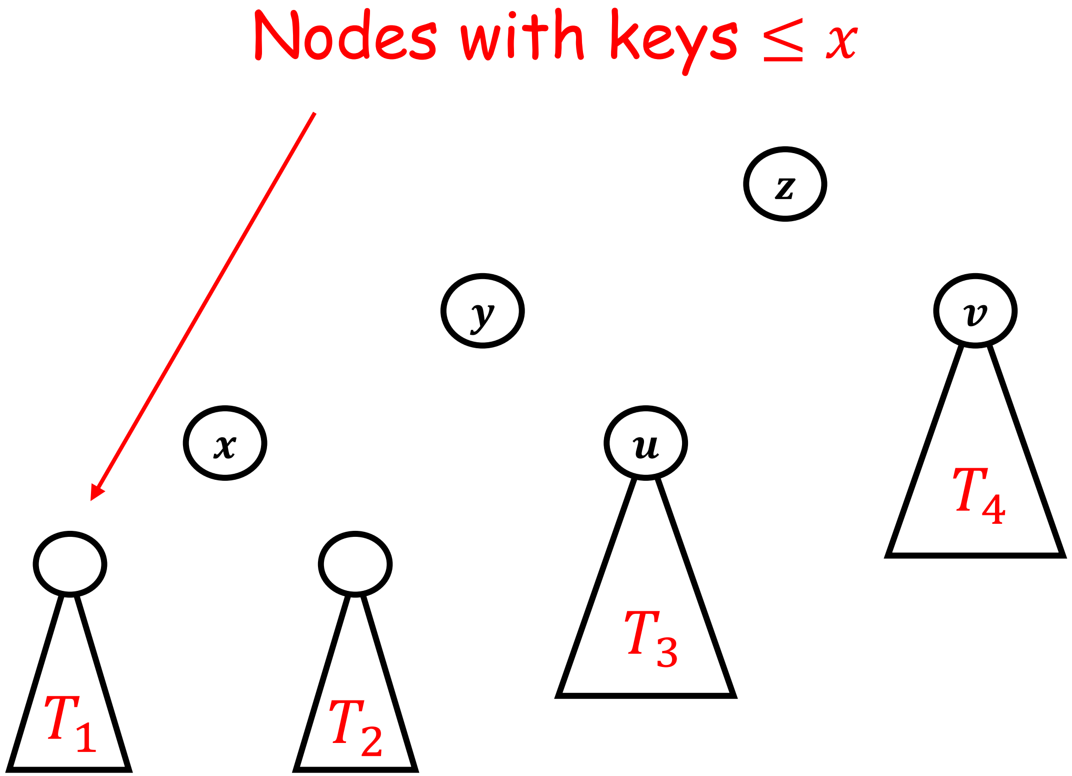


Insertions

- How to re-balance the tree?
- Find sub-trees of height $\leq h - 2$

Case-1

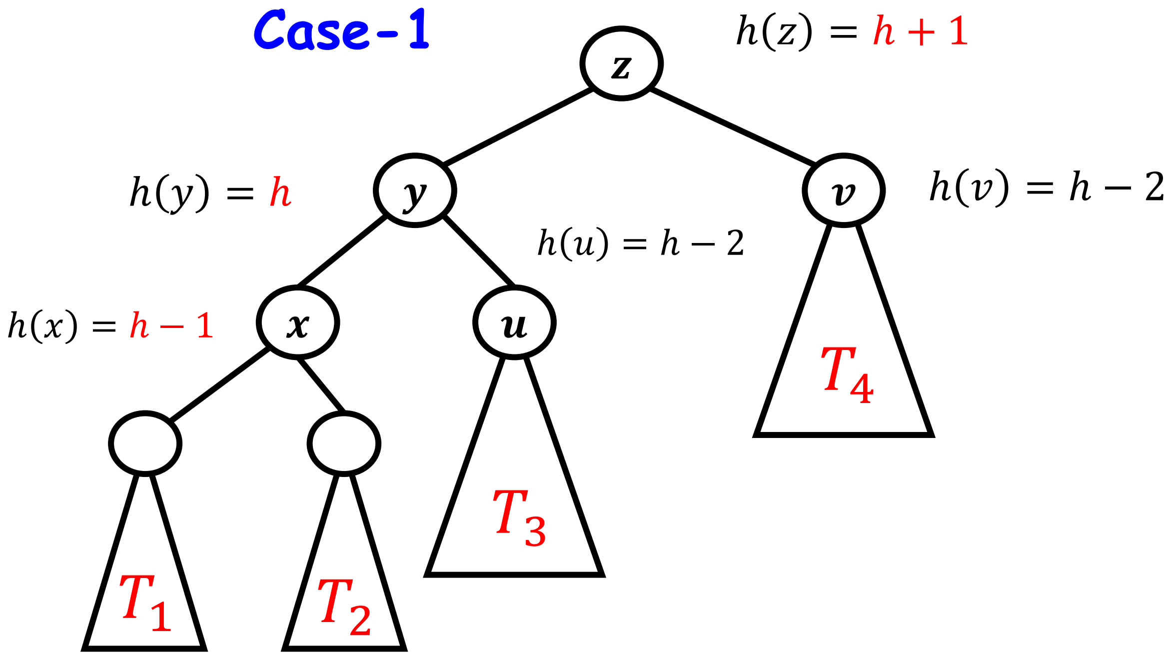


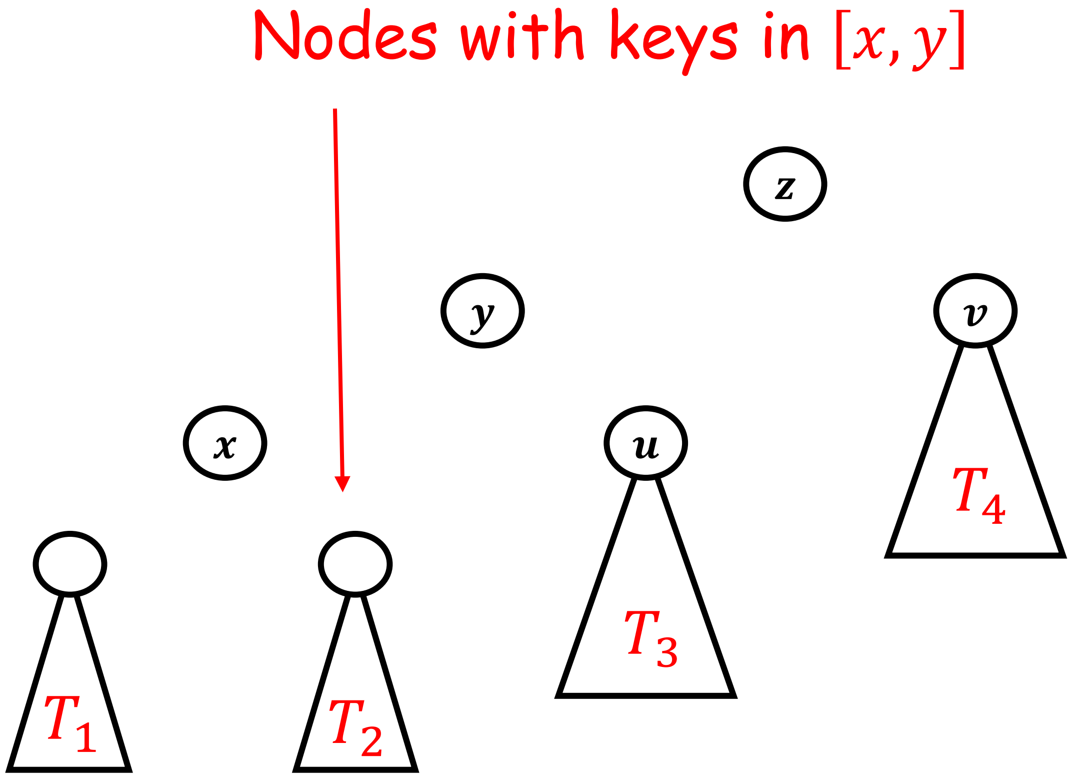


Insertions

- How to re-balance the tree?
- Find sub-trees of height $\leq h - 2$

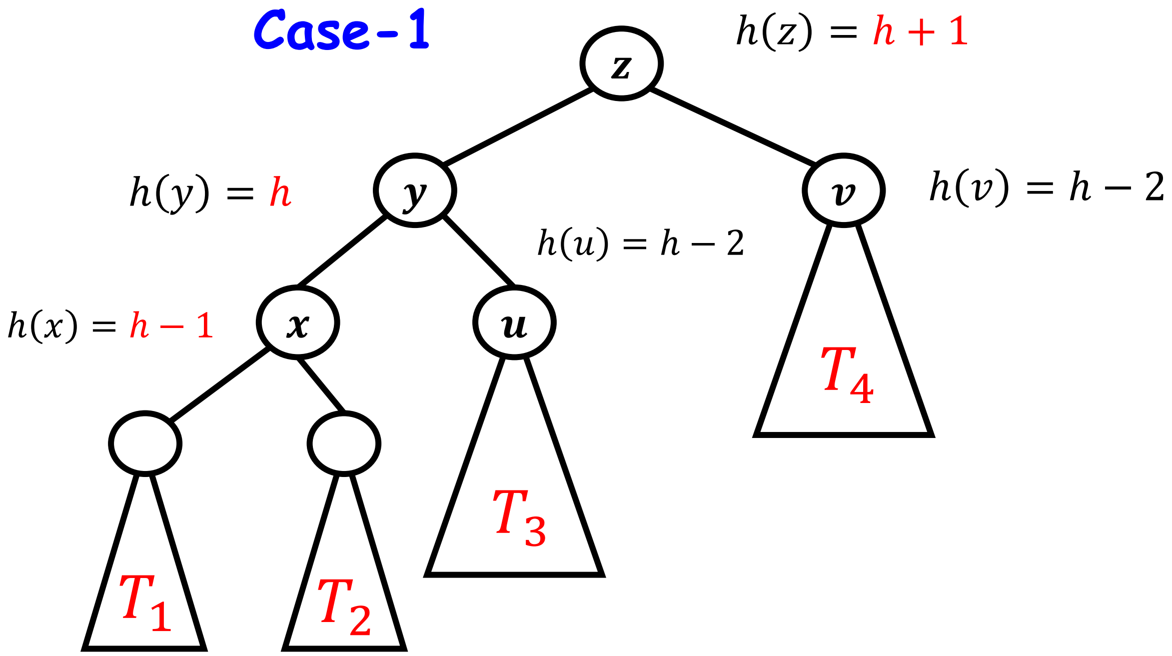
Case-1

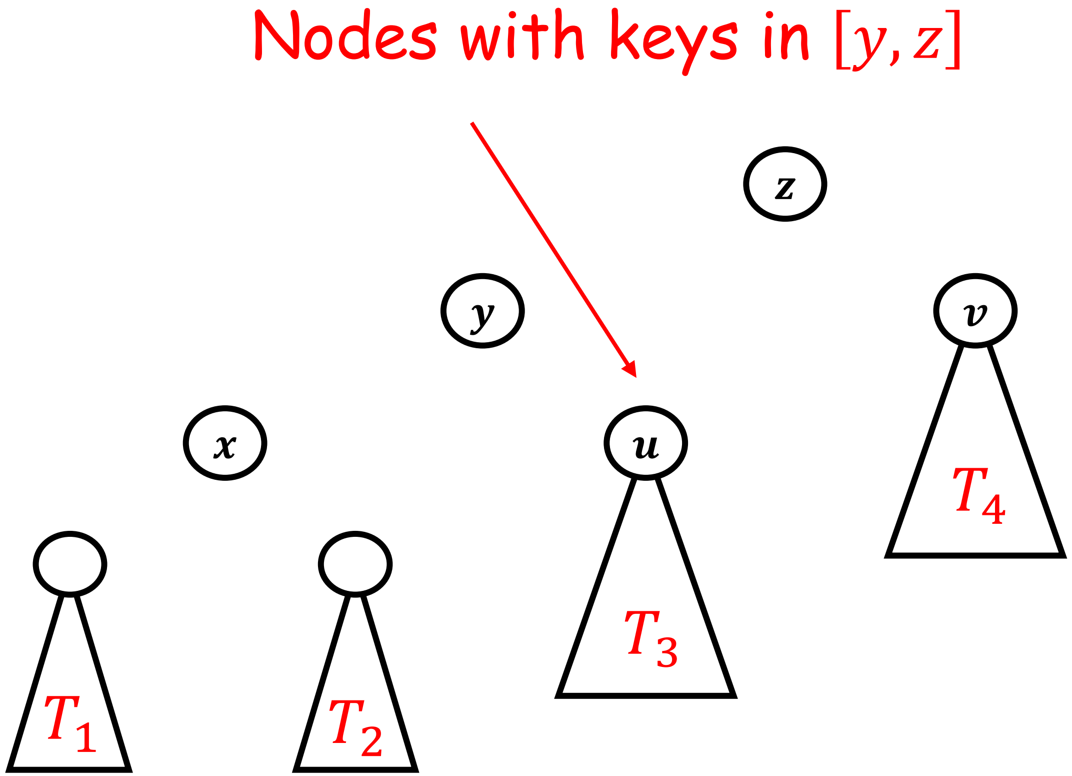




Insertions

- How to re-balance the tree?
- Find sub-trees of height $\leq h - 2$

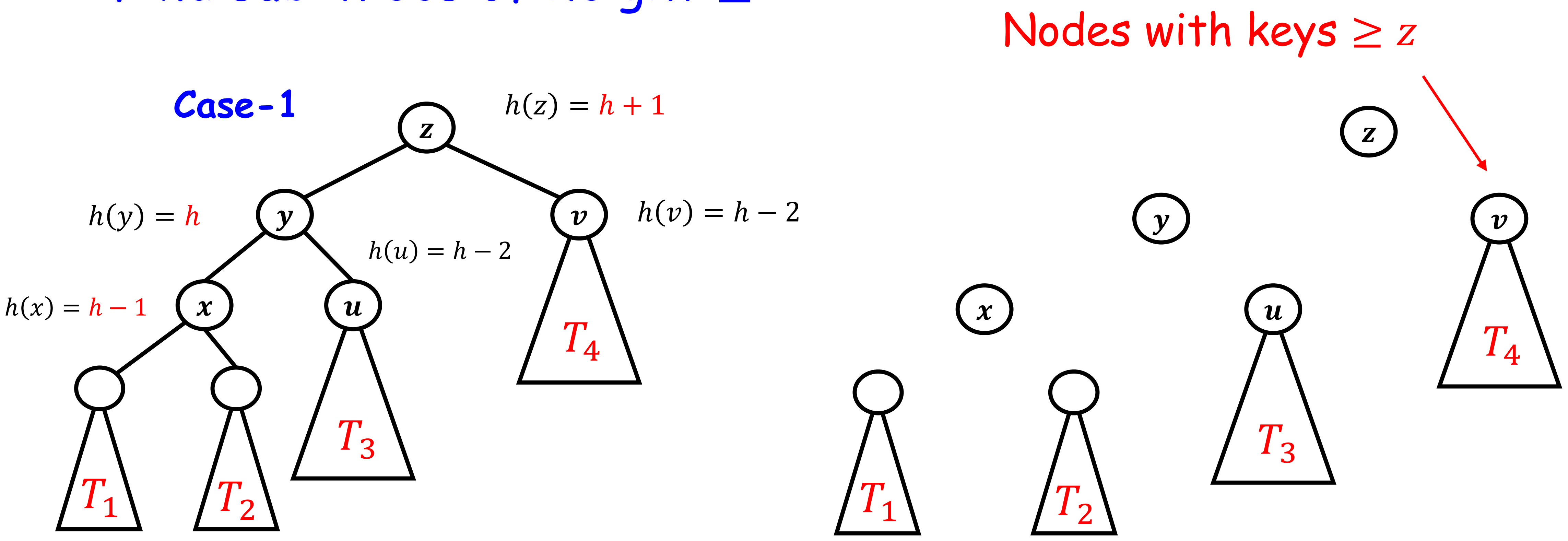




Insertions

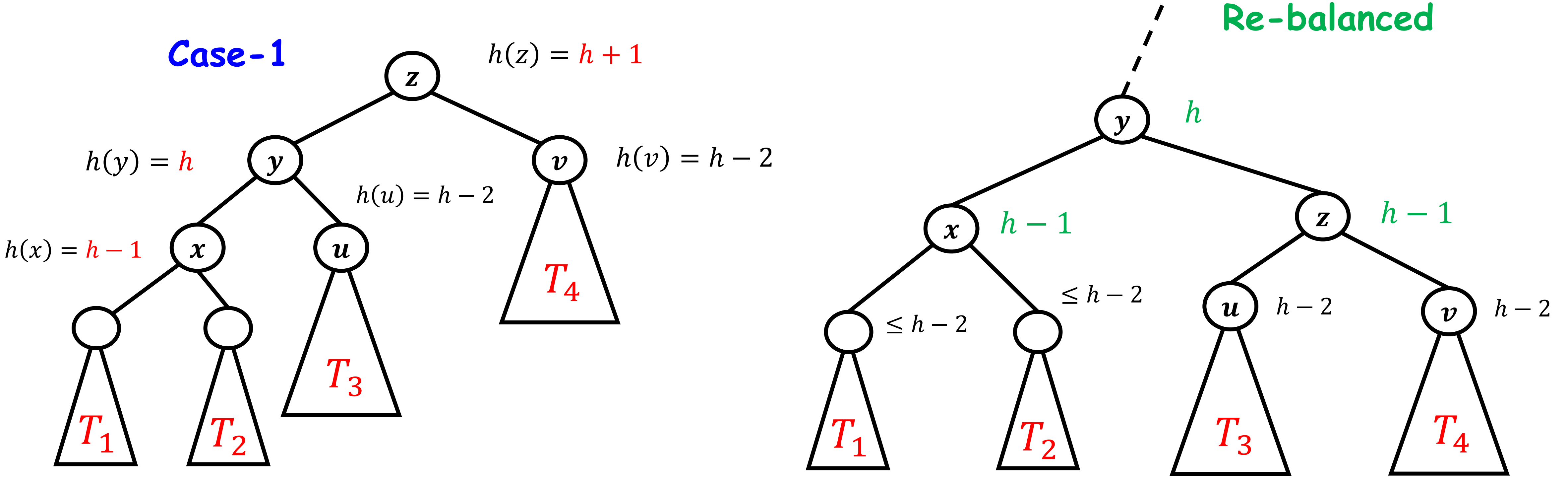
澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- How to re-balance the tree?
- Find sub-trees of height $\leq h - 2$



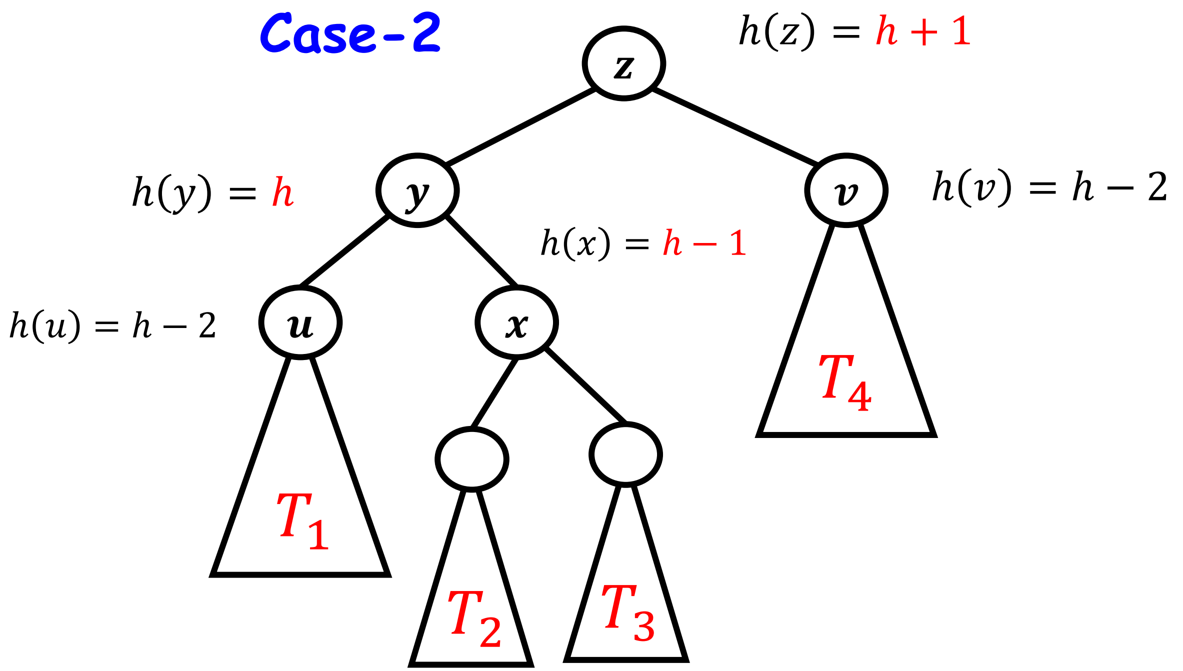
Insertions

- How to re-balance the tree?
- Find sub-trees of height $\leq h-2$



Insertions

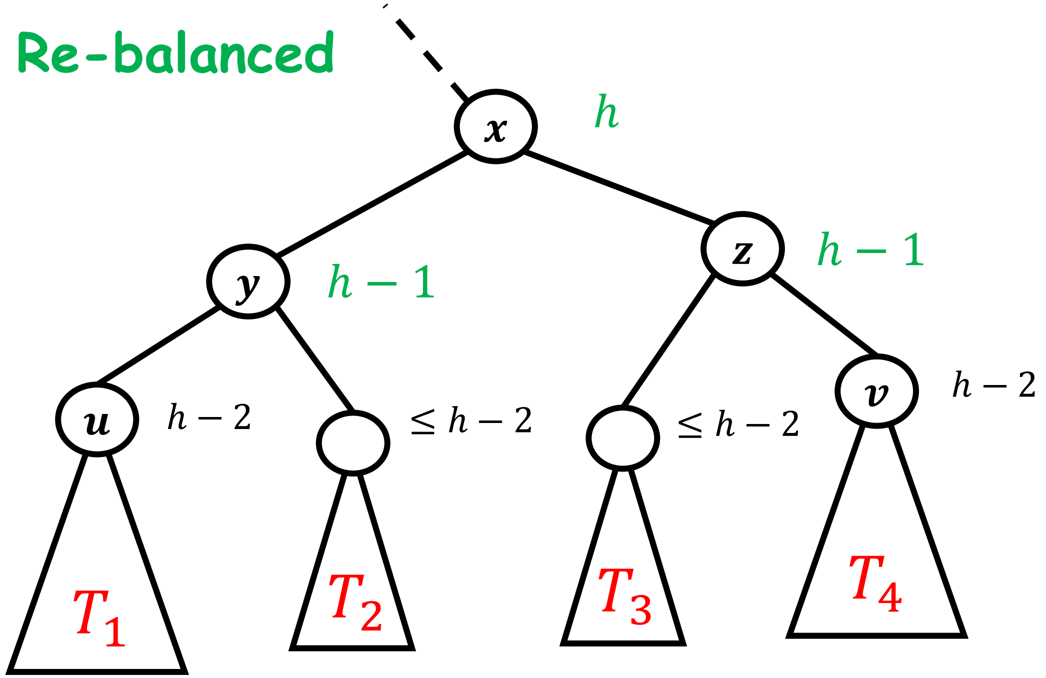
- How to re-balance the tree?
- Find sub-trees of height $\leq h - 2$

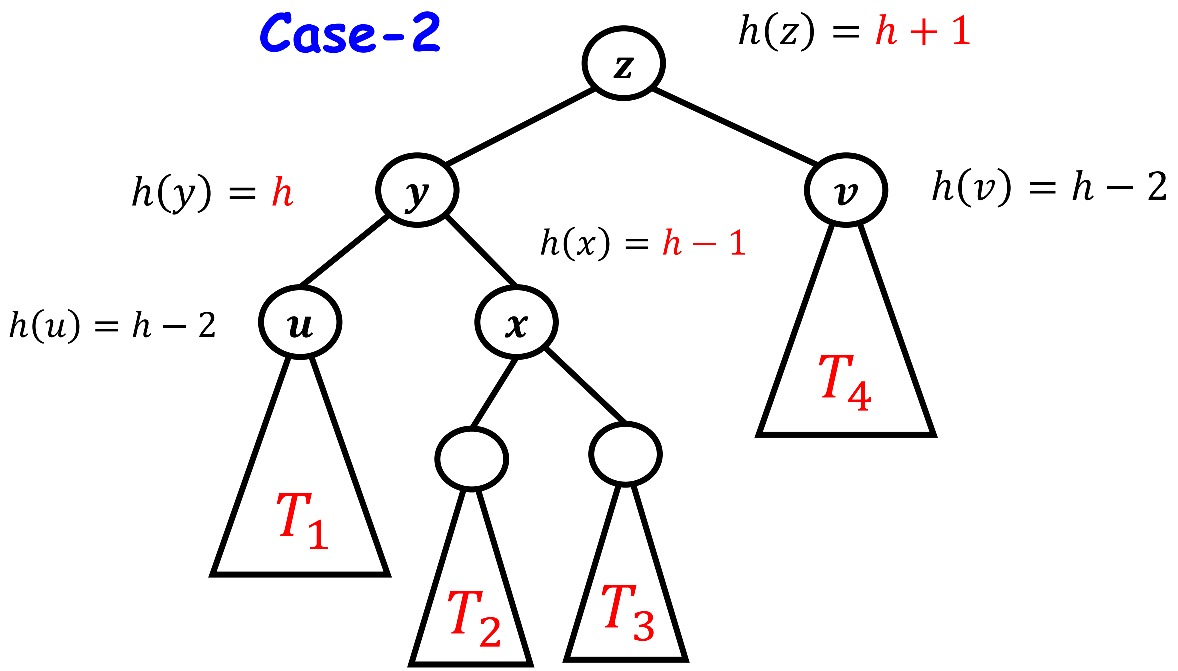


Insertions

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

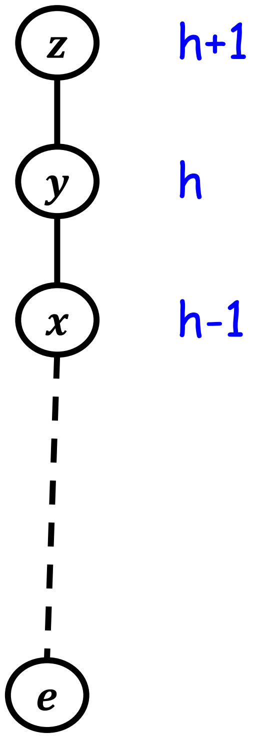
- How to re-balance the tree?
- Find sub-trees of height $\leq h-2$

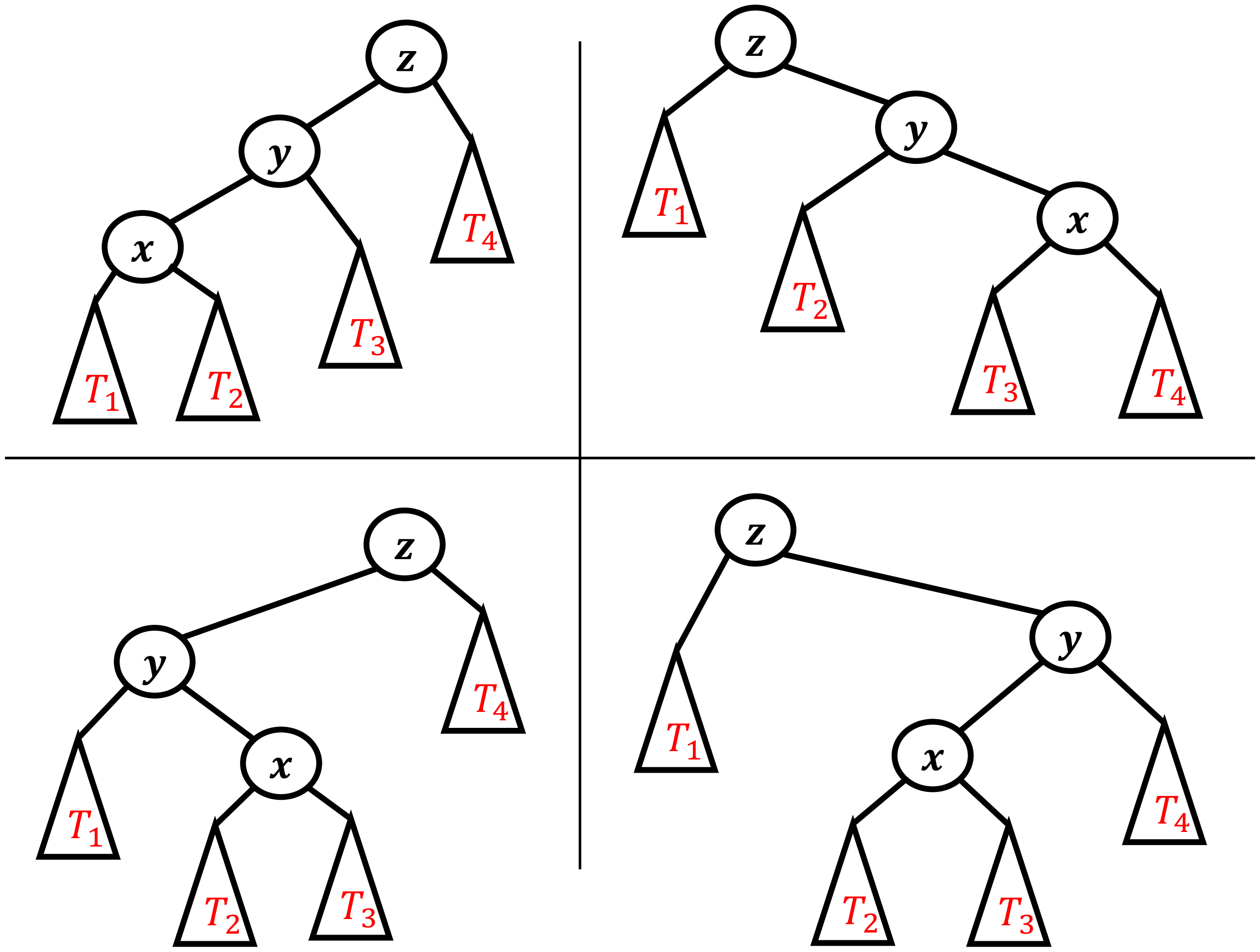




Insertions

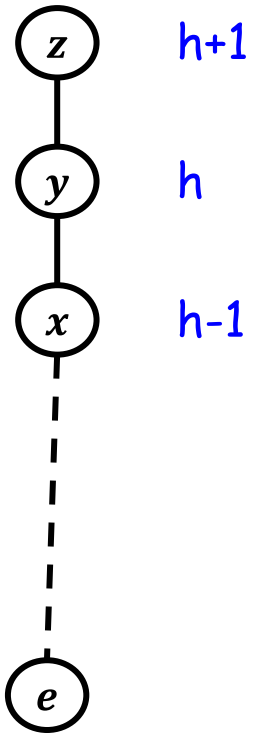
- After insert(e)

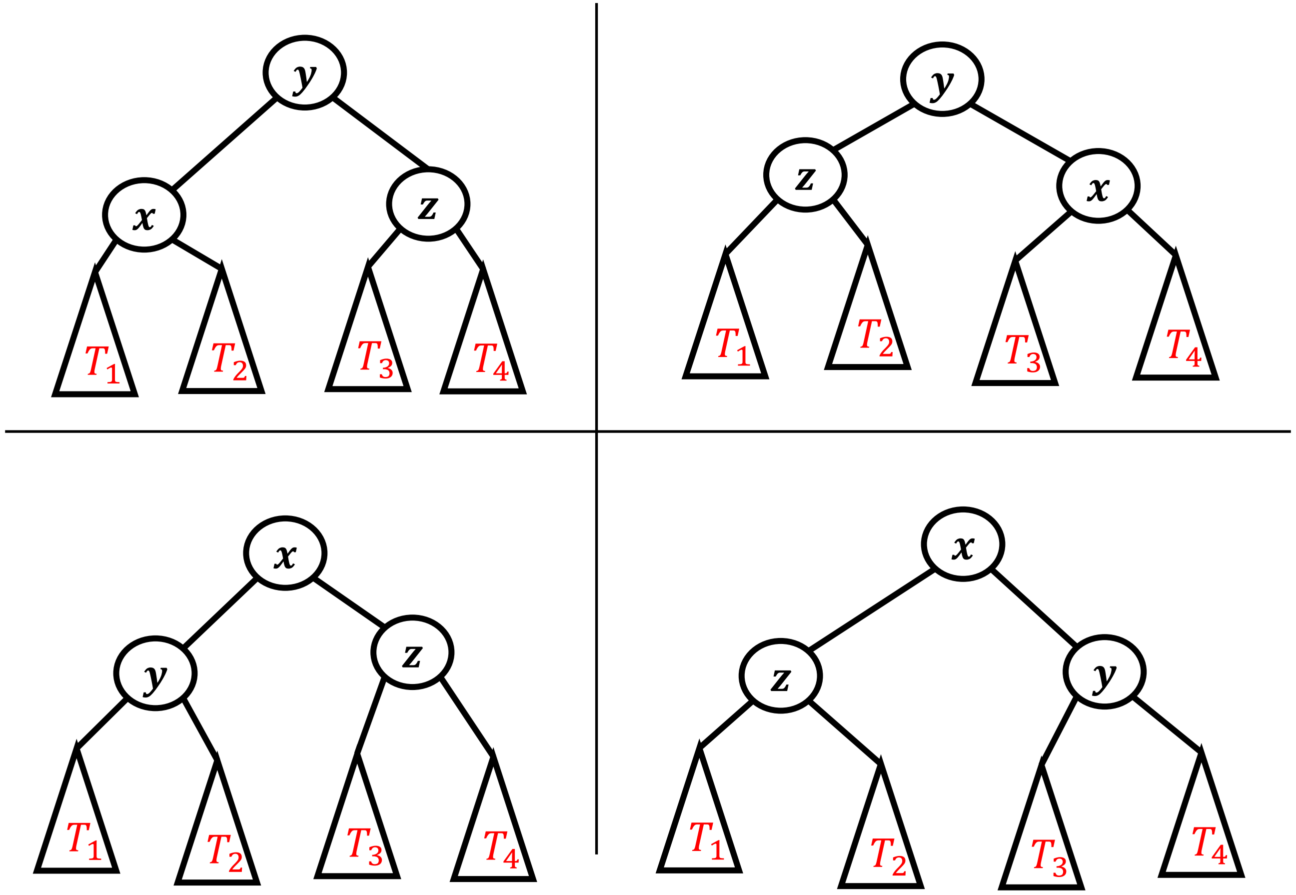




Insertions

- After insert(e)



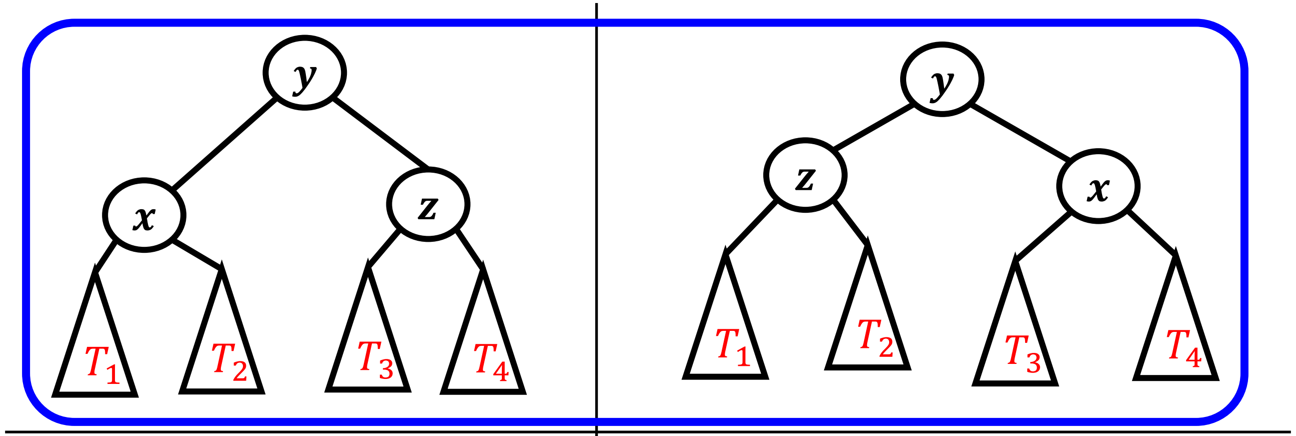


Insertions

- After insert(e)

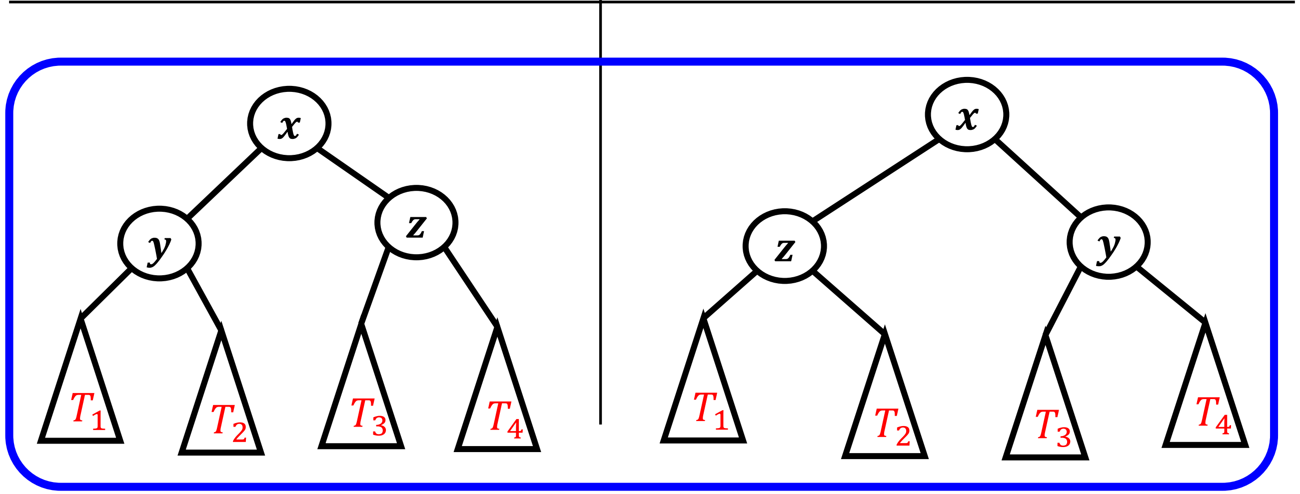
Single Rotation

- Rooted at y
- Subtree rooted at x does not change



Double Rotation

- Rooted at x
- Subtree rooted at x changed



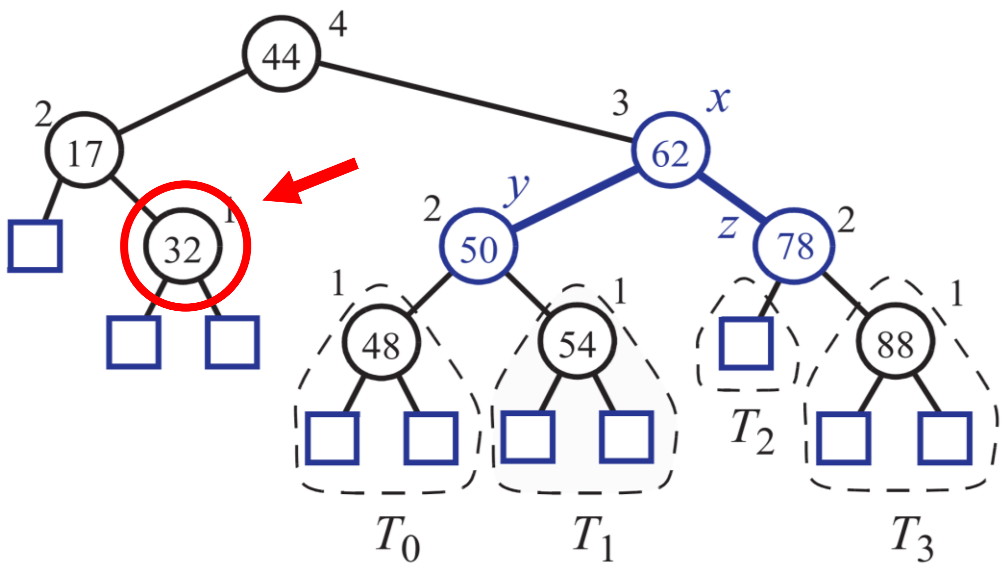
Insertions

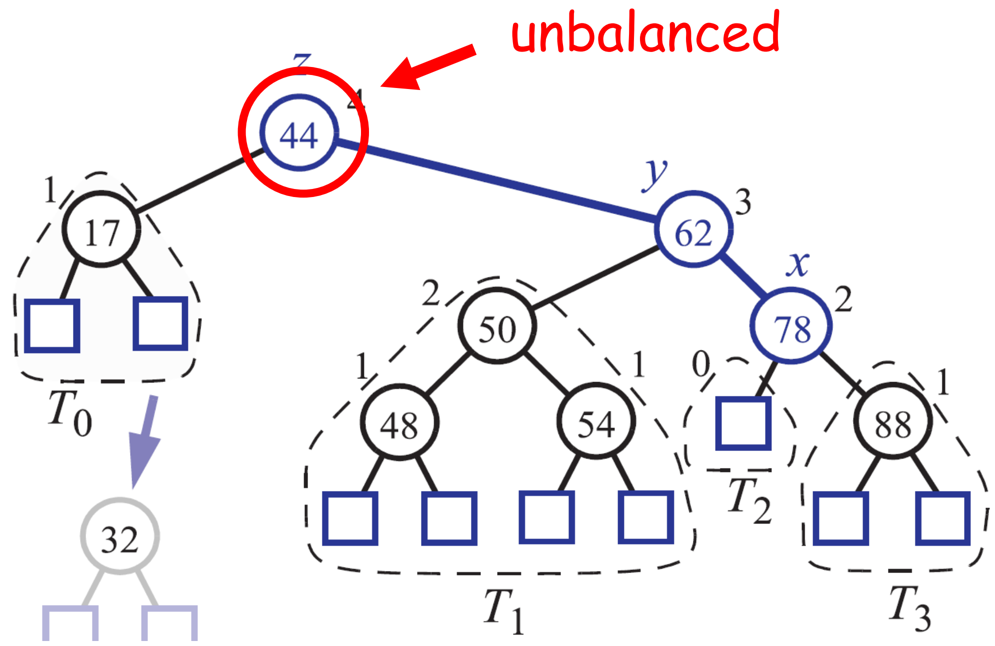
- After insert(e)
- Identify the lowest unbalanced node z
- Let x, y, z be the last three nodes on the path from e to z
- Perform single/double rotation to re-balance the subtree rooted at z.
    - Height of $T(z)$ changes from $h+1$ to h
- Observation. The whole tree is balanced again !
    - Height of $T(z)$: $h \rightarrow (h+1) \rightarrow h$
    - Complexity of re-balancing after insertion: $O(1)$

Deletions

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

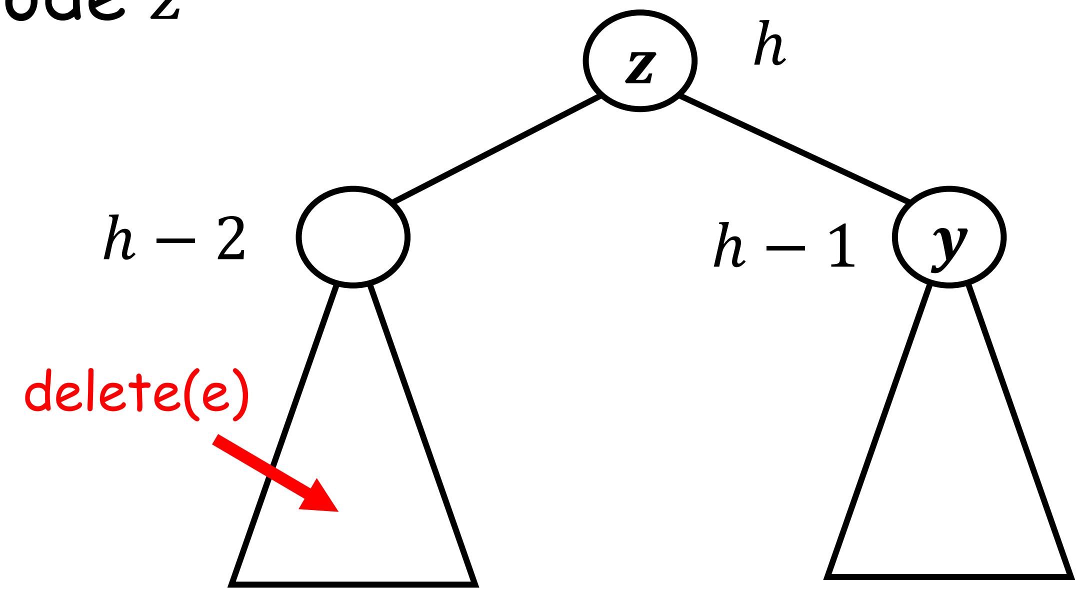
- Removal of element with key 32





Deletions

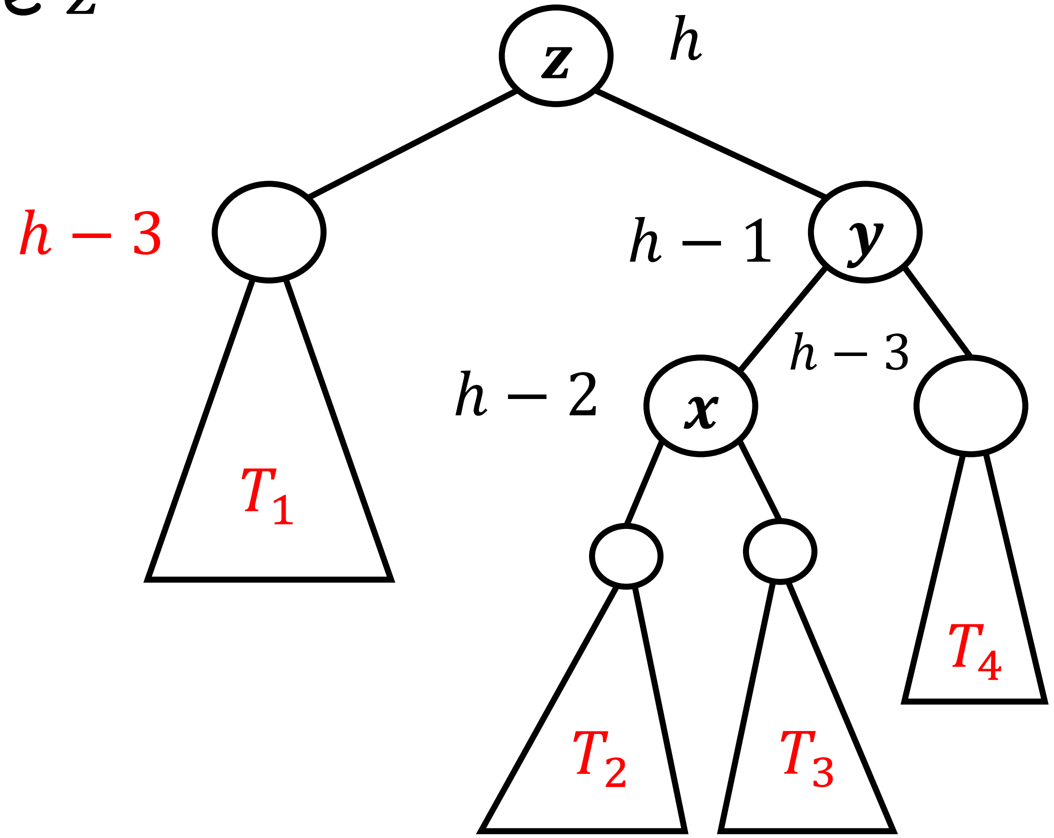
- Removal of element e
- Identify the lowest unbalanced node z
- Let y be the other child of z



Before deletion

Deletions

- Removal of element e
- Identify the lowest unbalanced node z
- Let y be the other child of z
- Let x be child of y such that
  - If subtrees of y have different heights: let x be the taller one

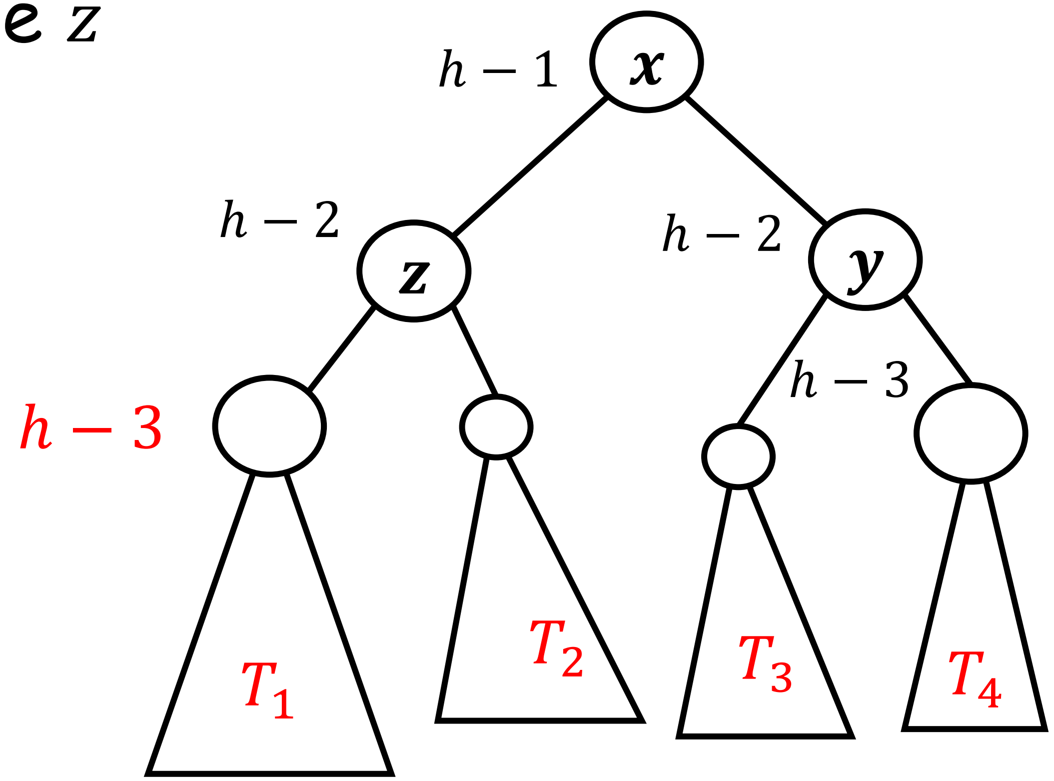


Deletions

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

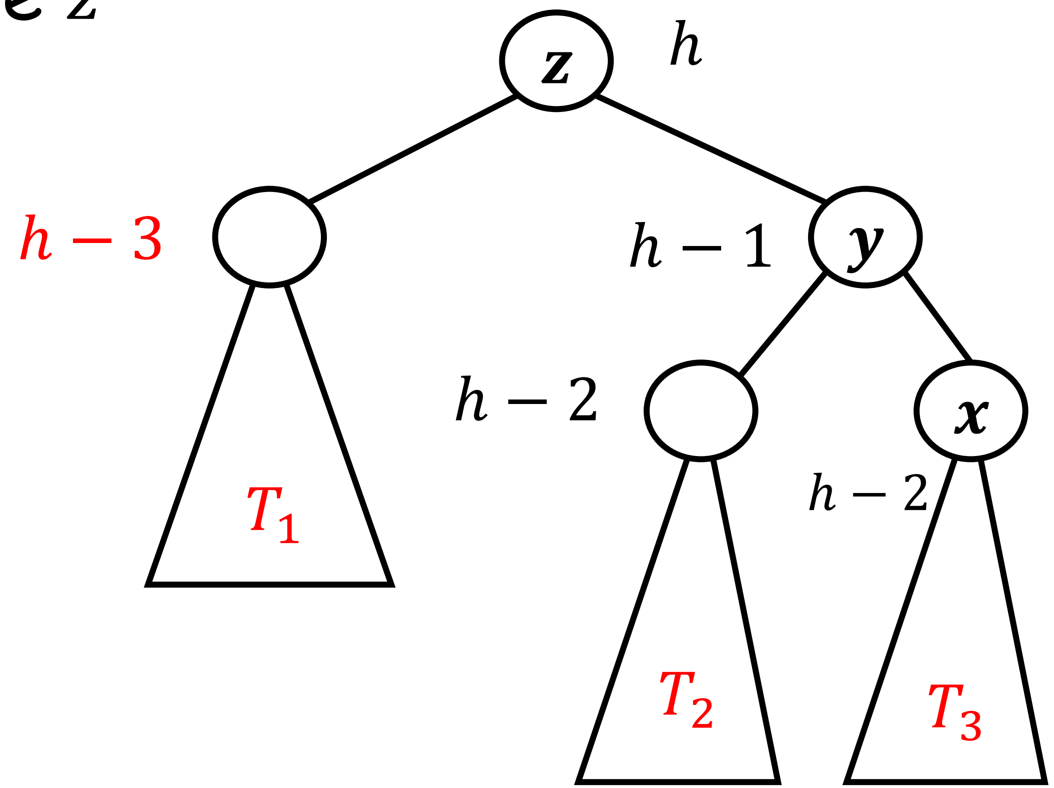
- Removal of element e
- Identify the lowest unbalanced node z
- Let y be the other child of z
- Let x be child of y such that
  - If subtrees of y have different heights: let x be the taller one

Re-balanced



Deletions

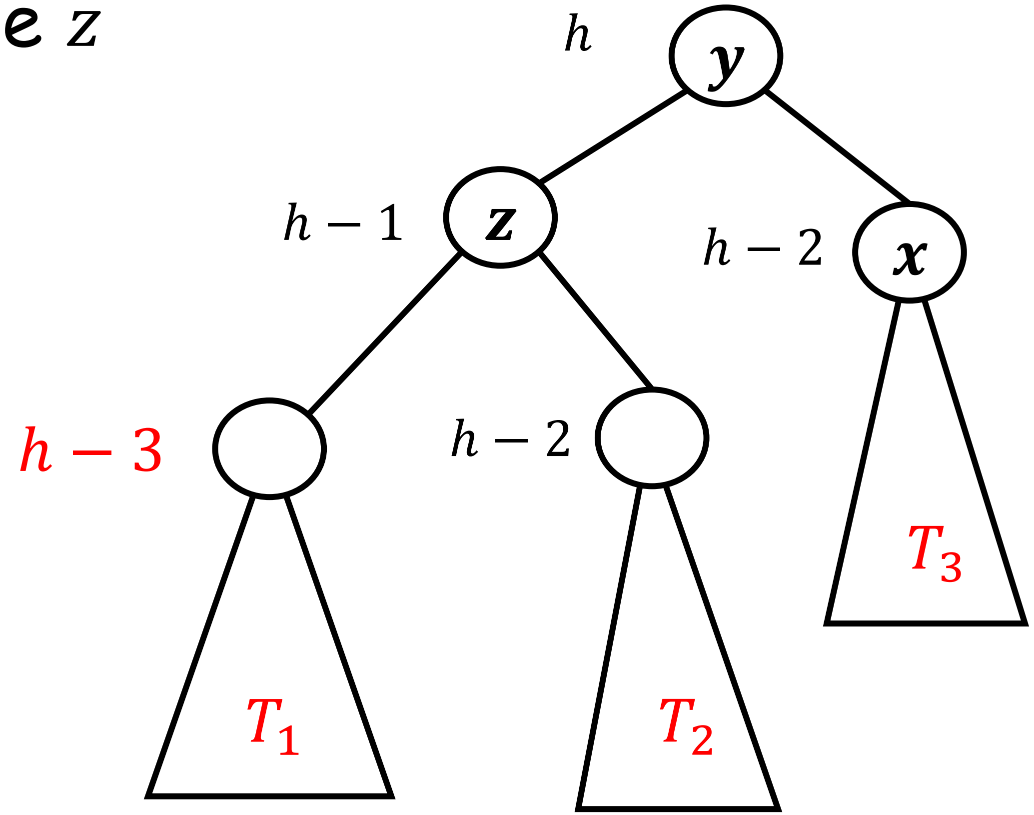
- Removal of element e
- Identify the lowest unbalanced node z
- Let y be the other child of z
- Let x be child of y such that
  - If subtrees of y have different heights: let x be the taller one
  - If subtrees of y have the same height: let x be on the same side as y being child of z.



Deletions

- Removal of element e
- Identify the lowest unbalanced node z
- Let y be the other child of z
- Let x be child of y such that
    - If subtrees of y have different heights: let x be the taller one
    - If subtrees of y have the same height: let x be on the same side as y being child of z.

Re-balanced



Deletions

- Removal of element e
- Identify the lowest unbalanced node z
- Let y be the other child of z
- Let x be child of y such that
  - If subtrees of y have different heights: let x be the taller one
  - If subtrees of y have same height: let x be on the same side as y
- Re-balance $T(z)$ be single/double rotation

- ~~Observation: The whole tree is balanced again ?~~

Deletions

- Removal of element e
- While(there exists an unbalanced node)
  - Identify the lowest unbalanced node z
  - Re-balance $T(z)$ by single/double rotation

- Observation: at most h re-balancings are necessary
  - As re-balancing happens at higher and higher places
- Complexity: $O(h)$
  - Each re-balancing takes $O(1)$ time

Binary Search Tree

- Binary Search Tree (BST) supports
  - insert(e) insert an element, e.g., e = (key, value)
  - find(key) find element e with e.key = key
  - remove(key) remove the element e with e.key = key
  - remove(p) remove the element e point p points to

- Complexity of AVL tree:

Operation	**Time***Time*
size, empty	$O(1)$
find, insert, erase	$O(\log n)$

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Priority Queue vs. BST

- BST is more "powerful"
  - Supports searching of any key
  - Can be easily extended to support min() and removeMin()
- Priority Queue is more efficient in finding the minimum
  - $O(1)$ in priority queue; $O(\log n)$ for BST
- Tree structures with different properties
  - Heap (for priority queue) : heap order property
  - AVL tree (for BST) : left $\leq$ middle $\leq$ right

Sorting Algorithms

Sorting

- **Input**: a sequence of numbers $A = (a_1, a_2, \dots, a_n)$
- **Output**: a sorted sequence of numbers in A, which is denoted by $B = (b_1, b_2, \dots, b_n)$, where $b_1 \leq b_2 \leq \cdots \leq b_n$.

General version:

- Numbers -> Elements
- Need to define the comparator
  - elements $x \preccurlyeq y$ iff compare(x, y) = True

Sorting

- **Simple Sorting Algorithms:**
  - Insertion Sort
  - Bubble Sort
  - Selection Sort

- **Advanced Sorting Algorithms:**
  - Heap-Sort
  - Merge-Sort
  - Quick-Sort

Insertion Sort

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Main Idea**:
  - For each $i = 1,2,\dots,n$, sort prefix $(a_1, a_2, \dots, a_i)$
  - For each a_i, find the right position in $\{1,2,\dots,i\}$ by moving forward.

- **for** $(i = 2, \dots, n)$: // try to find a position for a_i
  - $e \leftarrow a_i, j \leftarrow i - 1$
  - **while** $(j \geq 1$ and $a_j > e)$ // a_j should appear after e
    - $a_{j+1} \leftarrow a_j, j \leftarrow j - 1$
  - $a_{j+1} \leftarrow e$

Insertion Sort

- **Main Idea:**
  - For each $i = 1,2,\dots,n$, sort prefix $(a_1, a_2, \dots, a_i)$
  - For each a_i, find the right position in $\{1,2,\dots,i\}$ by moving forward.

- **Example:** input = (5,7,2,6,9,3)
  - $i = 2$: (5,7,2,6,9,3) -> (5,7,2,6,9,3)
  - $i = 3$: (5,7,2,6,9,3) -> (2,5,7,6,9,3)
  - $i = 4$: (2,5,7,6,9,3) -> (2,5,6,7,9,3)
  - $i = 5$: (2,5,6,7,9,3) -> (2,5,6,7,9,3)
  - $i = 6$: (2,5,6,7,9,3) -> (2,3,5,6,7,9)

Insertion Sort

- **Main Idea:**
  - For each $i = 1,2,\dots,n$, sort prefix $(a_1, a_2, \dots, a_i)$
  - For each a_i, find the right position in $\{1,2,\dots,i\}$ by moving forward.

- **Complexity:** $O(n^2)$
  - There are $n-1$ for-loops
  - Each for-loop executes in $O(i) = O(n)$ time
    - It takes $O(i)$ time to locate the position to "insert" a_i

Bubble Sort

- **Main Idea:**
  - Do several linear scanning over the sequence
  - For each element, move it backward if it is larger than its successor

- **for** $(i = 1,2,\ldots,n-1)$: // round-i
  - **for** $(j = 1,2,\ldots,n-i)$ // linear scanning
    - **if** $(a_j > a_{j+1})$
      - swap the values of a_j and a_{j+1}

Bubble Sort

- **Main Idea:**
  - Do several linear scanning over the sequence
  - For each element, move it backward if it is larger than its successor
- **Example:** input = (5,7,2,6,9,3)
  - Round 1: (5,7,2,6,9,3) -> (5,2,6,7,3,9)
  - Round 2: (5,2,6,7,3,9) -> (2,5,6,3,7,9)
  - Round 3: (2,5,6,3,7,9) -> (2,5,3,6,7,9)
  - Round 4: (2,5,3,6,7,9) -> (2,3,5,6,7,9)
  - Round 5: (2,3,5,6,7,9) -> (2,3,5,6,7,9)

Bubble Sort

- **Correctness:**
  - After round-i, the top-i maximum elements are well-sorted
  - At most $n-1$ rounds are necessary
  - In round-i, we only need to scan elements up to a_{n-i}

- **Complexity:** $O(n^2)$
  - At most $n-1$ rounds
  - Each round terminates in $n-i=O(n)$ time

Swap-Based Sorting

- Both Insertion-Sort and Bubble-Sort are swap-based
  - Every operation swaps positions of two adjacent elements
- Complexity of the algorithms: number of inversions
  - Pair (a_i, a_j) is an inversion if $i < j$ but $a_i > a_j$
  - Each swap decreases the number of inversions by one (why?)
  - No inversion in the sorted list
- Expected number of inversions in a random sequence: $\Theta(n^2)$

Selection Sort

- **Main Idea:**
  - For $i = 1,2,\ldots,n$:
    (1) select the smallest element e from A;
    (2) set $b_i \leftarrow e$;
    (3) remove e from A.

- **Algorithms:**
  - Sorting with priority queues implemented by unsorted list
  - Complexity: $O(n^2)$

Heap Sort

- **Main Idea:**
  - For $i = 1,2, \ldots, n$:
    (1) select the smallest element e from A;
    (2) set $b_i \leftarrow e$;
    (3) remove e from A.

- **Algorithms:**
  - Sorting with heap
  - Complexity: $O(n \log n)$

Merge-Sort

Merge-sort is based on divide-and-conquer.

- Divide: If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution obtained. Otherwise, divide the input data into two or more disjoint subsets.
- Recurse: Recursively solve the subproblems associated with the subsets.
- Conquer: Take the solutions to the subproblems and "merge" them into a solution to the original problem.

Merge-Sort

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Merge-sort is based on divide-and-conquer.

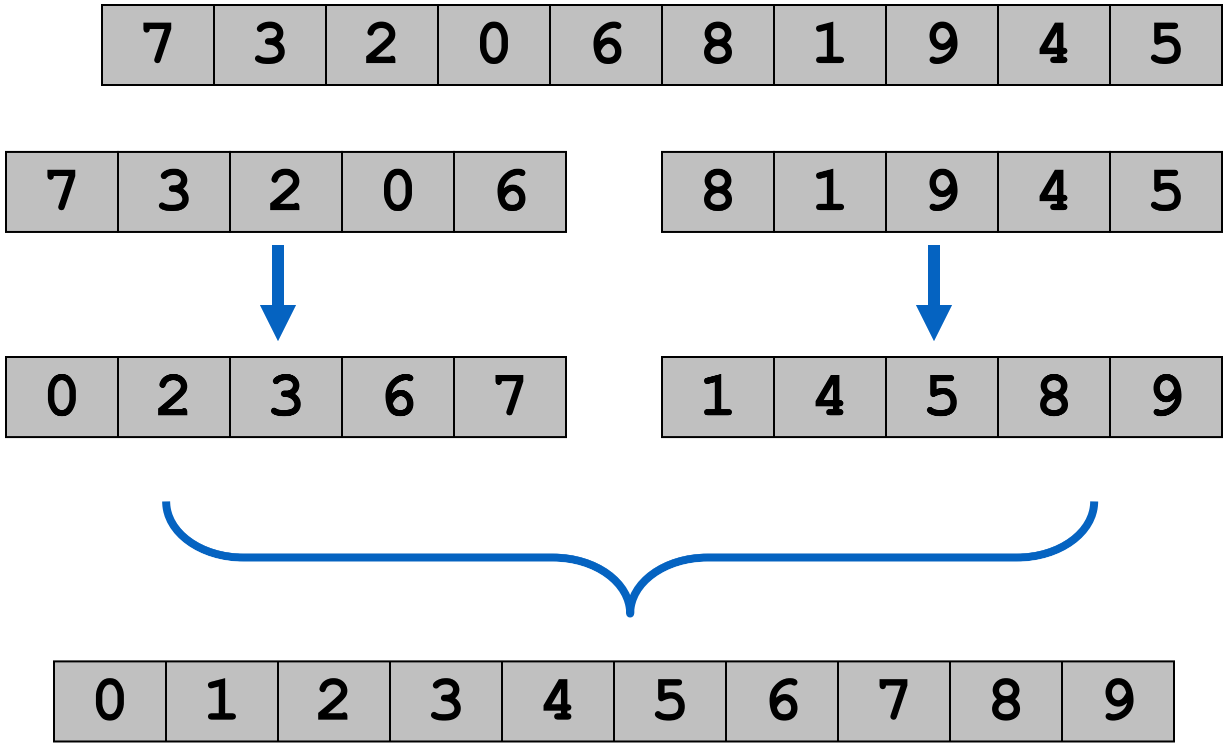
- Divide:
  - (base case) if $n \leq 1$: the sequence is sorted.
  - otherwise let $k = \lfloor n/2 \rfloor$
  - partition $A = (a_1, a_2, \dots, a_n)$ into $L = (a_1, \dots, a_k)$ and $R = (a_{k+1}, \dots, a_n)$.
- Recurse:
  - Sort L into B_L; sort R into B_R
- Conquer:
  - Merge the two sequences B_L and B_R into a sorted sequence B
  - Since B_L and B_R are sorted, the merging can be done in $O(n)$ time

Merge-Sort(A)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Input**: $A = (a_1, a_2, \dots, a_n)$
- **If** $(n \leq 1)$: **return** A
- **If** $(n \geq 2)$:
  - let $k \leftarrow \lfloor n/2 \rfloor$
  - partition $A = (a_1, \dots, a_n)$ into $L = (a_1, \dots, a_k)$ and $R = (a_{k+1}, \dots, a_n)$
  - $B_L \leftarrow$ Merge-Sort(L)
  - $B_R \leftarrow$ Merge-Sort(R)
  - $B \leftarrow$ Merge(B_L, B_R)
  - **return** B

Example

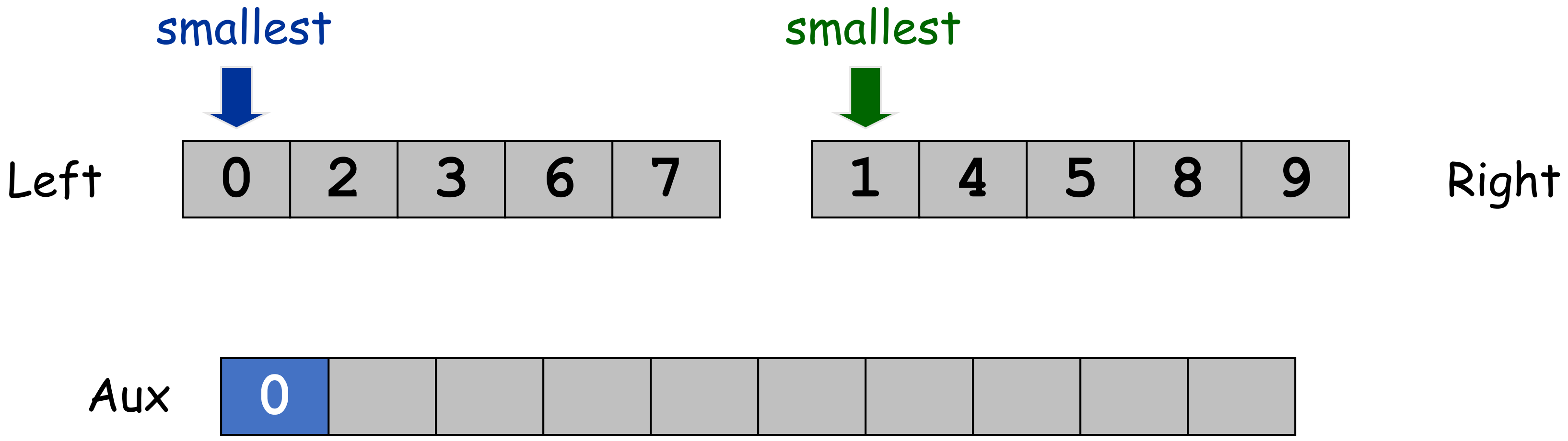


Divide: $O(1)$ time

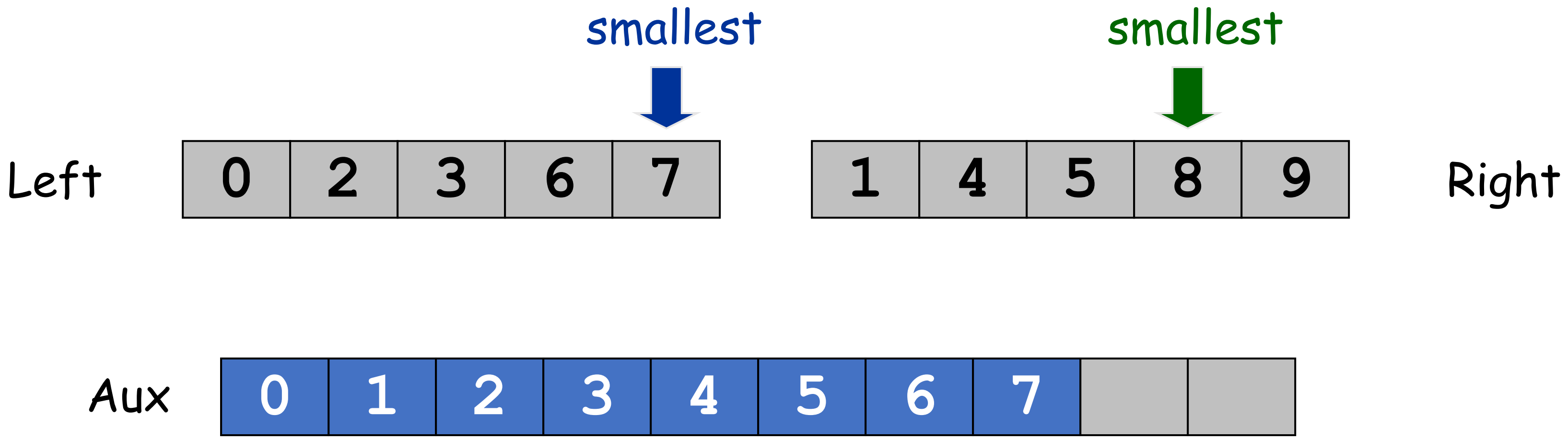
Sort recursively

Merge: $O(n)$ time

Merging



Merging



Merge(X, Y)

- **Input**: Sorted sequences $X = (x_1, \ldots, x_a)$ and $Y = (y_1, \ldots, y_b)$
- assume $x_{a+1} = y_{b+1} = \infty$
- let $i \leftarrow 1$ and $j \leftarrow 1$
- **for** $(k = 1,2, \ldots a + b)$:
  - if $(x_i \leq y_j)$: $z_k \leftarrow x_i, i \leftarrow i + 1$
  - if $(x_i > y_j)$: $z_k \leftarrow y_j, j \leftarrow j + 1$
- **return** $(z_1, \ldots, z_{a+b})$

Complexity of Merge

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Lemma. Given any two sorted arrays X and Y, Merge(X, Y) finishes in $O(|X| + |Y|)$ time.
- Proof.
  - Consider the value of pointers $i + j$
  - initially $i + j = 2$ and after each for-loop $i + j$ increases by one
  - $i + j$ is at most $|X| + |Y| + 2$
  - each for-loop executes in $O(1)$ time
  - the whole algorithm finishes in $O(|X| + |Y|)$ time

Complexity of Merge-Sort

- Let $T(n)$ be the complexity with input size n.
  - $T(n)$ is the number of comparisons
- For all $n \geq 2$ we have (where $c \geq 1$ is a constant)
- $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c \cdot n.$

- **Claim:** $T(n) \leq c \cdot n \cdot \log_2 n = O(n \log n)$

Complexity of Merge-Sort

- **Claim:** $T(n) \le c \cdot n \cdot \log_2 n.$

- **Proof 1.**

- $$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n = 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + \frac{c \cdot n}{2}\right) + c \cdot n$$
  $$= 4 \cdot T\left(\frac{n}{4}\right) + 2c \cdot n = 8 \cdot T\left(\frac{n}{8}\right) + 3c \cdot n = \cdots$$
  $$= \frac{n}{2} \cdot T(2) + \left(\log \frac{n}{2}\right) \cdot c \cdot n \le c \cdot n \cdot \log n.$$

Mathematical Induction

- Prove that: $n^3 - n$ is divisible by 3 for every $n \geq 2$.
- Let $p(n)$ be the above statement.

- Suppose $p(k)$ is true, we can prove $p(k+1)$ easily:
  - $(k+1)^3 - (k+1) = k^3 + 3k^2 + 3k + 1 - k - 1$
  - $= (k^3 - k) + 3(k^2 + k)$
  - $(k^3 - k)$ is divisible by $p(k)$,
  - $3(k^2 + k)$ is obviously divisible by 3
- Statement $p(k+1)$ is also true.

Mathematical Induction

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Statement: $P(n)$ for all $n \in \boldsymbol{N}$.
- **Induction Principle:**
- $\left(P(1) \wedge \left(\forall k \in \boldsymbol{N}: P(k) \rightarrow P(k+1)\right)\right) \rightarrow \forall n \in \boldsymbol{N}: P(n)$.
- **Proof Steps:**
  - Base case: prove statement $P(1)$
  - Induction step:
    - **(Induction Hypothesis)** Assume $P(k)$ is true for some $k \in N$
    - Prove $P(k+1)$.
  - Conclusion: the statement $P(n)$ holds for all $n \in \boldsymbol{N}$.

Complexity of Merge-Sort

- **Claim:** $T(n) \leq c \cdot n \cdot \log_2 n.$

- **Proof 2.** (Mathematical Induction)
- Base Case: $n = 2$: $T(n) = 1 \leq c \cdot n \cdot \log_2 n$
- Induction hypothesis: $T(k) \leq c \cdot k \cdot \log k$ for all $k \leq n - 1$
- Induction step: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n \leq c \cdot n \cdot \log\left(\frac{n}{2}\right) + c \cdot n = n \cdot \log_2 n$
- By M.I., we have $T(n) \leq c \cdot n \cdot \log_2 n$ for all $n \geq 2$.

Quick-Sort

Quick-sort is based on divide-and-conquer.

- Divide:
  - (base case) if $n \leq 1$: the sequence is sorted.
  - otherwise pick a **pivot** k and partition $A = (a_1, a_2, \ldots, a_n)$ into
    - $L = \{a_i \in A : a_i < k\}$ and
    - $M = \{a_i \in A : a_i = k\}$
    - $R = \{a_i \in A : a_i > k\}$.
  - $\Rightarrow O(n)$ time
- Recurse: Sort L into B_L; sort R into B_R $\quad\Rightarrow O(?)$ time
- Conquer: Return $B = B_L + M + B_R$ $\quad\Rightarrow O(1)$ time

Quick-Sort(A)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Input**: $A = (a_1, a_2, \dots, a_n)$
- **If** $(n \leq 1)$: **return** A
- **If** $(n \geq 2)$:
  - $k \leftarrow$ PickPivot(A)
  - initialize $L \leftarrow \emptyset, M \leftarrow \emptyset, R \leftarrow \emptyset$
  - **For** $(i = 1,2, \dots, n)$
    - put a_i in (L / M / R) if ($a_i < k$ / $a_i = k$ / $a_i > k$)
  - $B_L \leftarrow$ Quick-Sort(L), $B_R \leftarrow$ Quick-Sort(R)
  - $B \leftarrow B_L + M + B_R$
  - **return** B

PickPivot(A)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

How to pick the pivot?

- Median
  + produces perfectly balanced recursions
  - time consuming to find median (best algorithm: $\Omega(n)$)
- First element
  + very convenient
  - may produce highly unbalanced recursions when A is structured
- Random element
  + efficient and produce balanced recursions on average

Quick-Sort(A)

- PickPivot(A): pick a random element

- **Theorem**: The expected running time of randomized Quick-Sort on a sequence of n elements is $O(n \log n)$.
- **Justification:**
  - $T(n) = T(|L|) + T(|R|) + O(n)$
  - The probability of having $|L| < n/4$ or $|R| < n/4$ is low
  - It is fine to have some unbalanced recursions

Quick-Sort(A)

- PickPivot(A): pick a random element
- **Theorem**: The expected running time of randomized Quick-Sort on a sequence of n elements is $O(n \log n)$.
- Worst Case complexity is still $O(n^2)$
- In practice: median of three random elements

Linear-time Sorting

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Input**: a sequence of numbers $A = (a_1, a_2, \dots, a_n)$, where all numbers are integers in $\{1,2,\dots,k\}$, for some $k < n$.
- E.g., sorting a group of people by ages.

- Linear-time Sorting: $\Rightarrow O(k+n) = O(n)$ time
  - Initialize array $L[1,\dots,k]$, each entry corresponds to an initially empty linked list. $\Rightarrow O(k)$ time
  - Insert each element $a_i = t$ into linked list $L[t]$. $\Rightarrow O(n)$ time
  - Output L as a sorted array. $\Rightarrow O(n)$ time

Linear-time Sorting

- Is it possible to have an $O(n)$ time sorting algorithm for general input instances? **NO !**

- Theorem: The running time of any comparison-based algorithm for sorting an n-element sequence is $\Omega(n \log n)$ in the worst case
- **Justification**: given $A = (a_1, \ldots, a_n)$
  - Total number of possible outputs: $n! = n \times (n-1) \times \cdots \times 2 \times 1$.
  - Each comparison eliminates at most half of the possible outputs

Linear-time Sorting

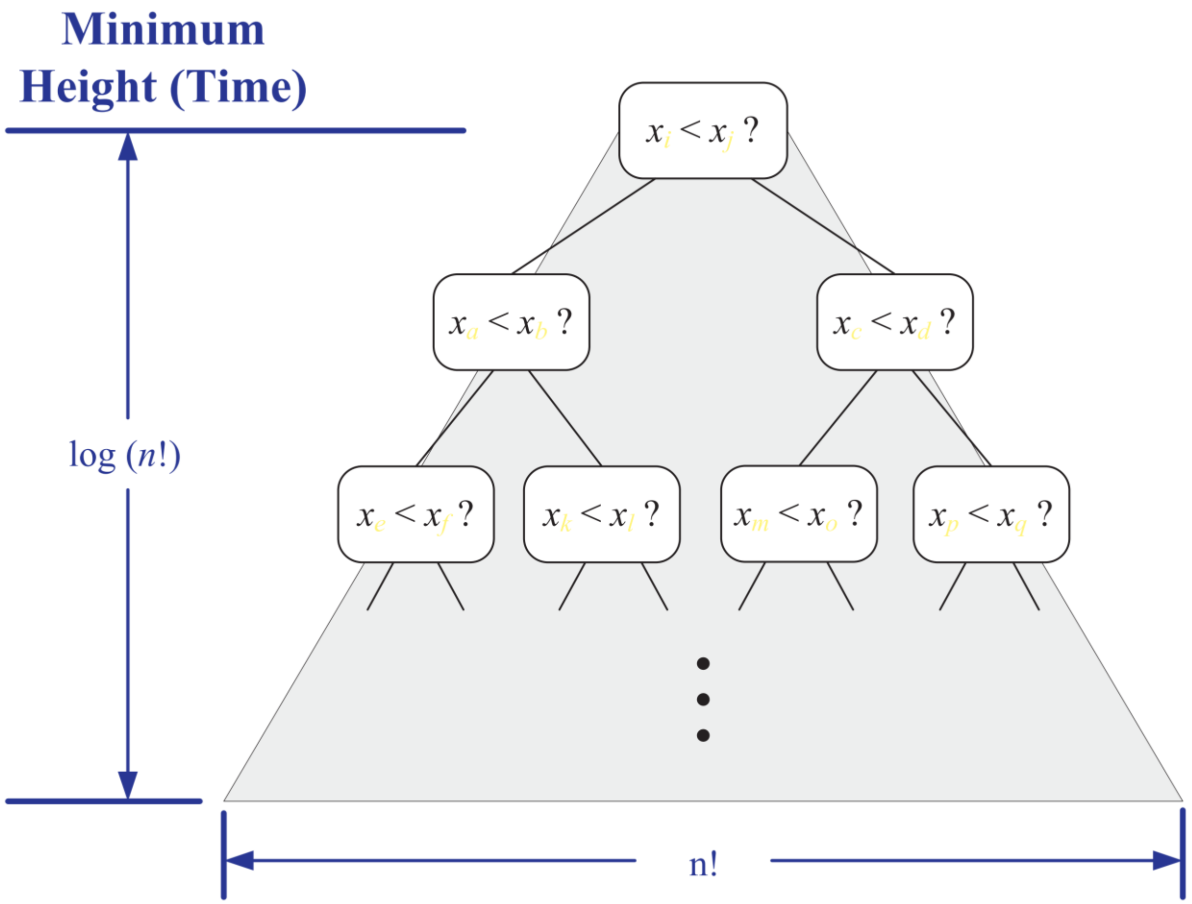
- Decision tree

- Comparisons needed:

$$\log(n!) = \Omega(n \log n)$$

  - Stirling's approximation

$$n! \approx \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$$



Algorithm Design Techniques

Problems

- **Decision Problem**
  - Given an instance of the problem, decides YES/NO
  - **Example**: "Given x, decide if it is a prime number."

- **Optimization Problem**
  - **Constraints**: any feasible solution must satisfy
  - **Objective**: maximize/minimize some function of a solution
  - **Example**: Knapsack problem

Optimization Problem

- **Optimization Problem**
  - **Constraint**: feasibility of solution
  - **Objective**: maximize/minimize ….
- Feasible Solution: a solution that meets all constraints
- Optimal Solution: feasible solution with best objective
- Optimal Objective: objective of the optimal solution
  - For a maximization problem: if there is no feasible solution, then $\mathrm{OPT} = -\infty$
  - For a minimization problem: if there is no feasible solution, then $\mathrm{OPT} = +\infty$

Divide and Conquer

- Divide and Conquer
  - Break up a problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.
- Most common usage
  - Break up a problem of size n into two equal parts of size $\frac{1}{2}n$.
  - Combine two solutions into overall solution in linear time.
  - Typical time complexity: $O(n \log n)$.
- Example: Merge-Sort

Greedy Algorithms

Problem: among a collection of elements, select a subset of them to maximize/minimize some objective

Greedy Algorithm

- Fix some ordering of the elements.
- Consider the element in this order one-by-one.
- Select an element if
  (1) it does not cause infeasibility;
  (2) selecting the element gives a better objective

A Schedule Problem

- **Problem**: n professors want to use the same classroom. Each of them has a time interval during which he wants to use the place. Find the largest non-conflicting subset of professors.

- **Input**: A set N of n professors such that
  - Each professor $i \in N$ has interval (s_i, f_i) for the use of classroom.
- **Output**: Subset $A \subseteq N$ such that
  - (constraint) for any two different $i, j \in A$, $(s_i, f_i) \cap (s_j, f_j) = \emptyset$.
  - (objective) the size $|A|$ is as large as possible

Greedy Algorithms

- Fix some ordering of the intervals.
- Consider the intervals in this order one-by-one.
- Include an interval if it does not cause infeasibility.
  - Accept those compatible with the ones already accepted.

Greedy Algorithms

- [Earliest start time first]
  - Consider intervals in ascending order of start time s_j.
- **[Earliest finish time first]**
  - Consider intervals in ascending order of finish time f_j.
- [Shortest interval first]
  - Consider intervals in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts first]
  - For each interval, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

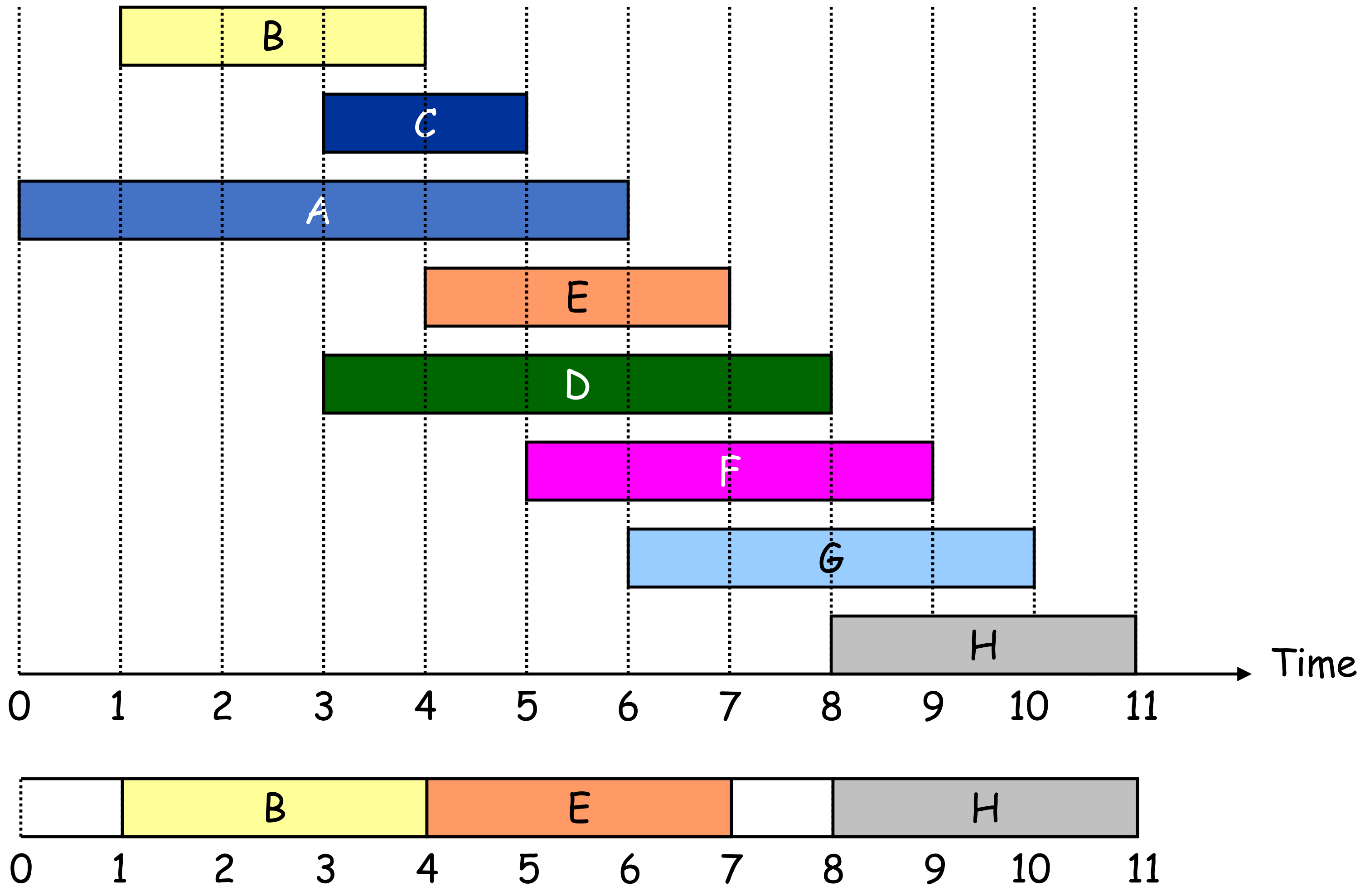
Greedy Algorithm

- Sort the intervals in increasing order of finish time.
  - "earliest finish time first"
- Consider the intervals one by one; accept an interval if it is compatible with the ones already accepted.

- **Theorem**: The above Greedy algorithm always computes an optimal solution, for any given input.

Greedy Algorithm

澳 門 大 學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU



Observation

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Claim**: there must exists an optimal solution that includes the interval with minimum finish time.

- **Proof Sketch**.
  - Fix any optimal solution and suppose it does not contain interval 1 (the interval with minimum finish time).
  - Remove the interval in the solution with minimum finish time.
  - Include interval 1.
  - The resulting solution must be feasible and optimal.

Observation

- **Theorem**: Greedy algorithm computes an **optimal solution.**

- **Proof.**
  - There exists an optimal solution containing interval 1.
  - Intervals conflicting with interval 1 do not appear in this optimal solution. We can remove these intervals.
  - //It remains to compute an optimal solution for the resulting instance
  - For the resulting instance, there exists an optimal solution containing the interval with minimum finish time.

Greedy Algorithms

- **Most common usage**
  - Consider the elements of the problem one by one.
  - Take/remove each element if it helps improve the solution.

- Greedy algorithms are usually efficient and easy to implement.
- Greedy algorithms are not always optimal.

Classic Greedy Algorithms

- **Optimal**:
  - Scheduling algorithms
  - Kruskal's algorithm, Prim's algorithm
  - Caching algorithms
- **Good Approximations**:
  - Greedy matching
  - Set cover problem
  - Online load balancing

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

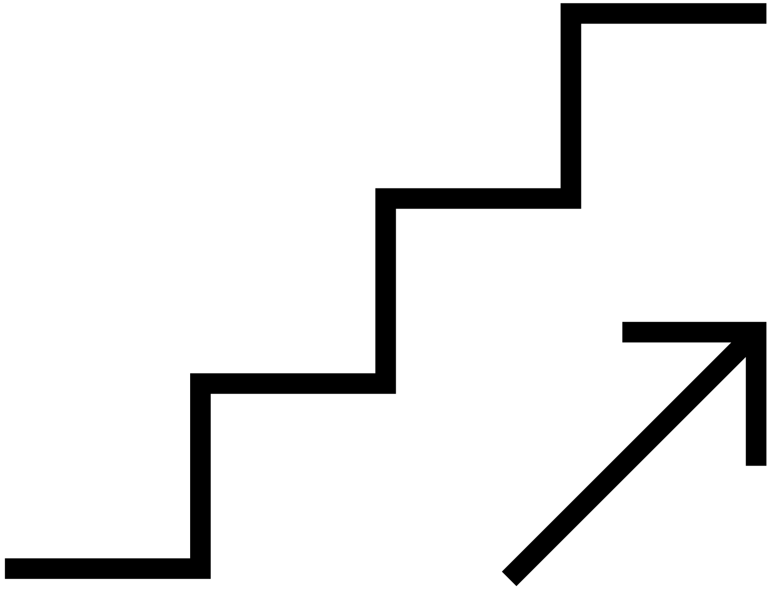
Dynamic Programming

Number of Walks

- Walk from stair 0 to stair n
- Step size: 1 and 2

- **Question:** how many ways?

- **Example:** n=5, f(n)=8



Number of Walks

- Let $f(n)$ be the number of ways to reach stair n.
  - Number of sequences of $\{1,2\}$ that sum to n.

- We have: $f(n) = f(n-1) + f(n-2)$

number of ways to reach stair n - 2

number of ways to reach stair n - 1

Number of Walks

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Let f(n) be the number of ways to reach stair n.
  - Number of sequences of {1,2} that sum to n.

- To compute f(n): ~~recursion?~~ Record f(1), f(2), …, f(n-1).

Number of Walks

- **General Problem:** compute the number f(n) of sequences of $\{s_1, s_2, \ldots, s_k\}$ that sum to n.
  - Target number: n
  - Step sizes: $s_1, s_2, \ldots, s_k$

- **Observation:** $f(n) = \sum_{i \le k : s_i \le n} f(n - s_i)$

Number of Walks

- **General Problem:** compute the number f(n) of sequences of $\{s_1, s_2, \ldots, s_k\}$ that sum to n.
    - Target number: n
    - Step sizes: $s_1, s_2, \ldots, s_k$

- **Observation:** $f(n) = \sum_{i=1}^{k} f(n - s_i)$
- Base case: $f(0) = 1$; $f(x) = 0$ if $x < 0$.

- **Solution:** compute f(1), then f(2), then f(3), ...

Number of Walks

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **General Problem**: compute the number f(n) of sequences of $\{s_1, s_2, \dots, s_k\}$ that sum to n.
  - Target number: n
  - Step sizes: $s_1, s_2, \dots, s_k$

- **Observation**: $f(n) = \sum_{i \le k : s_i \le n} f(n - s_i)$
- Base case: $f(0) = 1$

- **Solution**: compute $f(i)$ for $i = 1,2, \dots, n$

Best Walk

- **General Problem**: compute the length of shortest sequence of $\{s_1, s_2, \ldots, s_k\}$ that sums to n.
  - Target number: n
  - Step sizes: $s_1, s_2, \ldots, s_k$

- **Example**: $n = 33$, $s_1 = 3$ and $s_2 = 7$:
- **Answers**: 7 (steps): (7,7,7,3,3,3,3)

Change Making

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Given**: coins of integer values $s_1, s_2, \dots, s_k$
  - Each coin has infinitely many of them
- **Target value**: n
- **Problem**: compute the minimum number $f(n)$ of coins that sum to exactly n.

- **Recursion**: $f(n) = 1 + \min\{f(n - s_i) : s_i \leq n\}$
- **Base Case**: $f(0) = 0$

Change Making

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Given**: coins of integer values $s_1, s_2, \ldots, s_k$
  - Each coin has infinitely many of them
- **Target value**: n
- **Problem**: compute the minimum number $f(n)$ of coins that sum to exactly n.

- **Recursion**: $f(n) = 1 + \min\{f(n - s_i) : i \leq k\}$
- **Base Case**: $f(0) = 0$, $\forall x < 0$: $f(x) = +\infty$

Pseudo-code

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU
澳門大學

- let n be the target value and $s_1, s_2, \dots, s_k$ be the coin values
- initialize $f(0) \leftarrow 0$
- **for** ($i = 1,2, \dots, n$):
  - initialize $f(i) \leftarrow \infty$
  - **for** ($j = 1, \dots, k$):
    - **if** ($s_j \le i$): $f(i) \leftarrow \min\{f(i), 1 + f(i - s_j)\}$
- **return** $f(n)$

Knapsack Problem

- **Knapsack** of capacity C
- **Items** N: each item $i \in N$ has a value v_i and a size s_i
- **Problem**: find a subset $A \subseteq N$ of items with total size at most C, such that the total value is maximized.
    - Size constraint: $s(A) = \sum_{i \in A} s_i \leq C$
    - Objective: maximize $v(A) = \sum_{i \in A} v_i$

- **Assumption**: all sizes and the capacity are integers.

Example

- **Knapsack** of capacity $C = 10$
- **Items**: each item i is represented by (v_i, s_i)
  - $N = \{(3,2), (2,3), (4,5), (1,3), (6,4), (7,6)\}$

    1 2 3 4 5 6

- Question: should we take item 6 ?
  - What is the best solution if we take it?
  - What is the best solution if we don't?
  - Choose the better solution between the two.

Example

- What is the **optimal solution** if we take 6 ?
  - Remaining items:

    1 2 3 4 5

  - Remaining capacity: $C - s_6 = 10 - 6 = 4$

- Compute the **optimal solution** $B \subseteq \{1,2,3,4,5\}$ for the sub-problem with capacity 4.
- Our solution: $A = B \cup \{6\}$ (with value $v(A) = v(B) + v_6$)

Example

- What is the **optimal solution** if we take 6 ?
  - Remaining items:

    $\{(3,2),(2,3),(4,5),(1,3),\boxed{(6,4)}\}$

  - Remaining capacity: $C - s_6 = 10 - 6 = 4$

- Compute the **optimal solution** $B = \{5\}$ for the sub-problem with capacity 4.
- Our solution: $A = B \cup \{6\}$ (with value $v(A) = v(B) + v_6$)

Example

- What is the **optimal solution** if we don't take 6 ?
  - Remaining items:

    1 2 3 4 5

  - Remaining capacity: $C = 10$

- Compute the **optimal solution** $B \subseteq \{1,2,3,4,5\}$ for the sub-problem with capacity 10.
- Our solution: $A = B$ (with value $v(A) = v(B)$)

Example

- What is the **optimal solution** if we don't take 6 ?
  - Remaining items:

$$\{\boxed{(3,2),(2,3)},(4,5),(1,3),\boxed{(6,4)}\}$$

  - Remaining capacity: $C = 10$

- Compute the **optimal solution** $B = \{1,2,5\}$ for the sub-problem with capacity 10.
- Our solution: $A = B$ (with value $v(A) = v(B)$)

Example

- **Optimal solution** if we take 6 :
- $A = \{5, 6\}$ (with value $v(A) = v_5 + v_6 = 13$)

- **Optimal solution** if we don't take 6 :
- $A = \{1,2,5\}$ (with value $v(A) = 11$)

Knapsack Problem

- **Knapsack** of capacity C
- **Items** $N = \{1,2,\ldots,n\}$: each $i \in N$ has a value v_i and a size s_i

- Let $f(i,b)$ be the value of optimal solution
  - on items $\{1,2,\ldots,i\}$ $(i \leq n)$
  - with capacity b $(b \leq C)$

Knapsack Problem

- Let $f(i, b)$ be the value of optimal solution
  - on items $\{1,2, \dots, i\}$ $(i \leq n)$
  - with capacity b $(b \leq C)$
- If $b \geq s_i$:

$$\boxed{f(i, b) = \max\{f(i-1, b-s_i) + v_i, f(i-1, b)\}}$$

- If $b < s_i$: $\boxed{f(i, b) = f(i-1, b)}$

- **Base Case**: $f(i, b) = 0$ if $i = 0$

Pseudo-code

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- let C be the capacity and (v_i, s_i) be size and value of item i
- initialize $f(0,b) \leftarrow 0$ for all $b = 0,1,\ldots,C$
- **for** $(i = 1,2,\ldots,n)$:
    - **for** $(b = 0,1,\ldots,C)$:
        - **if** $(b \geq s_i)$: $f(i,b) = \max\{f(i-1,b-s_i)+v_i, f(i-1,b)\}$
        - **else**: $f(i,b) = f(i-1,b)$
- **return** $f(n,C)$

Example

	Capacity									
	1	2	3	4	5	6	7	8	9	10
1 (3,2)	0	3	3	3	3	3	3	3	3	3
2 (2,3)	0	3	3	3	5	5	5	5	5	5
3 (4,5)	0	3	3	3	5	5	7	7	7	9
4 (1,3)	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	9
5 (6,4)	0	3	3	6	6	9	9	9	11	11
6 (7,6)	0	3	3	6	6	9	9	10	11	$\boxed{13}$

items

"obtained by taking item 6"

Example

Capacity

items	1	2	3	4	5	6	7	8	9	10
5 (6,4)	0	3	3	6	6	9	9	9	11	11
6 (7,6)	0	3	3	6	6	9	9	10	11	13

Maintain a table $d(i, b)$ such that

- $d(i, b) =$ Y if $f(i, b)$ is achieved by taking item i
- $d(i, b) =$ N if $f(i, b)$ is achieved by not taking item i

Example

items		Capacity									
		1	2	3	4	5	6	7	8	9	10
	1 (3,2)	N	y	y	y	y	y	y	y	y	y
	2 (2,3)	N	N	N	N	y	y	y	y	y	y
	3 (4,5)	N	N	N	N	N	N	y	y	y	y
	4 (1,3)	N	N	N	N	N	N	N	N	N	N
	5 (6,4)	N	N	N	y	y	y	y	y	y	y
	6 (7,6)	N	N	N	N	N	N	N	y	N	y

Example

items	Capacity 1	2	3	4	5	6	7	8	9	10
1 (3,2)	N	y	y	y	y	y	y	y	y	y
2 (2,3)	N	N	N	N	y	y	y	y	y	y
3 (4,5)	N	N	N	N	N	N	y	y	y	y
4 (1,3)	N	N	N	N	N	N	N	N	N	N
5 (6,4)	N	N	N	y	y	y	y	y	y	y
6 (7,6)	N	N	N	N	N	N	N	y	N	y

Pseudo-code

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- let C be the capacity and (v_i, s_i) be size and value of item i
- initialize $f(0, b) \leftarrow 0$ for all $b = 0,1, \ldots, C$
- **for** $(i = 1,2, \ldots, n)$:
  - **for** $(b = 0,1, \ldots, C)$:
    - **if** $(b \geq s_i)$: $f(i, b) = \max\{f(i-1, b-s_i) + v_i, f(i-1, b)\}$
      - **if** $f(i-1, b-s_i) + v_i > f(i-1, b)$: $d(i, b) \leftarrow Y$
      - **else**: $d(i, b) \leftarrow N$
    - **else**: $f(i, b) = f(i-1, b)$ and $d(i, b) \leftarrow N$
- print_solution(d, n, C) and **return** $f(n, C)$

Pseudo-code

print_solution(d, i, b)

- **if** ($i = 0$ or $b = 0$)
  - **return**
- **Elseif** $d(i, b) = Y$
  - print i
  - print_solution($d, i-1, b-s_i$)
- **Elseif** $d(i, b) = N$
  - print_solution($d, i-1, b$)

Longest Common Subsequence

- **Problem:** Given two strings X and Y, compute the longest string Z that is a common subsequence of X and Y.
  - A subsequence Z of a string X is a string that can be obtained by removing some letters from X.

- **Example:** X=CTGACA and Y=ACGCTAC
  - Z=CGA is a common subsequence
  - Z=CGAC is the longest common subsequence (How to find it?)

Observations

- $X = X_n = (x_1, x_2, \dots, x_n)$ and $Y = Y_m = (y_1, y_2, \dots, y_m)$
- **Prefix**: $X_i = (x_1, x_2, \dots, x_i)$ and $Y_j = (y_1, y_2, \dots, y_j)$
- Length of LCS of X_i and Y_j: $c(i, j)$

- **Observation 1.** If $x_i = y_j$ then

$$c(i, j) = c(i-1, j-1) + 1. \quad \text{(proof?)}$$

Observations

- $X = X_n = (x_1, x_2, \dots, x_n)$ and $Y = Y_m = (y_1, y_2, \dots, y_m)$
- **Prefix**: $X_i = (x_1, x_2, \dots, x_i)$ and $Y_j = (y_1, y_2, \dots, y_j)$
- Length of LCS of X_i and Y_j: $c(i,j)$

- **Observation 2.** If $x_i \neq y_j$ then

$$c(i,j) = \max\{c(i,j-1), c(i-1,j)\}.$$ (proof?)

Observations

- If $x_i = y_j$ then

$$c(i,j) = c(i-1,j-1) + 1.$$

- If $x_i \neq y_j$ then

$$c(i,j) = \max\{c(i,j-1), c(i-1,j)\}.$$

- Base case: $c(i,0) = c(0,j) = 0$, for all i,j

Example

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU
澳門大學

	•	A	C	G	C	T	A	C
•	0	0	0	0	0	0	0	0
C	0	0	1	1	1	1	1	1
T	0	0	1	1	1	2	2	2
G	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
C	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Observations

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- If $x_n = y_m$ then

$$c(n,m) = c(n-1, m-1) + 1.$$

$$b(n,m) = \nwarrow$$

- If $x_n \neq y_m$ then

$$c(n,m) = \max\{c(n, m-1), c(n-1, m)\}.$$

$$b(n,m) = \leftarrow \qquad\qquad b(n,m) = \uparrow$$

Example

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU
澳門大學

	•	A	C	G	C	T	A	C
•	0	0	0	0	0	0	0	0
C	0	↑	↖	←	↖	←	←	↖
T	0	↑	↑	↑	↑	↖	←	←
G	0	↑	↑	↖	←	↑	↑	↑
A	0	↖	↑	↑	↑	↑	↖	←
C	0	↑	↖	↑	↖	←	↑	↖
A	0	↖	↑	↑	↑	↑	↖	↑

Example

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

	•	A	C	G	C	T	A	C
•	0	0	0	0	0	0	0	0
C	0	↑	↖	←	↖	←	←	↖
T	0	↑	↑	↑	↑	↖	←	←
G	0	↑	↑	↖	←	↑	↑	↑
A	0	↖	↑	↑	↑	↑	↖	←
C	0	↑	↖	↑	↖	←	↑	↖
A	0	↖	↑	↑	↑	↑	↖	↑

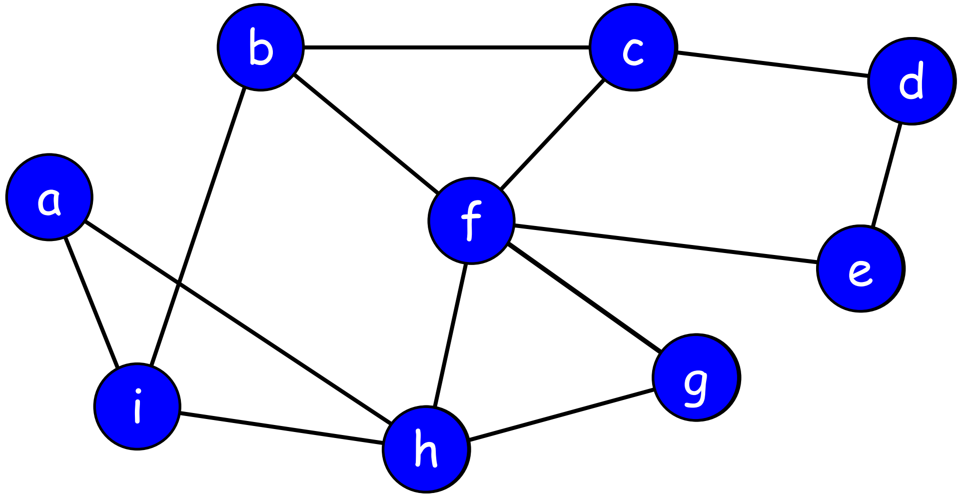
General Framework for DP

- Define the function to capture the problem
  - e.g., $f(i, b)$ for the Knapsack problem
- Formulate the recursion between the function values on different inputs
  - e.g., $f(i, b) = \max\{f(i-1, b-s_i) + v_i, f(i-1, b)\}$
- Compute the value table

Graphs

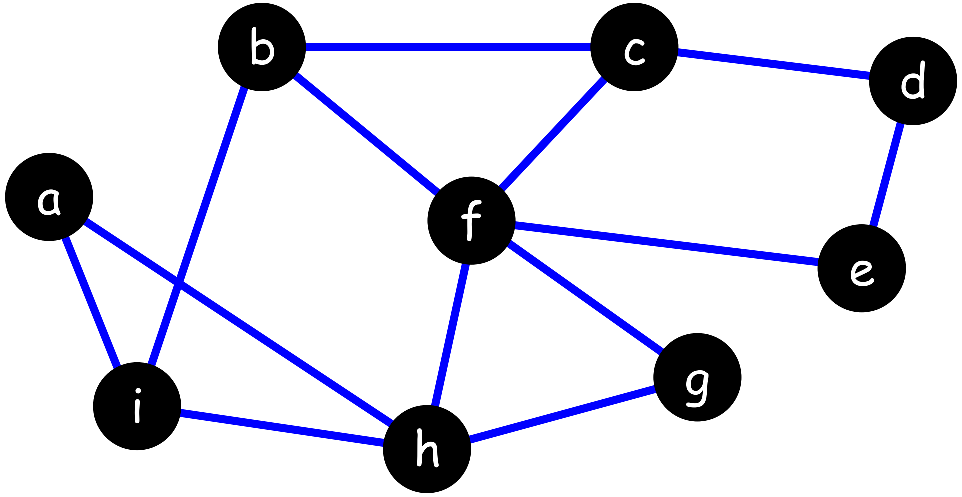
Graphs

- **Nodes:** $V = \{a, b, c, d, e, f, g, h, i\}$
- **Edges**: relations between pairs of objects



Graphs

- **Nodes:** $V = \{a, b, c, d, e, f, g, h, i\}$
- **Edges:** $E = \{(a,i), (a,h), (b,i), (b,f), (b,c), (c,d), (c,f), (d,e), (e,f), (f,g), (f,h), (g,h), (h,i)\}$



Graphs

- **Graph** $G(V, E)$ on nodes V and edges $E \subseteq V \times V$.

- **Examples:**
- Undirected graph: Facebook network.
  - $(u, v) \in E$ if users u and v are friends of each other.
- Directed graph: Twitter network
  - $(u, v) \in E$ if u follows v.

Graphs

- **Graph** $G(V, E)$ on nodes V and edges $E \subseteq V \times V$.

- **More Examples of Undirected Graphs:**
- Co-authorship network:
  - Nodes: authors;
  - Edges: $(u, v) \in E$ if u and v are co-authors.
- Road network:
  - Nodes: cities;
  - Edges: $(u, v) \in E$ if there is a rail connecting cities u and v.

Basic Concepts of Graphs

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- If $(u, v) \in E$:
  - u and v are **adjacent**
  - u and v are **neighbors** of each other

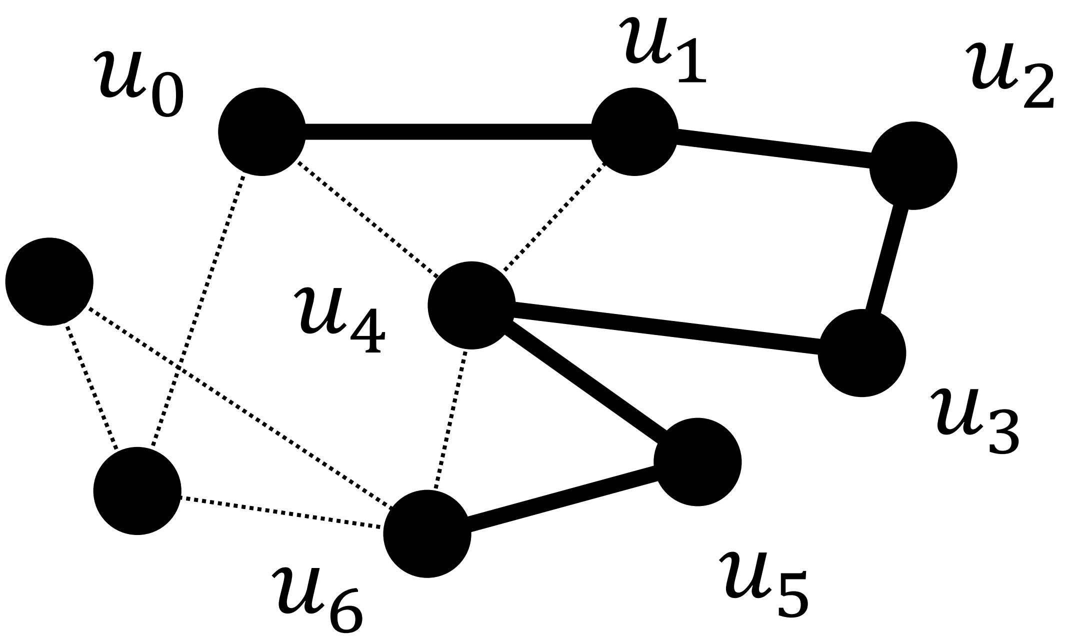
- Set of neighbors $N(u) = \{v \in V: (u, v) \in E\}$ of node u.
- Degree of node u: $d(u) = |N(u)|$.

- **Claim**: $\sum_{u \in V} d(u) = 2 \cdot |E|$.

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Basic Concepts of Graphs

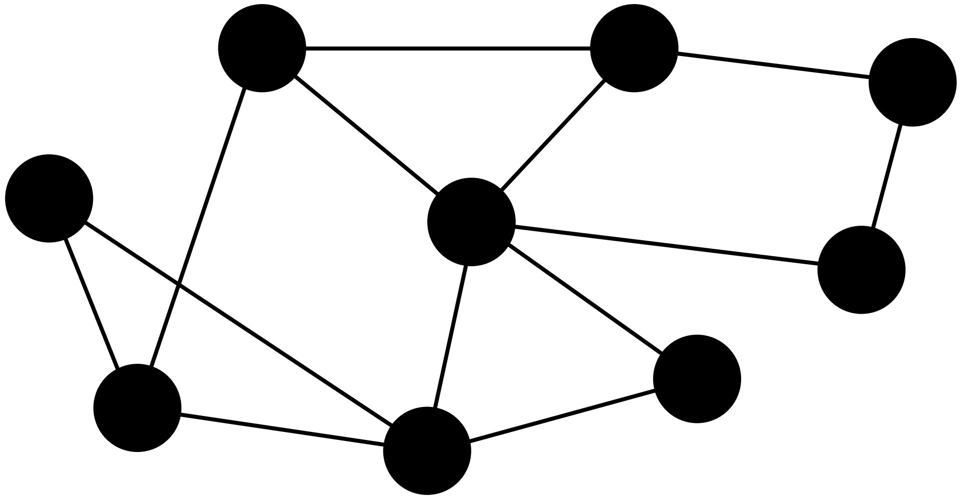
- Path $(u_0, u_1, \dots, u_l)$ connecting node u_0 and node u_l:
  - for all $i = 1,2, \dots, l$, we have $(u_{l-1}, u_l) \in E$
  - l is the length of the path (number of edges)



Basic Concepts of Graphs

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Nodes u and v are connected if there exists a path connecting u and v.
- A graph $G(V, E)$ is connected if any two nodes $u, v \in V$ are connected.

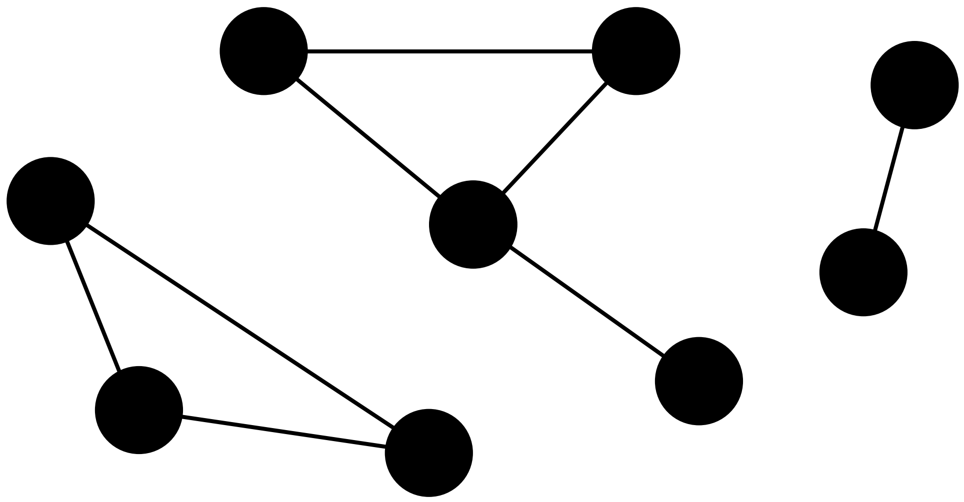


Connected

Basic Concepts of Graphs

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Nodes u and v are connected if there exists a path connecting u and v.
- A graph $G(V,E)$ is connected if any two nodes $u, v \in V$ are connected.

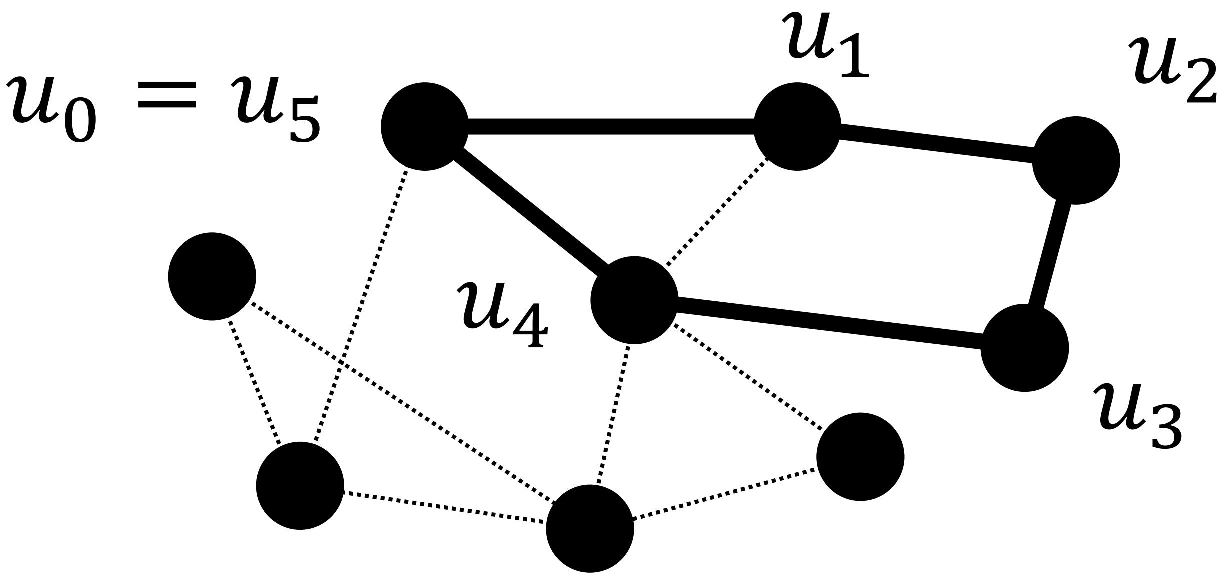


Disconnected

Basic Concepts of Graphs

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Cycle $(u_0, u_1, \dots, u_l)$ is a path with $u_0 = u_l$:
  - for all $i = 1,2, \dots, l$, we have $(u_{i-1}, u_i) \in E$
  - l is the length of the cycle (number of edges)



Graph ADT

- Two basic types: **nodes** and **edges**
  - nodes(): return the set of nodes V
  - edges(): return the set of edges E
  - each node u supports
    - incidentEdges(): return the set of edges incident to u
    - neighbors(): return the set of neighbors of u
    - isNeighbor(v): check whether v is a neighbor of u
  - each edge $e = (u, v)$ supports
    - endNodes(): return the set $\{u, v\}$
    - opposite(u): return node v (the endpoint other than u)
    - isAdjacentTo(f): check whether e and f have common endpoints

Graph ADT

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Two basic types: **nodes** and **edges**
- Update operations:
  - insertNode(u): insert a node u to the graph (without incident edge)
  - insertEdge(u, v): insert an edge between existing nodes u and v
  - removeNode(u): remove node u and all its incident edges
  - removeEdge(u, v): remove the edge between node u and v

Implementation (trivial)

Every graph $G(V, E)$ maintains

- A set of n node objects, e.g., as an array/linked list
- A set of m edge objects, e.g., as an array/linked list

Implementation (trivial)

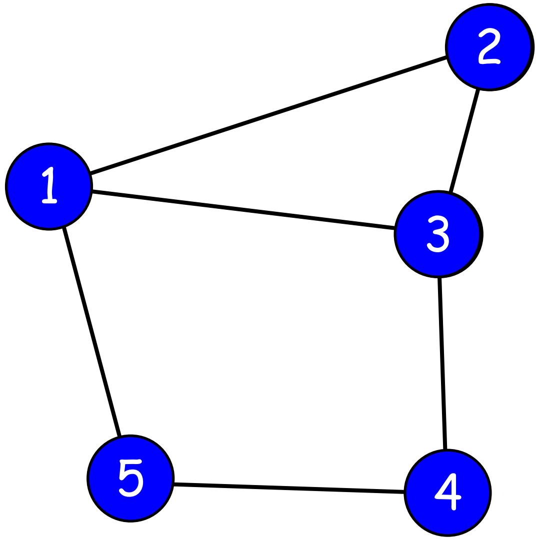
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Two basic types: **nodes** and **edges**
  - nodes(): return the set of nodes V — $O(n)$ time
  - edges(): return the set of edges E — $O(m)$ time
  - each node u supports
    - incidentEdges(): — $O(m)$ time
    - neighbors(): — $O(m)$ time
    - isNeighbor(v): — $O(m)$ time
  - each edge $e = (u, v)$ supports
    - endNodes(): — $O(1)$ time
    - opposite(u): — $O(1)$ time
    - isAdjacentTo(f): — $O(1)$ time

Adjacency Matrix

- Adjacency Matrix $A = \{a_{ij}\}_{V\times V}$: binary $a_{uv} = 1 \Leftrightarrow (u, v) \in E$

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



Adjacency Matrix

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

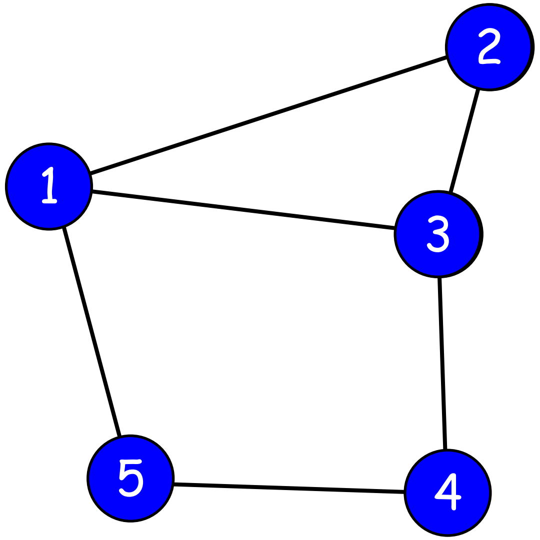
- Two basic types: **nodes** and **edges**
  - nodes(): return the set of nodes V — $O(n)$ time
  - edges(): return the set of edges E — $O(m)$ time
  - each node u supports
    - incidentEdges(): — $O(n)$ time
    - neighbors(): — $O(n)$ time
    - isNeighbor(v): — $O(1)$ time
  - each edge $e = (u, v)$ supports
    - endNodes(): — $O(1)$ time
    - opposite(u): — $O(1)$ time
    - isAdjacentTo(f): — $O(1)$ time

Adjacency Matrix

- Adjacency Matrix $A = \{a_{ij}\}_{V \times V}$: binary $a_{uv} = 1 \Leftrightarrow (u, v) \in E$

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

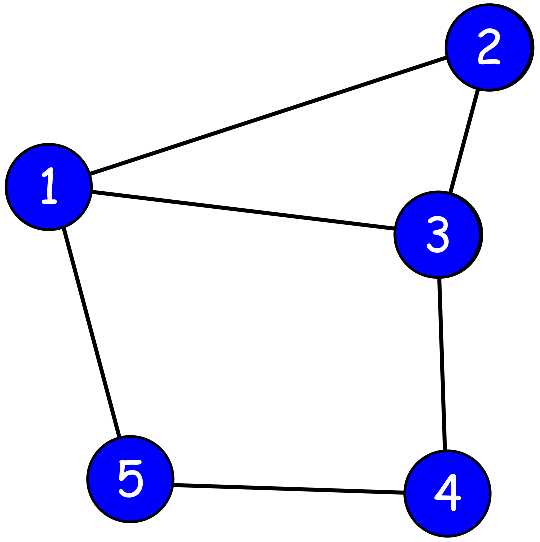
Space complexity : $O(n^2)$



Adjacency List

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Each node u maintains the set of incident-edges/neighbors
- Example:
  - $u_1.neighbors() = \{u_2, u_3, u_5\}$
  - $u_2.neighbors() = \{u_1, u_3\}$
  - $u_3.neighbors() = \{u_1, u_2, u_4\}$
  - $u_4.neighbors() = \{u_3, u_5\}$
  - $u_5.neighbors() = \{u_1, u_4\}$



Space complexity : $O(m)$

Adjacency List

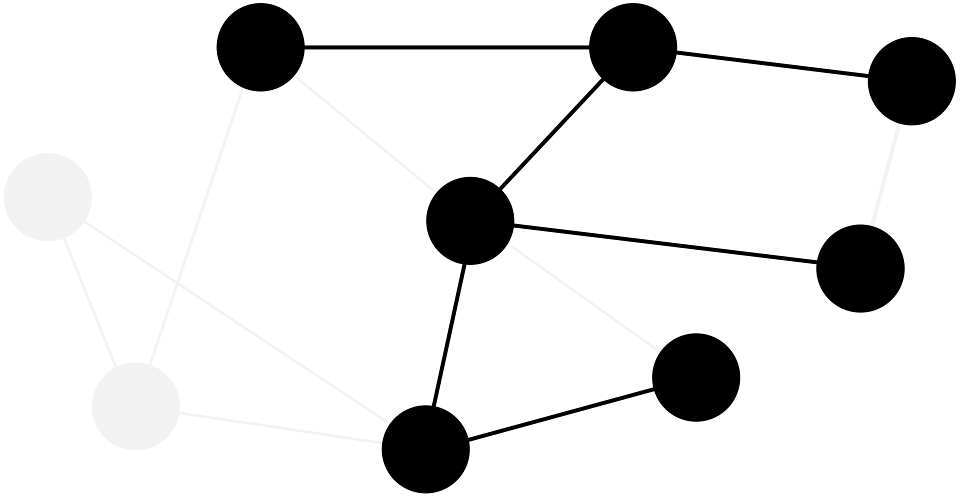
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Two basic types: **nodes** and **edges**
  - nodes(): return the set of nodes V — $O(n)$ time
  - edges(): return the set of edges E — $O(m)$ time
  - each node u supports
    - incidentEdges(): — $O(d(u))$ time
    - neighbors(): — $O(d(u))$ time
    - isNeighbor(v): — $O(d(u))$ time
  - each edge $e = (u, v)$ supports
    - endNodes(): — $O(1)$ time
    - opposite(u): — $O(1)$ time
    - isAdjacentTo(f): — $O(1)$ time

Subgraphs

- Graph $H(V_H, E_H)$ is a subgraph of $G(V, E)$ if
  - $V_H \subseteq V$ and $E_H \subseteq E$

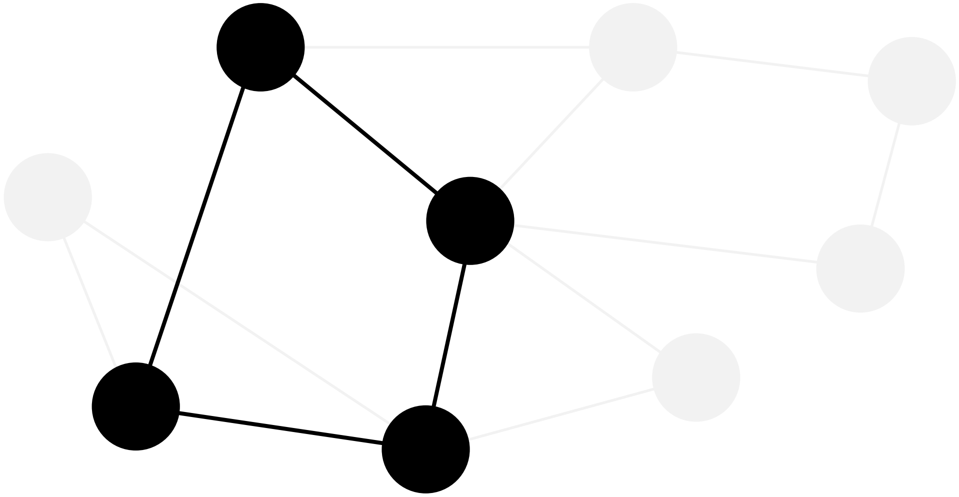
$H(V_H, E_H)$



Induced Subgraphs

- Graph $H(V_H, E_H)$ is an induced subgraph of $G(V, E)$ if
  - $V_H \subseteq V$ and $E_H = \{(u, v) \in E : u \in V_H \text{ and } v \in V_H\}$
  - Notation: $H(V_H, E_H) = G[V_H]$

$H(V_H, E_H)$



Connected Components

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

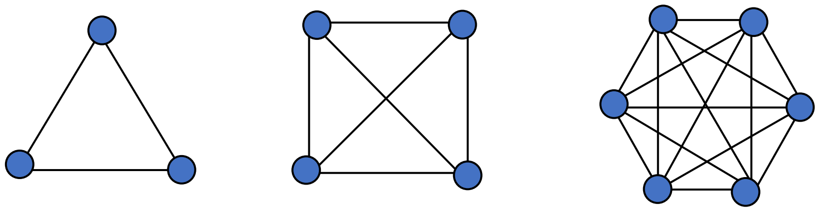
- The connected components of $G(V, E)$ is the collection of all maximal connected induced subgraphs of $G(V, E)$.

- Maximal Connected Induced Subgraph: $H(V_H, E_H)$:
  - Induced $H(V_H, E_H) = G[V_H]$
  - $H(V_H, E_H)$ is Connected
  - Maximal: for any node $x \in V \setminus V_H$: $G[V_H \cup \{x\}]$ is **not** connected

Some Special Graphs

- Complete graphs
  - $G(V, E)$ with $E = V \times V$
- Trees
  - unrooted
- Bipartite graphs

Complete graphs K_n



K_3 K_4 K_6

$|V| = n$, $|E| = n(n-1)/2$

Degree = n-1

Trees

- Connected graph $G(V, E)$ with $|E| = |V| - 1$.

- **Claim**: any connected graph on nodes V must have at least $|V| - 1$ edges.
- Proof:
  - Any connected graph $G(V, E)$ can be created by adding edges to an initially empty graph on nodes V (no edge).
  - Inserting an edge to a graph reduces the number of connected components by at most one.
  - Connected components: initially $n \rightarrow$ eventually 1

Trees

Property: a graph is a tree <u>if and only if</u>

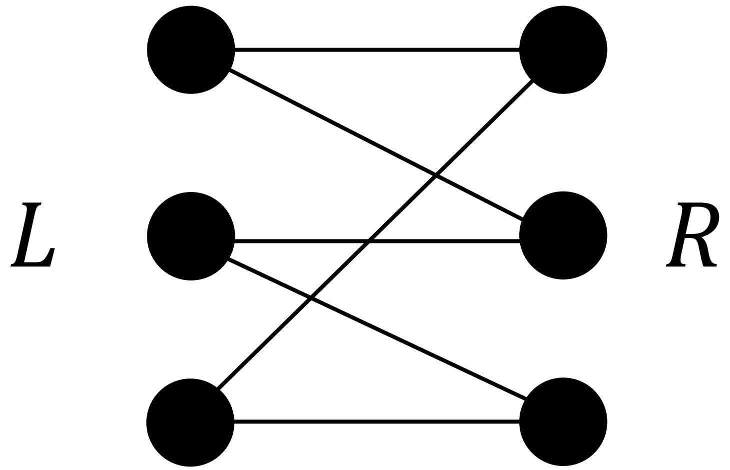
it is connected and contains no cycle.

Proof:

- **tree -> no cycle**: if there exists a cycle, then after removing any edge in the cycle, the graph remains connected. Hence (before the deletion) the graph contains at least |V| edges, which is not a tree.
- **no cycle -> tree**: removing any edge (u,v) disconnects u and v (otherwise there is a cycle). Hence there are at most |V|-1 edges in the graph. Since the graph is connected, it is a tree.

Bipartite Graphs

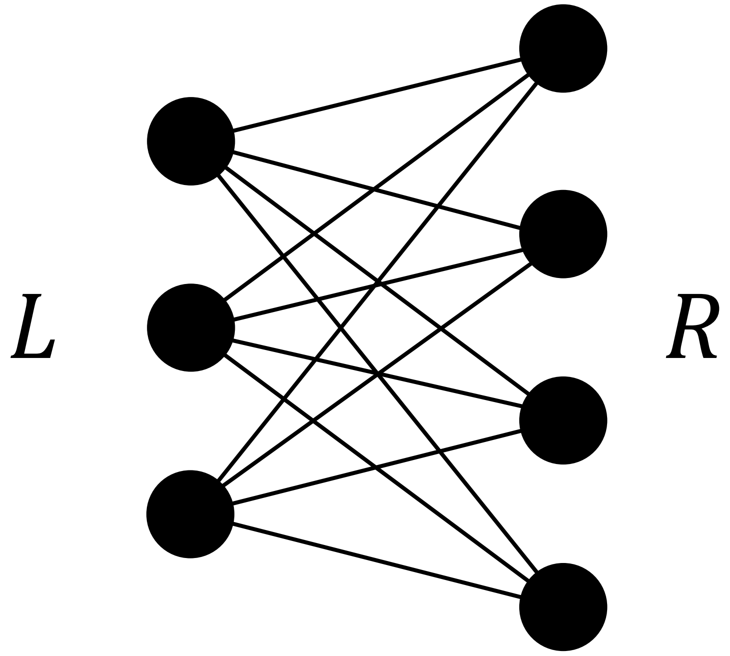
- Graph $G(V, E)$ in which the nodes V can be partitioned into two sets L and R such that every edge $e \in E$ connects a node from L and a node from R.
- Notation: $G(L \cup R, E)$, where $E \subseteq L \times R$.



Complete Bipartite Graphs

- Bipartite graph $G(L \cup R, E)$ with $E = L \times R$.

$K_{3,4}$



Bipartite Graphs

Property: $G(V, E)$ is bipartite <u>if and only if</u> it contains no odd cycle.

- bipartite → no odd cycle:
  - In any cycle $(u_0, u_1, \ldots, u_l)$, if $u_i \in L$ then $u_{i+1} \in R$
  - $u_0 = u_l \rightarrow l$ (length of cycle) is even
- no odd cycle → bipartite:
  - Color an arbitrary node black.
  - Repeat: color neighbors of black nodes white; color neighbors of white nodes black.
  - no odd cycle → no conflict
  - L: black nodes; R: white nodes

Graphs Traversal

Graph Traversal

- A traversal is a systematic procedure for exploring a graph by examining all its nodes and edges.
- Depth-first-search (DFS) from a node $u \in V$
  - recursively visit an unvisited neighbor
  - easy to implement
- Breadth-first-search (BFS) from a node $u \in V$
  - visit all neighbors of u, then neighbors of neighbors u, etc
  - closest nodes are visited first

Depth-first-search (DFS)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

DFS(u)

- label u as "visited" and print u
- **for** (each neighbor v of u):
  - **if** (v is not visited):
    - label edge (u, v) as a discovery-edge
    - DFS(v)
  - **elseif** (edge (u, v) has not been labelled)
    - label edge (u, v) as a back-edge

Depth-first-search (DFS)

- Produce a traversal order of the nodes
- Classify all edges into discovery-edges and back-edges
- **Complexity**: $O(n+m)$
  - Excluding recursive calls, DFS(u) runs in $O(d(u)+1)$ time.
  - For each node v, DFS(v) will be called exactly once
  - Total time = $\sum_{v \in V} O(d(v)+1) = O(n+m)$

Breadth-first-search (BFS)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

BFS(u)

- initialize an empty queue Q
- Q.enqueue(u) and label u as visited
- **while** (Q is not empty)
    - $v \leftarrow Q$.dequeue() and print v
    - **for** (each neighbor w of v)
        - **if** (w is not visited)
            - Q.enqueue(w) and label w as visited
            - label (v, w) as a discovery-edge

Breadth-first-search (BFS)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Produce a traversal order of the nodes
- Classify all edges into discovery-edges and cross-edges
- **Complexity**: $O(n+m)$
  - There is no recursive calls
  - Every node is enqueued and dequeued at most once
  - After node v is dequeued, $d(v)$ for-loops are executed
  - Total time = $\sum_{v \in V} O\big(1+d(v)\big) = O(n+m)$

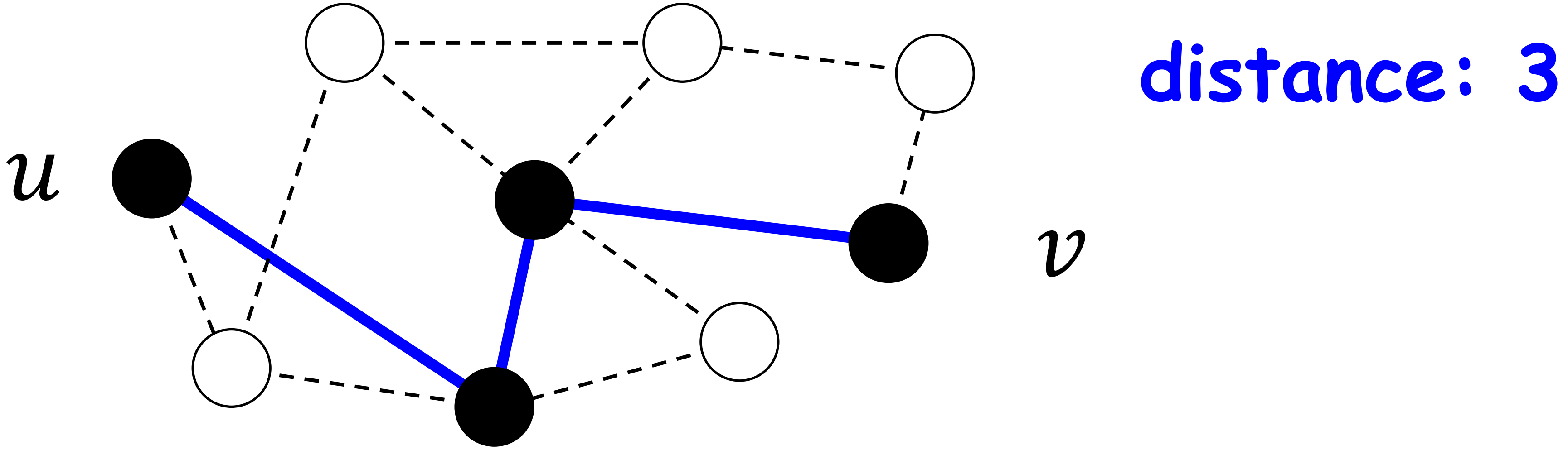
Graph Traversal

- A traversal is a systematic procedure for exploring a graph by examining all its nodes and edges.
- Depth-first-search (DFS) from a node $u \in V$
- Breadth-first-search (BFS) from a node $u \in V$
- Both runs in $O(n + m)$ time
- Outputs a tree containing all nodes connected to u
- Can be used to test whether G is connected

Shortest Path

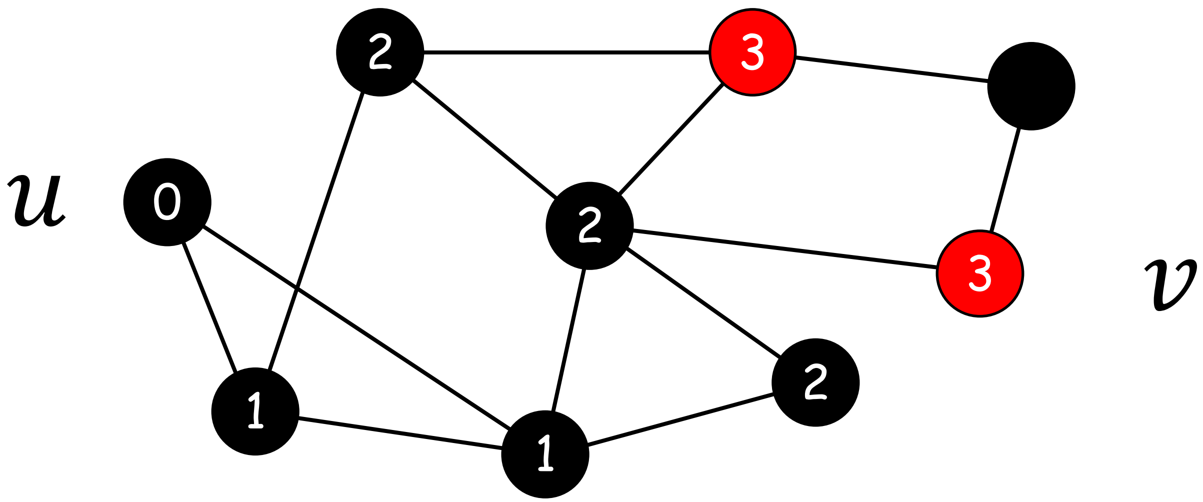
Shortest Path

- **Shortest path** between two nodes u and v : the path connecting u and v with the smallest length.
- **Distance** between u and v : length of the shortest path.



Compute Distances

- Let $d(u, v)$ be the distance between u and v.
- How to compute $d(u, v)$?



Unweighted Distance by BFS

procedure UnweightedDistance(G, u)

- let Q be a queue
- Q.enqueue(u), set $d(u,u) \leftarrow 0$ and label u as "visited"
- **while** Q is not empty **do**
  - $x \leftarrow$ Q.dequeue()
  - **for** each edge $(x,y) \in E$ **do**
    - **if** y is not "visited" **then**
      - Q.enqueue(y), set $d(u,y) \leftarrow d(u,x) + 1$
      - label y as "visited"

Edge-weighted Graphs

- Graph $G(V, E)$ on nodes V and edges $E \subseteq V \times V$.
- Every edge $e = (u, v) \in E$ has a weight $w_e = w_{uv} > 0$.
  - Unweighted graphs: $w_e = 1$ for all edge $e \in E$

- Path $(u_0, u_1, \dots, u_l)$ has **length** $w_{u_0 u_1} + w_{u_1 u_2} + \cdots + w_{u_{l-1} u_l}$
- **Distance** between u and v: length of the shortest path connecting u and v.

Single Source Shortest Path

- Given edge-weighted undirected graph $G(V, E)$.

- Let $d(s, u)$ be the (weighted) distance between s and u.

- **Problem**: for a fixed node $s \in V$ (the source node), compute the distance $d(s, u)$ for all $u \in V$.
  - Single Source Shortest Path (SSSP) problem

Dijkstra's Algorithm

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- S = { all nodes u for which $d(s,u)$ is **computed** }.
- Initially $S = \{s\}$, $d(s,s) = 0$.

- For each other node u, compute an estimate $d'(s,u)$ of the distance from s to u.
  - $d'(s,u)$ is an upper bound of the real distance
- Initially, let $d'(s,u) = w_{su}$ if edge $(s,u) \in E$ exists;
- otherwise let $d'(s,u) = \infty$.

Dijkstra's Algorithm

UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- In each step: how can we be sure that the estimate is correct, i.e., $d'(s,u)$ is the actual distance $d(s,u)$?
- Answer: the node u with the **smallest** $d'(s,u)$

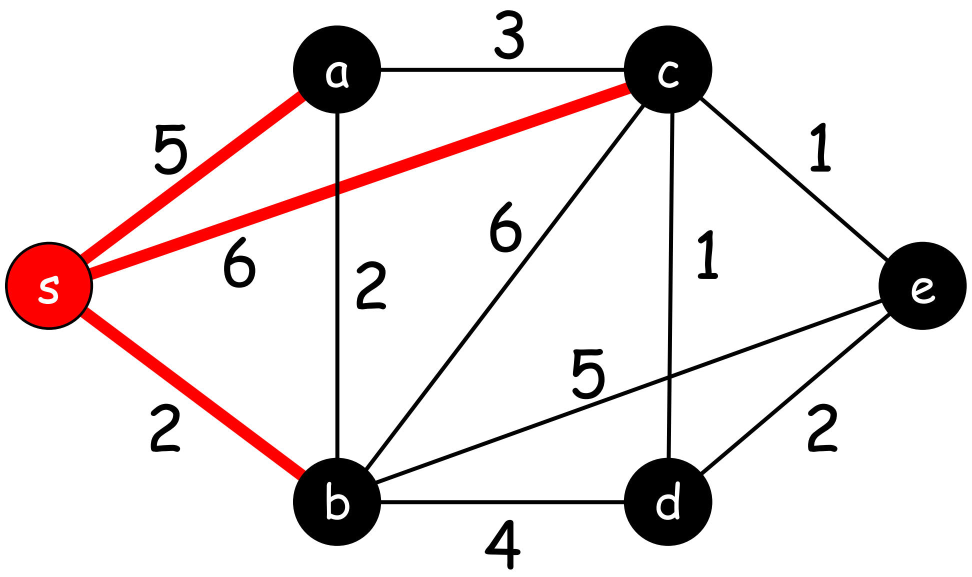
- In this case, we know $d(s,u) = d'(s,u)$.
- Insert u into S & record its distance:
  - $S \leftarrow S \cup \{u\}$.
  - $d(s,u) \leftarrow d'(s,u)$.

Update the Estimate

Initialize $S \leftarrow \{s\}$, $d(s, s) \leftarrow 0$.
$d'(s, u) \leftarrow w_{su}$ if edge $(s, u) \in E$ exists;
otherwise let $d'(s, u) \leftarrow \infty$.

1. **Repeat**
2. let $u \in V \setminus S$ have the **smallest** $d'(s, u)$;
3. $d(s, u) \leftarrow d'(s, u)$; insert u into S: $S \leftarrow S \cup \{u\}$.
4. For each edge $e = (u, v) \in E$ and $v \in V \setminus S$,
5. if $d'(s, v) > d(s, u) + w_{uv}$ then update $d'(s, v) \leftarrow d(s, u) + w_{uv}$
6. **until** $S = V$.

Example



$d(s, s) = 0$

$d'(s, a) = 5$

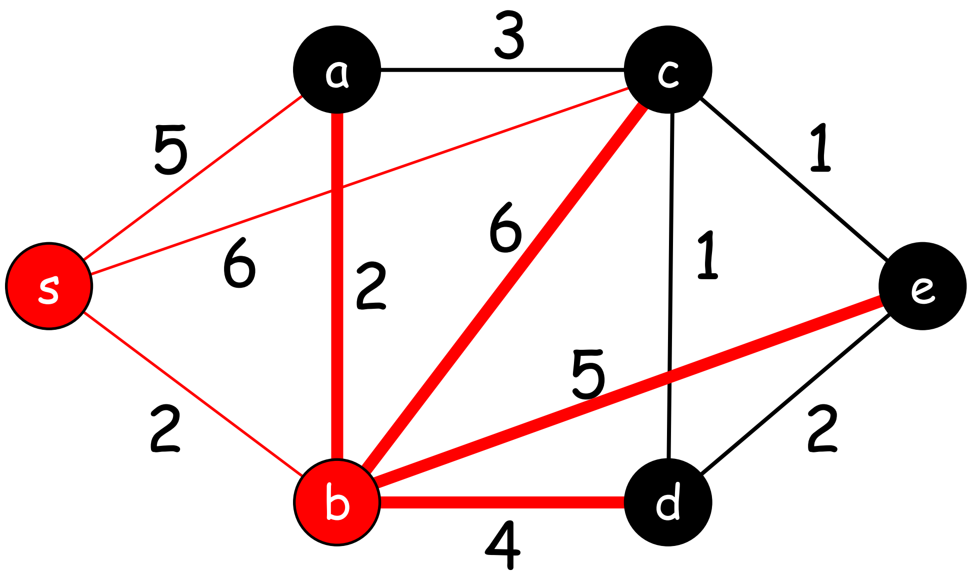
$d'(s, b) = 2$

$d'(s, c) = 6$

$d'(s, d) = \infty$

$d'(s, e) = \infty$

Example



$d(s,s) = 0$

$d'(s,a) = 4$

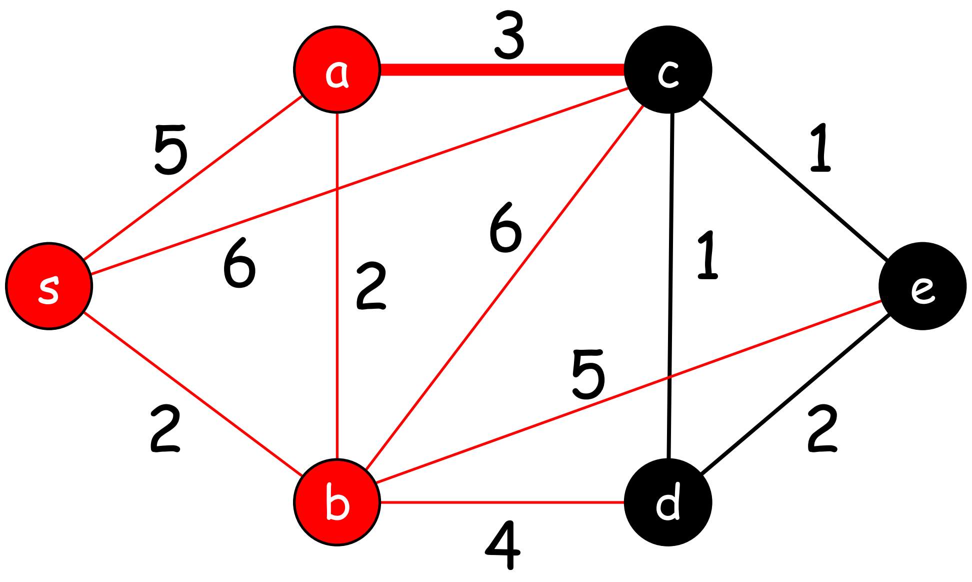
$d(s,b) = 2$

$d'(s,c) = 6$

$d'(s,d) = 6$

$d'(s,e) = 7$

Example



$d(s,s) = 0$

$d(s,a) = 4$

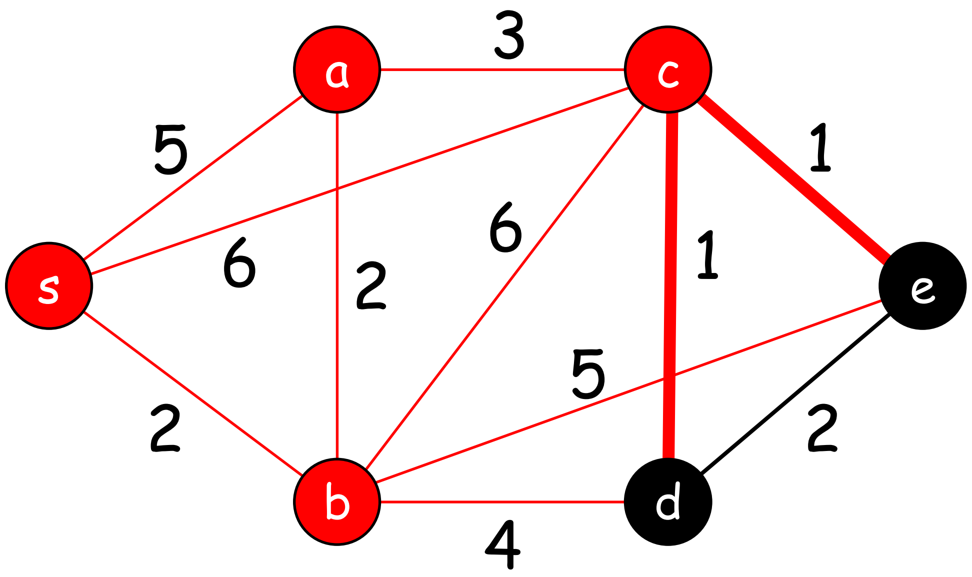
$d(s,b) = 2$

$d'(s,c) = 6$

$d'(s,d) = 6$

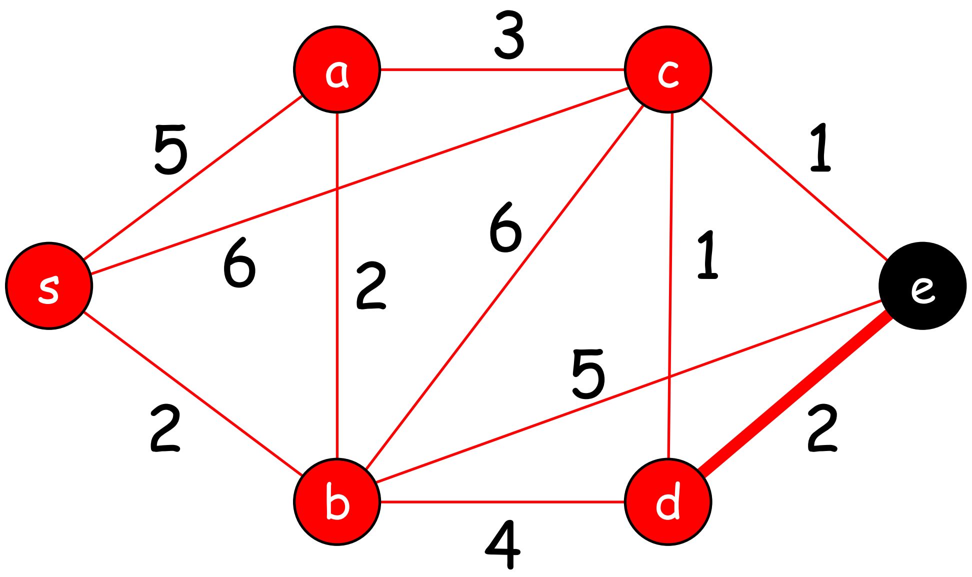
$d'(s,e) = 7$

Example



$d(s,s) = 0$
$d(s,a) = 4$
$d(s,b) = 2$
$d(s,c) = 6$
$d'(s,d) = 6$
$d'(s,e) = 7$

Example



$d(s,s) = 0$
$d(s,a) = 4$
$d(s,b) = 2$
$d(s,c) = 6$
$d(s,d) = 6$
$d'(s,e) = 7$

Implementation

- **Trivial implementation:** $O(n^2)$ time
  - $n-1$ rounds in total
  - each round takes $O(n)$ time (find-min and update estimates)
- Better implementation: Use an efficient data structure to store the estimates for all nodes in $V \setminus S$.
  - Query the minimum estimate.
  - Deletion of the minimum estimate.
  - Update of estimate $d'(s,u)$.

Implementation

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Binary Search Tree (AVL-Tree):**
  - Query the minimum estimate O(log n) (need to implement)
  - Deletion of the minimum estimate O(log n)
  - Update of estimate $d'(s, u)$ O(log n)
- **Priority Queue (Min-Heap):**
  - Query the minimum estimate O(1)
  - Deletion of the minimum estimate O(log n)
  - Update of estimate $d'(s, u)$ O(log n) (need to implement)
    - In practice: insert a new value

Running Time

- At most n Delete min operations.

- Each edge causes at most one Update operation.
- There are at most m Update key operations.

- Time complexity: $O((n + m) \log n)$.

Dynamic Programming Solution

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Problem: Given edge-weighted graph $G(V, E)$ and source node s, compute $d(s, u)$ for all $u \in V$.

Dynamic Programming Solution:

- Definition: $d(s, u, k)$ = smallest length of all paths between s and u using at most k edges.
- Observation: $d(s, u) = d(s, u, n - 1)$.
- Recursion: $d(s, u, k) = \min\{d(s, v, k - 1) + w_{uv}: v \in N(u)\}$
- Base case: $d(s, u, 1) = w_{su}$ if $u \in N(s)$; $d(s, u, 1) = \infty$ otherwise

Minimum Spanning Tree

Minimum Spanning Tree

- Given an edge-weighted undirected connected graph $G(V, E)$:
- A spanning tree of $G(V, E)$ is a subgraph of G that is a tree and connects all nodes in V.

Minimum Spanning Tree (MST) Problem:

- Compute a spanning tree $T(V, E_T)$ on V such that
  - $E_T \subseteq E$ and $\sum_{e \in E_T} w_e$ is minimized.
- For unweighted graphs: any spanning tree is an MST.

Applications

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- **Network design**: telephone, electrical, hydraulic, TV cable, computer, road networks.
- Approximation algorithms for NP-hard problems
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein

Algorithms

- Two Greedy algorithms:
  - Simple to understand and implement
  - Non-trivial to prove its correctness
  - Require special data structures to allow efficient implementation

- **Kruskal's Algorithm.**
- **Prim's Algorithm.**

Kruskal's Algorithm

- Start with an empty graph $T(V, F)$ with $F = \emptyset$.
- Repeatedly add the next **lightest edge e** that **does not create a cycle**.
  - Consider edges one-by-one in ascending order of edge weight.
- Terminate when all nodes are connected in T.

Implementation

- Start with an empty graph $T(V,F)$ with $F = \emptyset$.
- Sort the edges in ascending order of weights
- Repeatedly add the next lightest edge e that does not create a cycle.

- How to determine whether the new edge $e = (u,v)$ creates a cycle?
- **Trivial Algorithm:** Run DFS/BFS on T starting from u, and check whether v can be visited.
  - Running Time: $O(|V| + |F|) = O(n)$

Implementation

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Start with an empty graph $T(V,F)$ with $F = \emptyset$.
- Sort the edges in ascending order of weights
- Repeatedly add the next lightest edge e that does not create a cycle.

- How to determine whether the new edge $e = (u,v)$ creates a cycle?
- **Union-Find** data structure.
  - Two nodes are connected in T if and only if they are in the same set of the Union-Find data structure

Union-Find Data Structure

- Maintains a collection of sets of elements and supports the following operations:

- Make-Set(x): Create a set containing only element x;
- Find-Set(x): Return the pointer to the set containing x;
- Union(p1, p2): Given pointers p1 and p2 to two different sets, merge the two sets.

Union-Find Data Structure

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Maintains a collection of sets of elements and supports the following operations:

- Make-Set(x): O(1) time;
- Find-Set(x): O(log n) time;
- Union(p1, p2): O(1) time.

- **Question**: how to implement such a data structure?
  - Hint: organize elements in one set as a tree

Pseudocode

procedure Kruskal(G)

Sort edges in ascending order of weights, and initialize $F \leftarrow \emptyset$
For each node $u \in V$ **do**
    Make-Set(u)
For each edge $e = (u, v) \in E$ in ascending order of w_e **do**
    If (Find-Set(u) $\neq$ Find-Set(v)) **then**
        $F \leftarrow F \cup \{e\}$.
        Union(Find-Set(u), Find-Set(v)).
Return F

Complexity

- Sorting all edges: O(m log m) = O(m log n) time
- Make-Set for each node: O(n) total time
- Check cycle for an edge: O(log n) time
  - m edges -> O(m log n) total time
- Insert an edge to F and Union: O(1) time
  - n-1 edges in F -> O(n) total time

- **Overall: O(m log n) time.**

Prim's Algorithm

- Start with a given node s
- Greedily grow a tree T from s outward.
- At each step, add the **lightest edge e** to T that connects to a node **outside T**.

Prim's Algorithm

- Start with a given node s. Let $V_T = \{s\}$ and $F = \emptyset$.
- Greedily grow a tree $T(V_T, F)$ from s outward.

- Boundary Edges: $\delta(V_T) = \{(u, v) \in E : u \in V_T, v \notin V_T\}$
  - Find edge $e \in \delta(V_T)$ with smallest edge weight w_e
  - Update the solution: Add e to F, and v to V_T
  - Update the data structure $\delta(V_T)$:
    - Remove $(x, v) \in \delta(V_T)$ for all $x \in V_T$
    - Include (v, y) to $\delta(V_T)$ for all $y \notin V_T$

Implementation

- **Binary Search Tree (AVL-Tree):**
  - Query the lightest edge O(log n) (need to implement)
  - Insertion of an edge O(log n)
  - Removal of an edge O(log n)
- **Priority Queue (Min-Heap):**
  - Query the lightest edge O(1)
  - Insertion of an edge O(log n)
  - Removal of an edge O(log n) (need to implement)

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

Pseudocode

procedure Prim($G(V, E)$, s)

Initialize $F \leftarrow \emptyset$ and Visited$[u] \leftarrow \text{False}$ for all $u \in V$

Create a Heap containing all edges incident to s and set Visited$[s] \leftarrow \text{True}$

While (Heap is not empty) **do**

    let (u, v) be the lightest edge in the Heap, with Visited$[v] = \text{False}$

    $F \leftarrow F \cup \{(u, v)\}$ and set Visited$[v] \leftarrow \text{True}$

    For each $x \in N(v)$ **do**

        If (Visited$[x] = \text{True}$) **then** Remove (x, v) from the Heap

        If (Visited$[x] = \text{False}$) **then** Insert (v, x) into the Heap

Return F

Complexity

澳門大學
UNIVERSIDADE DE MACAU
UNIVERSITY OF MACAU

- Initialization and update of Visited[*]: O(n) time
- Insert/remove an edge in the data structure: O(log n) time
  - Each edge is inserted and removed at most once
  - m edges -> O(m log n) total time
- Get min and insert the edge to the tree: O(log n) time
  - n-1 edges in F -> O(n log n) total time

- **Overall: O(m log n) time.**