# Data Structure

Jin Zengrui

# Growth of Functions

input size : n

each basic operation takes constant time

+ - * / % = [i] func(x) return val comparison

An example :

- How do we analyze an algorithm?
  - Count the number of basic operations

**sum(A,n)**

```
1  tempsum = 0
2  for i = 0 to n-1
3     tempsum += A[i]
4  return tempsum
```

Number of basic operations

1
2*n +2
2*n
1

Total: 4*n +4

Why 2*n +2? The actual effect of the loop is:
for (i=0; i<n ; i++)
We assign 0 to i once, increment i for n times, compare i<n n+1 times

Why 2*n? Accessing A[i] n times, addition to tempsum n times

It is annoying to count the detailed number of basic operations, we will show how to simplify the counting process later (Page 27).

e.g. linear search

```
for i=0 to n-1                    n+1
    if array[i] == target :       n
        return i                  1
return -1                         1
```

total : 4n+4

e.g binary search analysis

```
left = 0    right = n-1                          2
while left <= right:                             log2n + 1
    mid = (left + right) // 2                    3·log2n + 3
    if target < array[mid]:                      2·log2n + 2
        right = mid + 1                          2·log2n + 2
    else if: target > array[mid]:                2 log2n + 2
        left = mid + 1                           2 log2n + 2
    else:                                        1
        return mid                               1
return -1                                        1
```

total $12 \cdot (\log_2(n) + 17$

e.g. max sub array sum

```
maxSum = array[0]                                2
for i=0 to n-1                                   2n+2
    subSumFromI = 0                              n
    for j=i to n-1                               (2n+2)+(2(n-1)+2) + ... +(2·1+2)
        subSumFromI += array[j]                  2(n + n-1 + n-2 + ... 1)
        if subSumFromI > maxSum                  n + n-1 + n-2 + ... +1
            maxSum = subSumFromI                 n + n-1 + n-2 + ... +1
return maxSum                                    1
```

total: $3n^2 + 8n + 5$

## Big-Oh Notation

def $g(n) = O(f(n))$ iff exist a constant $c$ and $n_0$ s.t. $g(n) \le c \cdot f(n)$ for all $n \ge n_0$

即 $g(n)$ 以 $f(n)$ 为上界

describe running time of the algorithms using Big-Oh / Omega / Theta notations

linear search: $g(n) = 4n+4 = O(n)$ time complexity of the linear search algorithm is $O(n)$

any linear function: $g(n) = a \cdot n + b = O(n)$

log functions with different base $a>1$ and $b>1$:

$g(n) = \log_a n = O(\log_b n) = O(\log n)$

↑ can assume $\log n = \log_{10} n$ or $\log n = \log_2 n$

Practice:

Binary Search: prove that $g(n) = 12\log_2 n + 17 = O(\log n)$

can find $c=29$, $n_0=2$ such that

$g(n) = 12\log_2 n + 17 \le 12\log_2 n + 17\log_2 n = 29\log_2 n$ for $n \ge 2$

thus $g(n) = O(\log_2 n) = O(\log n)$

$g(n) = 1{,}000{,}000 = O(1)$

can find $c = 1{,}000{,}000$, $n_0 = 1$ s.t.

$g(n) = 1{,}000{,}000 \le 1{,}000{,}000 \cdot 1$ for $n \ge 1$

thus $g(n) = O(1)$

Disprove that $g(n) = 2n^2 = O(n)$

find a constant $c, n_0$, s.t.

$2n^2 \le cn$ for all $n \ge n_0 \iff n \le \frac{c}{2}$ for all $n \ge n_0$

b.c. $n$ can reach infinite, thus doesn't exist such constant $c$

- **Transitivity: Prove that if $g(n) = O(f_1(n)), f_1(n) = O(f_2(n))$, then $g(n) = O(f_2(n))$**

Proof: According to the definition, we know that:
There exist positive constants $c_1$ and $n_1$ for $g(n)$ such that

$$g(n) \leq c_1 \cdot f_1(n) \text{ for } n \geq n_1$$

and $c_2, n_2$ for $f_1(n)$ such that:

$$f_1(n) \leq c_2 \cdot f_2(n) \text{ for } n \geq n_2$$

Then, we know for $n \geq \max(n_1, n_2)$:

$$g(n) \leq c_1 f_1(n) \leq c_1 \cdot c_2 \cdot f_2(n).$$

Therefore $g(n) = O(f_2(n))$ according to the definition, which finishes the proof.

Big-Oh Rule (i) Given $a > b \geq 0$, $n^a$ grows faster than $n^b$

$$\lim_{n\to\infty} \frac{n^b}{n^a} = 0$$

(ii) Product property

$g_1(n) = O(f_1(n))$ $g_2(n) = O(f_2(n))$

Then, $g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$

(iii) Sum property only applies when the number of functions are constants

$g_1(n) = O(f_1(n))$, $g_2(n) = O(f_2(n))$

Then, $g_1(n) + g_2(n) = O(\max(f_1(n), f_2(n)))$

(iv) Log functions grow slower than power functions

$$\lim_{n\to\infty} \frac{(\log n)^a}{n^b} \to 0 \quad \text{for } a, b > 0$$

(v) Exponential functions grow faster than power functions

$$\lim_{n\to\infty} \frac{n^k}{a^n} \to 0 \quad \text{if } a > 1 \text{ for any } k$$

Growth: power → log → constant

# Complexity Analysis

## Big-Omega and Big-Theta

Big-Omega def $g(n) = \Omega(f(n))$ iff exist a constant $c$ and $n_0$ s.t. $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$.

即 $g(n)$ 以 $f(n)$ 为下界

Big-Omega Rules: Big-Oh rules (i)-(v) all apply to Big-Omega

Big-Theta: Big-Oh & Big-Theta are not tight

Big-Theta def. If $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$
$g(n) = \Theta(f(n))$

# Linked List

**Insertion**

insert( L, e) : add e at the end of the linked list

step 1 : identify the last node $u_{last}$ stored in the linked link

step 2 : allocate a new node $u_{new}$

step 3 : $u_{new}$.element = e

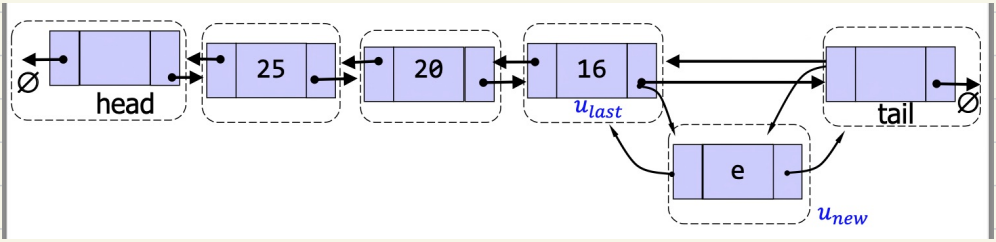
step 4 : $u_{new}$.prev = $u_{last}$

step 5 : $u_{new}$.next = tail

step 6 : $u_{last}$.next = $u_{new}$

step 7 : tail.prev = $u_{new}$

order can be changed




**Insertion in Place** : insertInPlace ( L, p, e) : insert element e after node p

step 1. allocate a new node $u_{new}$

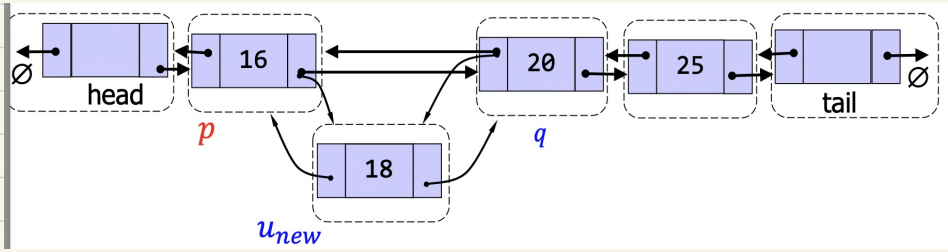
step 2 : set $u_{new}$.element = e

step 3 : set $u_{new}$.prev to the address of p

step 4 : Set $u_{new}$.next to the address of $q = p.next$

step 5 : set p.next to the address of $u_{new}$

step 6 : set $q$.prev to the address of $u_{new}$



Deletion : delete node p in the linked list.

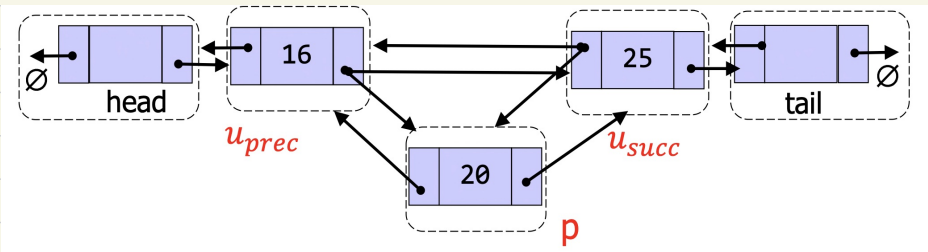
step 1 : identify the preceding node $u_{prec}$ of node p

step 2 : identify the succeeding node $u_{succ}$ of node p

step 3 : let $u_{prec}$.next points to $u_{succ}$

Step 4 : let $u_{succ}$.prev points to $u_{prec}$

Step 5 : free the memory of node p



# Stack

Last-In-First-Out

Balanced Symbol Checking

Evaluation of Expressions

Infix expression ⇒ Postfix Expressions

stack

e.g. (1) infix: 7 / (2+3) * 4

postfix: 7 2 3 + / 4 *


e.g. (2) infix : 2 * ( 3 + 2 * 4 )

postfix : 2 3 2 4 * + *

# Queue First-In-First-Out

Implementation: Linked List

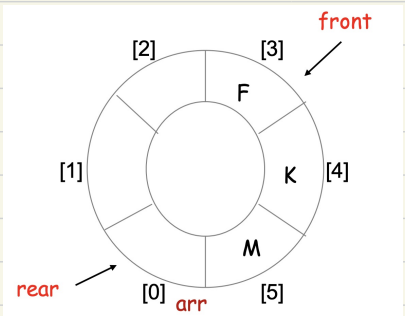
No capacity constraint

Array: circular structure

Better practical performance

Enquene: change the rear position to the next position

Dequene: change the front position to the next position




The next of capacity - 1 is 0

The next rear:

```
if rear != capacity - 1
    rear = rear + 1
else
    rear = 0
```

Or: (rear + 1) % capacity

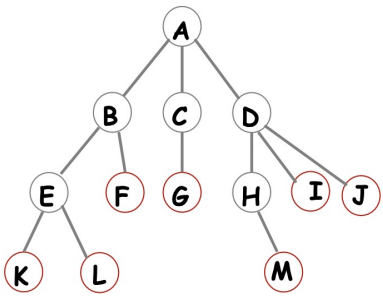
The next front.

(front + 1) % capacity

# Tree

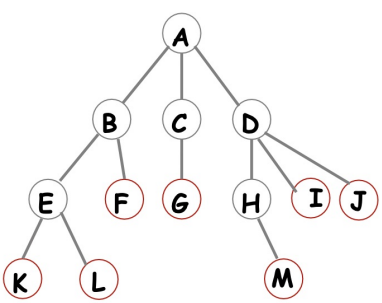
Terminology :

- Node: the elements plus the branches to each node.
- Edge: nodes are connected by edges.
- Branch of a node: the number of subtrees of a node
- Leaf nodes (or external node): nodes that have zero branch
- Internal nodes: nodes that don't belong to leaf nodes.



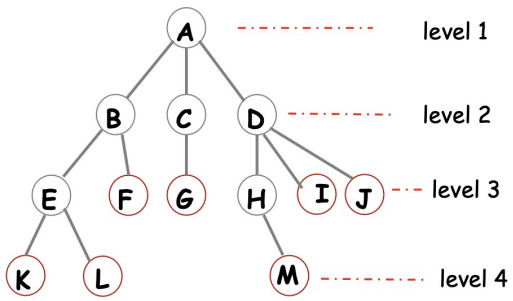

- A is the root node
- The branch of node B is 2
- F, G, I,J, K, L, M are leaf nodes
- A, B, C, D, E, H are internal nodes

- Children: the roots of the subtrees of a node X are the children of X
- Parent: X is the parent of its children
- Siblings: children of the same parent are said to be siblings.
- Ancestors of a node: parent, grand-parent, grand-grandparent,etc.
- Descendants of a node: child, grandchild, grand grand-child, etc.




- E and F are the children of B
- B is the parent of E and F
- C is the sibling of B
- A, B, E are the ancestors of K
- H, I, J, M are the descendants of node D

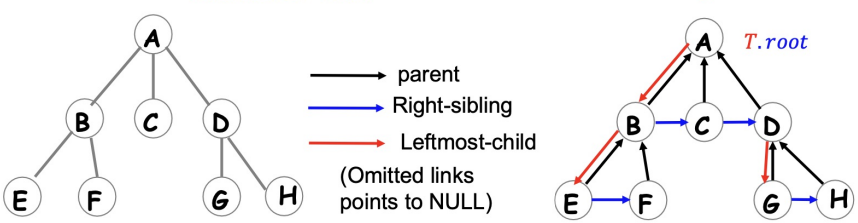
- The level of a node: defined by letting the root be at level one. If a node is at level l, then it children are at level l+1.
- Height (or depth): the maximum level of any node in the tree




The height (depth) of the tree is 4

# of nodes = # of edges + 1

- Pointer representation of a tree $T$
  - Maintain $root$ to store the address of the root node.
  - For each node, it further maintains the address of its parent, leftmost-child, and right-sibling.
    - For example, for node B
      - B.parent stores the address of node A.
      - B.right-sibling stores the address of node C.
      - B.leftmost-child stores the address of node E.



# Binary Tree

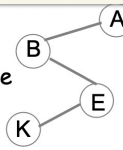
maximum number of nodes on level $i$ of a binary tree is $2^{i-1}$ for $i \geq 1$

maximum number of nodes in a binary tree of height $k$ is $2^k - 1$ for $k \geq 0$

height of a binary tree with $n$ nodes falls in the range between $\lceil \log_2(n+1) \rceil$ and $n$

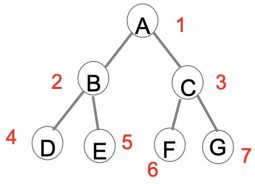
## Special Binary Tree

- Skewed binary tree
  - All the nodes have only either one child or no child.



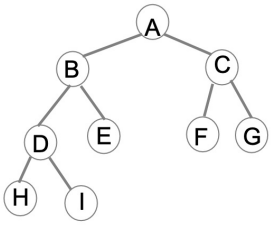

Skewed binary tree

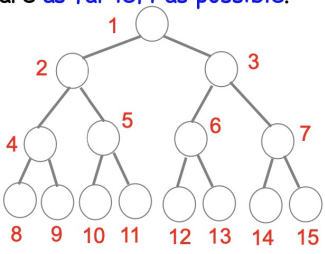
- Full binary tree
  - every node other than the leaves has two children.
  - If the height is k, the number of nodes is $2^k - 1$ (i.e., the maximum we can have with height k)
  - The nodes of a full binary tree can be numbered from 1 to $2^k - 1$ .




Full binary tree

- Complete binary tree
  - A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.




  - Its nodes correspond to the nodes numbered from 1 to $n$ in the full binary tree of depth $k$.
    - In this example, we have 9 nodes. The complete binary tree should corresponds to the nodes numbered 1-9 in the full binary tree

# Binary Tree Implementation

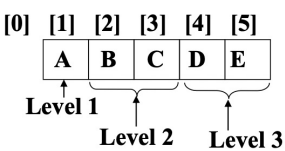
Pointer :

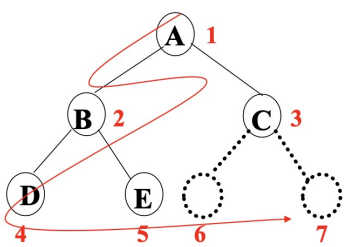
node :

parent : ptr
leftchild : ptr
rightchild : ptr
element : value

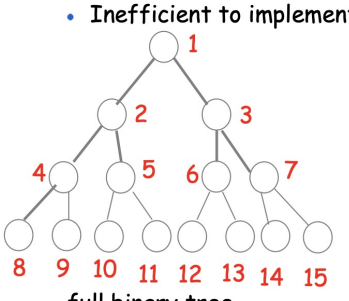
Array : will be used for heap

- An array representation
  - Given a complete binary tree with $n$ nodes. For any $i$-th node, $1 \leq i \leq n$,
    - parent($i$) is $\lfloor i/2 \rfloor$
    - leftChild($i$) is at $2i$ if $2i \leq n$. Otherwise, $i$ has no left child.
    - rightChild($i$) is at $2i+1$ if $2i+1 \leq n$. Otherwise, $i$ has no right child.

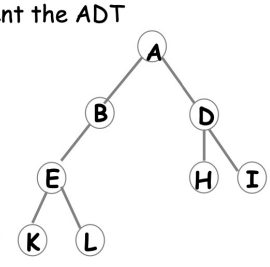


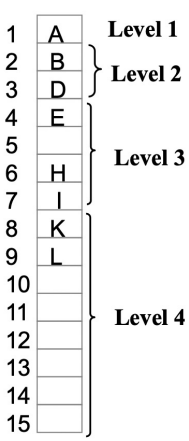




- An array representation
  - Generalize to all binary trees
  - Efficient for complete binary trees.
  - But inefficient for skewed binary trees.
  - Inefficient to implement the ADT



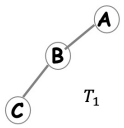

full binary tree



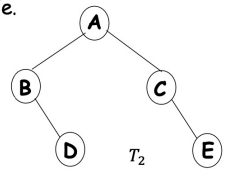



e.g. Practice

- What are the array representation of the following binary trees?
  - Show the content in the array.
  - Hint: first obtain the ID for each node.




$T_1$




$T_2$

$T_1$: | A | B | C | | | |

$T_2$: | A | B | C | | D | | E |

Inorder Traversal:

Inorder traverse its left-subtree
Then visit the root node
Inorder traverse its right-subtree

Postorder Traversal

Postorder traverse its left-subtree
Postorder traverse its right-subtree
Then visit the root node

Preorder Traversal

Visit the root node
Preorder traverse its left-subtree
Preorder traverse its right-subtree

- Calculate the height of a binary tree.

Algorithm: ***height(btree)***

```
1  if isEmpty(btree)
2    return 0
3  lheight ← height(Lchild(btree))
4  rheight ← height(Rchild(btree)
5  return 1+ max(lheight, rheight)
```

```
1  ADT BinTree
2   BinTree create(bt);
3   Boolean isEmpty(bt)
4   BinTree MakeBT(bt1,
5  item, bt2);
6   BinTree Lchild(bt);
7   BInTree Rchild(bt);
8   element Data(bt);
```

- Sum over all the elements in the binary tree

Algorithm: ***sum(btree)***

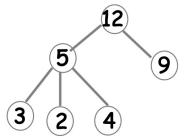
```
1  if isEmpty(btree)
2    return 0
3  lsum ← sum(Lchild(btree))
4  rsum ← sum(Rchild(btree)
5  return Data(btree) + lsum+ rsum
```

# Heap

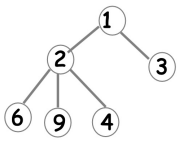
A special form of complete binary tree

e.g. max heap or min heap

- Max (resp. Min) heap is defined based on max (resp. min) tree and complete binary tree
- Max (resp. min) tree: A *max (resp. min) tree* is a tree in which the element, or call it the key value, in *each node* is no smaller (resp. no larger) than the key values in its children.




An example of a max tree

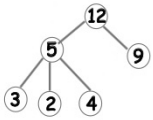



An example of a min tree

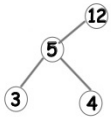
Max Heap a complete binary tree that is also a max tree

root contains the largest element

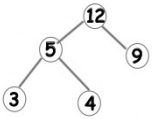
easy to find the maximum




A max tree, but not a binary tree




A max tree, but not a complete binary tree

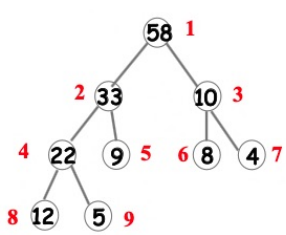



A max tree and also a complete binary tree

- A complete binary tree can be represented with arrays
  - For any $i$-th node, $1 \leq i \leq n$,
    - parent($i$): $\lfloor i/2 \rfloor$
    - leftChild($i$): $2i$ if $2i \leq n$. Otherwise, no left child.
    - rightChild($i$): $2i + 1$ if $2i + 1 \leq n$. Otherwise, no right child.
- A max *heap* can be represented as an array arr of elements such that
  - $arr[i] \geq arr[2i]$ if $2i \leq n$
  - $arr[i] \geq arr[2i + 1]$ if $2i + 1 \leq n$

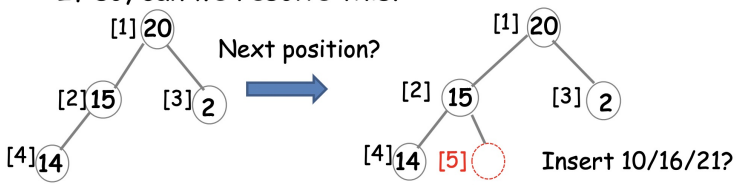
| [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|
| 58 | 23 | 10 | 22 | 9 | 8 | 4 | 12 | 5 |

33

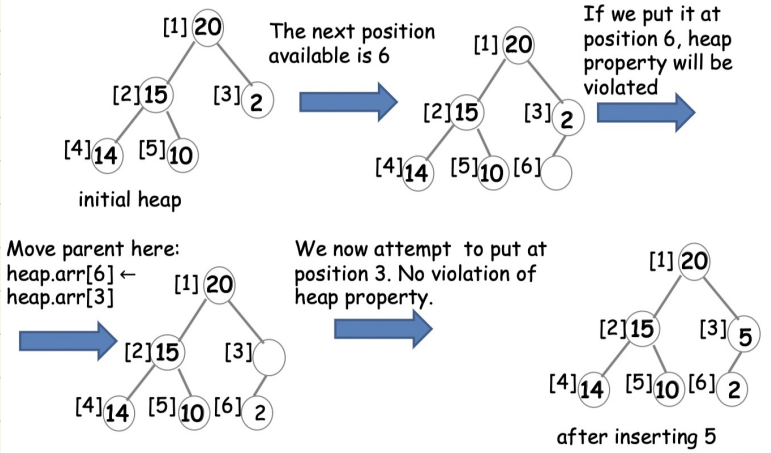


Heap Insertion
O(log n)

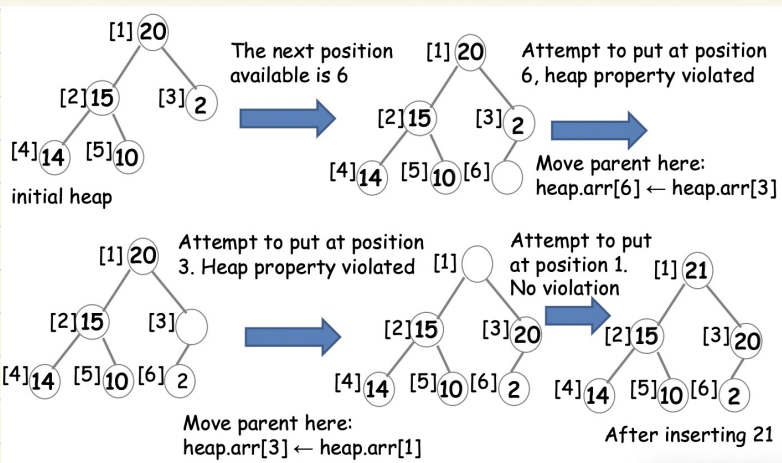
- **Questions:**
  - Where to insert?
    - After insertion, it must still be a complete binary tree.
    - If the current heap includes x nodes, then the new element should be inserted at the (x+1)-th position of the complete binary tree
  - After insertion, does it violate the heap property? If so, can we resolve this?




Insert 5




Insert 21



## Heap Insertion Implementation

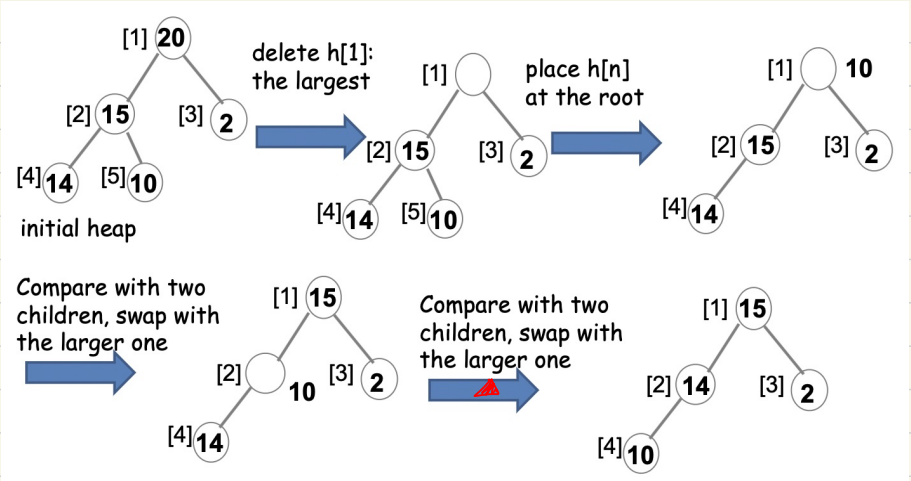
- Put it at the (size+1)-th position in the complete binary tree.
- But this may violate the heap property
  - Denote the current position as i. Its parent position is at i/2
  - If the current position is at the root, just insert item to root
  - Otherwise, compare the parent and the item to be inserted.
    - If the parent has smaller key, move parent to current position, change the current position to i/2, and repeat above process.
    - Otherwise, insert the item to the current position

Algorithm: *InsertHeap(heap, item)*

```
1  if isFull(heap)
2     error "Heap full"
3  heap.size = heap.size+1
4  i = heap.size
5  While i!= 1 and item.key > heap.arr[i/2].key
6      heap.arr[i] = heap.arr[i/2]
7      i = i/2
8  heap.arr[i] = item
```

## Heap Deletion O(logn)

- Return and delete the largest element,
  - Which one to delete?
    - According to the max-heap property, the largest element is at the root
  - Which one to fill at the root?
    - The tree must still be a complete binary tree: Use the last one to fill the root position
    - This obviously violates the heap property: how to resolve?



# Binary Search Tree

Heap, max tree, binary tree are inefficient for searching

In a sorted array:

Linear scan: O(n) cost

Binary search: O(log n) cost

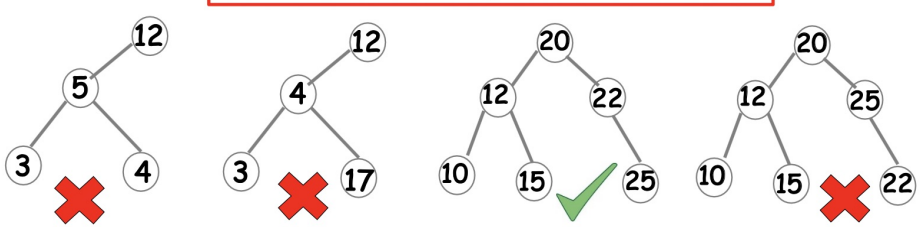
Main Idea: compare the searchnum with the middle element of the array, then we can reduce the search space by half

How to map to binary trees?

Binary Search Tree (BST) definition:

- A **binary search tree** is a binary tree that satisfies the following properties:
  - Every node has a **unique key**.
  - For each internal node x:
    - All nodes in a nonempty **left subtree** of x must have keys **smaller** than the key of x.
    - All nodes in a nonempty **right subtree** of x must have keys **larger** than the key of x.

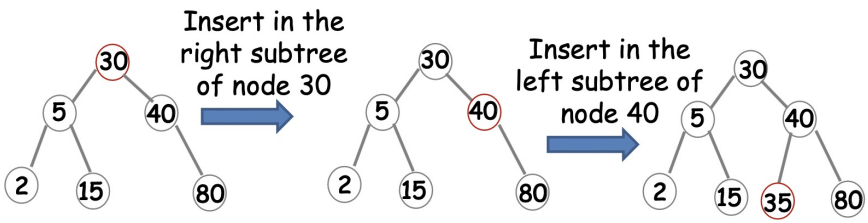
Are these trees binary search trees? Why?




Search cost: Denote h as the height of the BST. time complexity O(h)

Insertion of BST:

- No duplicate in the binary search tree
  - Ignore the insertion if it contains a key that exists in the binary search tree
- Where to insert?
  - After insertion, we should not violate the BST property



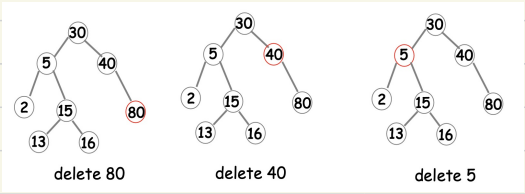

Insert 35

- Base case: the root is empty, insert the data at the root.
- Otherwise, follow the search path until we find a node $x$ such that
  - (i) $x.key = data.key$, simply return since BST does not allow duplicate keys
  - (ii) node $x$ has no right child and $x.key < data.key$
    - Insert as the right child of $x$
  - (iii) the node has no left child and $x.key > data.key$
    - Insert as the left child of $x$
- Pseudocode: left as self exercise
- Time complexity?
  - $O(h)$, where h is the height of the BST

Deletion of BST : Case 1 : delete a leaf node

Case 2 : delete a node with only one child

Case 3 : delete a node with two children

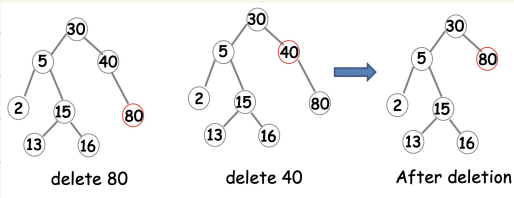



## Case 1 : delete leaf node

If the leaf node is a right (resp. left) child, just set the right (resp. left) child of its parent to NULL.

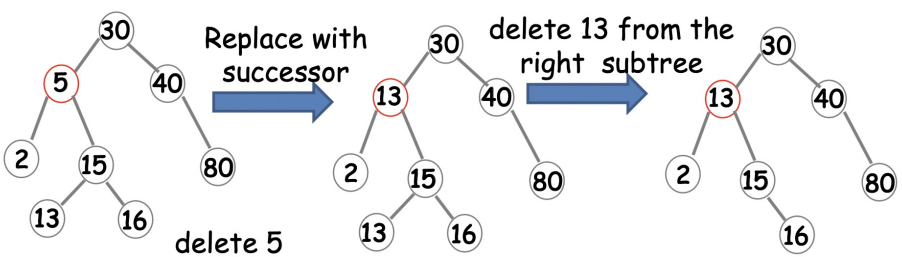
## Case 2 : delete a node with a single child

Put the single child at the place of the deleted.




## Case 3 : delete a node with two children

- We should replace a node here. But which node to put here?
- The successor of the node to be deleted
  - The successor of node 5 is node 13
- Replace the current node with its successor. Then delete the successor from the right sub-tree.
  - The deletion of successor falls into case 1 or case 2.
  - Think why?




time complexity : search, maximum / minimum / successor / predecessor, insertion, deletion

O(h), where h is the height of the tree.

worst case : O(n), where n is the # of nodes

# Balanced Binary Search Tree

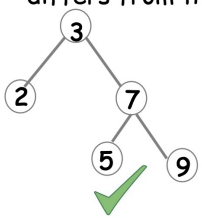
Goal: Keeping the height of the binary search tree low

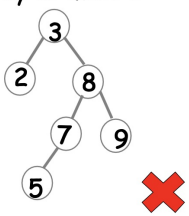
Recap: A complete binary tree has height of $O(\log n)$

Solution: relax the condition a little bit.

A binary search tree is balanced if:

- For every node in the tree, the height of the left subtree differs from the height of the right subtree by at most 1



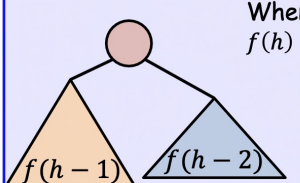




## Height of Balanced Binary Search Tree

**Theorem 1:** Given a balanced binary search tree $T$ of $n$ nodes, the height, or equivalently the depth, of $T$ is $O(\log n)$.

**Proof:** Let $f(h)$ be the minimum number of nodes of a balanced binary search tree of height $h$. Then, it is easy to verify that $f(1) = 1, f(2) = 2$.

For any $h \geq 3$, we have that $f(h) = f(h-1) + f(h-2) + 1$




When $h$ is even number:

$$f(h) > f(h-1) + f(h-2)$$
$$> 2f(h-2)$$
$$> 4f(h-4)$$
$$\cdots$$
$$> 2^{\frac{h}{2}-1} \cdot f(2) = 2^{\frac{h}{2}}$$

When $h$ is odd number:

$$f(h) > f(h-1)$$
$$> 2^{\frac{h-1}{2}}$$

Therefore, given a balanced BST of $n$ nodes of height $h$, we have:

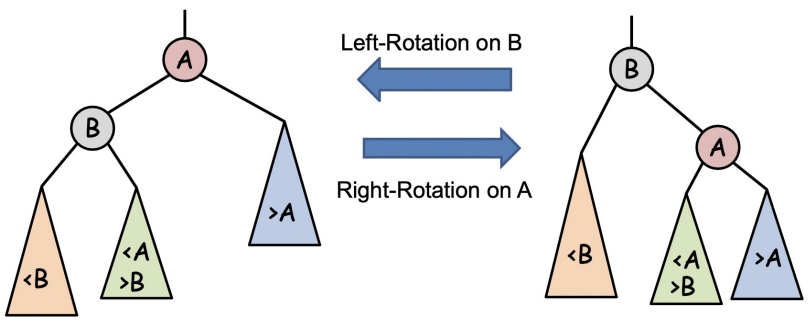
$$n > 2^{\frac{h-1}{2}} \Rightarrow h < 2\log_2 n + 1 \Rightarrow h = O(\log n)$$

## How to Keep a BST Balanced ?

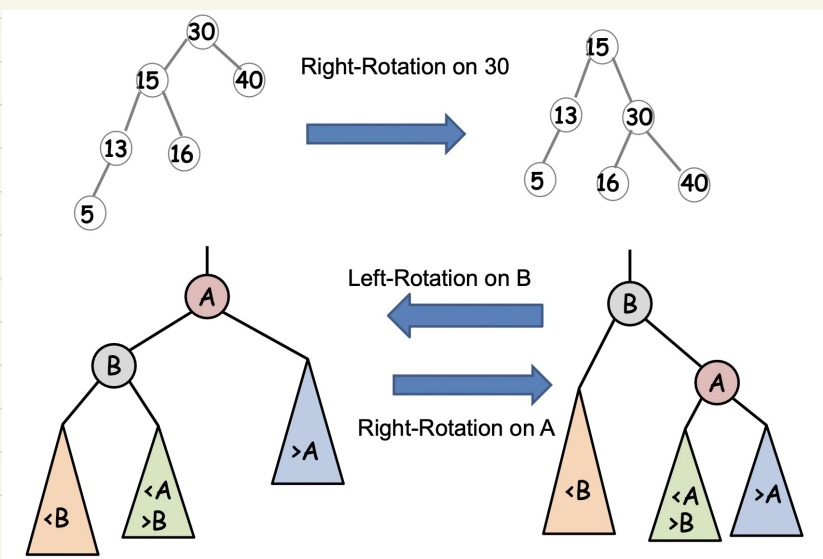
Dynamic Balancing: After updating the BST, we rebalance the BST using rotation


Rotation :

- Rotation allows us to change the structure without violating the binary search tree property




Rotation Example :



# Sorting Algothrim

## Selection sort

- Step 1: Scan all the n elements in the array to find the position $i_{max}$ of the largest element $maxnum$

$i_{max} = 4$, $maxnum = 9$

| 4 | 2 | 3 | 6 | 9 | 5 |
|---|---|---|---|---|---|

- Step 2: swap the position of the last one and $maxnum$

| 4 | 2 | 3 | 6 | 5 | 9 |
|---|---|---|---|---|---|

- Step 3: We have a smaller problem: sorting the first n-1 elements

| 4 | 2 | 3 | 6 | 5 | sorted |
|---|---|---|---|---|---|

time complexity:

to find the maximum from n elements: a cost of n

total cost for selection cost:

$$n + (n-1) + (n-2) + \dots + 1 = \frac{n(n+1)}{2}$$

time complexity: $O(n^2)$

## Insertion sort:

- If we have a sorted array, and a new element comes, we insert the new element in the correct place
  - If the first i elements in an array are sorted, we only need to insert the (i+1)-element to the right place to make the first (i+1) elements in the array are sorted

```
1  if arr.length <= 1 //Array is sorted if it has length <=1
2      return arr
3  for i from 1 to arr.length-1 //insert arr[i] to the right position
4       //move left if it is smaller than left element
5      for j from i downto 1
6          if arr[j] > arr[j-1]
7              break
8          temp ← arr[j]
9          arr[j] ← arr[j-1]
10         arr[j-1] ← temp
```

time complexity : $O(n^2)$ (worst case)

Let arr[i] to be left out of order (LOO) iff

$arr[i] < \max\{arr[0], arr[1], \ldots, arr[i-1]\}$

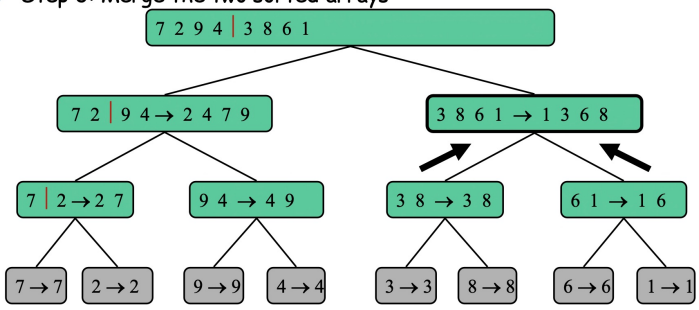
If we have K LOO array entries, the time complexity is $O((k+1)n)$

Merge sort: High Level Idea

- Given n elements, we divide them into roughly two equal part
  - The first part: $\lceil n/2 \rceil$ elements
  - The second part: $\lfloor n/2 \rfloor$ elements
  - Sort the two subproblems recursively
    - Base case: array of size 1
- Then, combine the two sorted parts using an operation called merge.
  - After merge, the n elements are sorted

An example

- Step 1: Partition the initial array into two subarrays.
- Step 2: Sort each half
  - If the array only has one element, then it is sorted
  - Otherwise, return to Step 1 to sort it.
- Step 3: Merge the two sorted arrays



## Implementation:

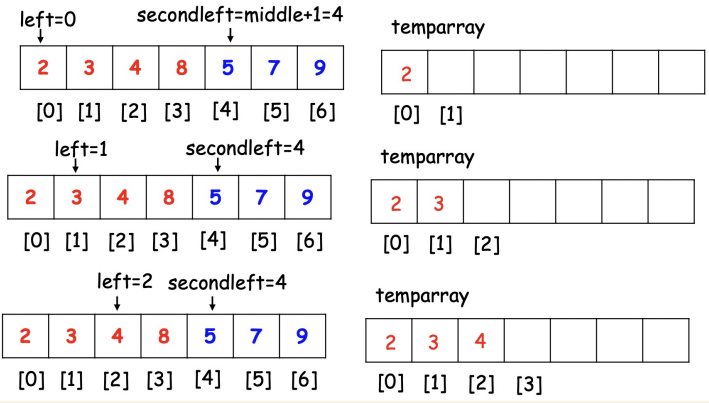
- If the array only has one element, then it is sorted
- Otherwise, we partition them into two subarrays
  - Then using recursion for these two subarrays
  - Using the left and right variable to record the start and end positions of the subarray, respectively

**Algorithm: *mergesort(arr, left, right)***

```
1  if left == right
2     return
3  middle ← (left+right)/2
4  mergesort(arr, left, middle)
5  mergesort(arr, middle+1, right)
6  merge(arr, left, middle, right)
```

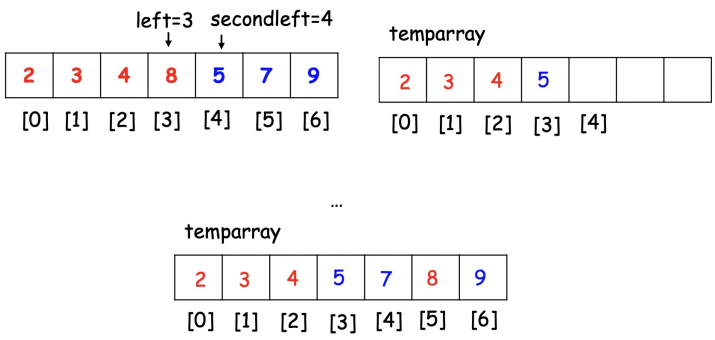
## Merge Operation

- `merge(arr, 0, 3, 6)`
  - left=0, middle=3, right=6, we use a temparray to store the merge result




## Time complexity:

- If we have $n_1$ elements in the first subarray and $n_2$ in the second subarray, what is the time complexity of merge?
  - $O(n_1 + n_2)$



## Merge Operation Implementation

Algorithm: ***merge(arr, left, middle, right)***

```
1  n=right-left+1; start = left
2  temparray ← a new array of size n
3  secondleft = middle+1
4  filled = 0
5  while left <= middle  and secondleft <=right //case 1: Neither of
6  the two subarray has reached the end
7     if arr[left] < arr[secondleft]
8         temparray[filled++] = arr[left++]
9     else
10        temparray[filled++] = arr[secondleft++]
11 while left <= middle //case 2: second subarray reached the end
12       temparray[filled++] = arr[left++]
13 while secondleft <= right //case 3: first subarray reached the end
14        temparray[filled++] = arr[secondleft++]
15 for i from 0 to n-1 //copy the sorted data back to arr
16     arr[start+i] = temparray[i]
17 free temparray // free the temparray
```

## Merge Operation Complexity Analysis

- Let $f(n)$ be the number of basic operations executed by merge sort

Algorithm: ***mergesort(arr, left, right)***

| | | |
|---|---|---|
| 1 | if left == right | $O(1)$ |
| 2 | return | $O(1)$ |
| 3 | middle ← (left+right)/2 | $O(1)$ |
| 4 | mergesort(arr, left, middle) | $f(\lceil n/2 \rceil)$ |
| 5 | mergesort(arr, middle+1, right) | $f(\lceil n/2 \rceil)$ |
| 6 | merge(arr, left, middle, right) | $O(n)$ |

What is the time complexity of merge sort? Which method shall we use to analyze the time complexity?

$$g(n) \le a \cdot g\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^{\lambda})$$

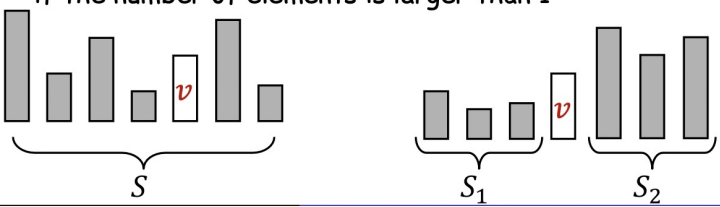
If $\log_b a < \lambda$, $g(n) = O(n^{\lambda})$
If $\log_b a = \lambda$, $g(n) = O(n^{\lambda} \cdot \log n)$
If $\log_b a > \lambda$, $g(n) = O(n^{\log_b a})$.

$$f(n) \le 2f\left(\left\lceil \frac{n}{2} \right\rceil\right) + c \cdot n$$

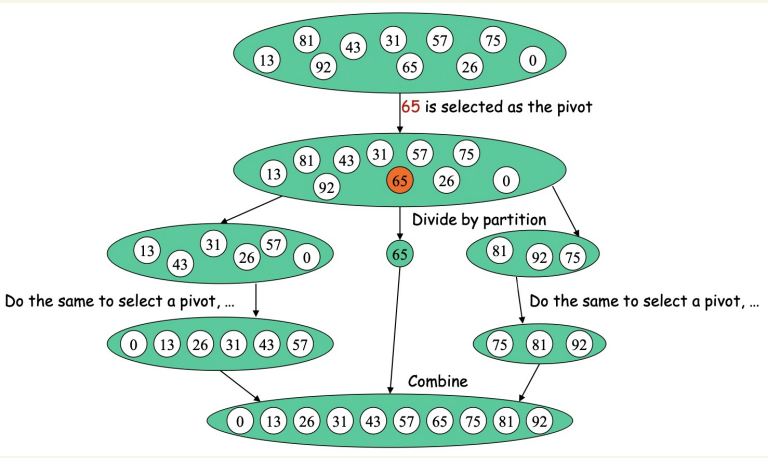
By master theorem: $a = 2, b = 2, \lambda = 1$. $f(n) = O(n \cdot \log n)$

# Quick sort (divide and conquer)

- A randomized algorithm using divide-and-conquer
  - Time complexity: $O(n \cdot \log n)$ expected running time
- High level idea: (we assume that elements are distinct)
  - Randomly pick an element, denoted as the pivot, and partition the remaining elements to three parts
    - The pivot
    - The elements in the left part: smaller than the pivot
    - The elements in the right part: larger than the pivot
    - For the left part and right part: repeat the above process if the number of elements is larger than 1




## An example



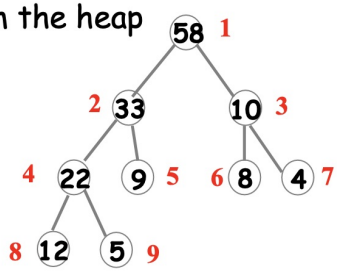

## Complexity Analysis : O(n·log n)

# Heap Sort & Sorting Lower Bound

Sorting with data structure

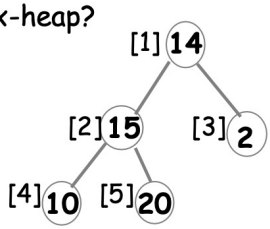
Max Heap : a max-tree + a complete binary tree

- To sort an array arr, we first create a heap *H* with a capacity of arr.length+1
- Then, we insert arr[0] into *H*, insert arr[1] into *H*,..., *insert arr[arr.length-1] into H.*
- Then, we repeatedly delete from the heap until the heap becomes empty




| 4 | 5 | 8 | 9 | 10 | 12 | 22 | 33 | 58 |
|---|---|---|---|---|---|---|---|---|

- In the previous solution, we need an additional array to create a max-heap, and then insert all numbers to this max-heap
  - Can we do faster to create a max heap?
  - Do we need to use two arrays?
- Reconsider an example of (14, 15, 2, 10, 20). We can still view it as a complete binary tree.
  - Can we make a complete binary tree as a max-heap?

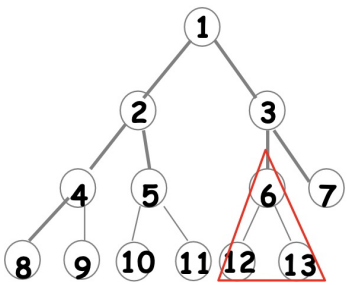


↙ 基于 Heap Adjust 实现 Heap sort

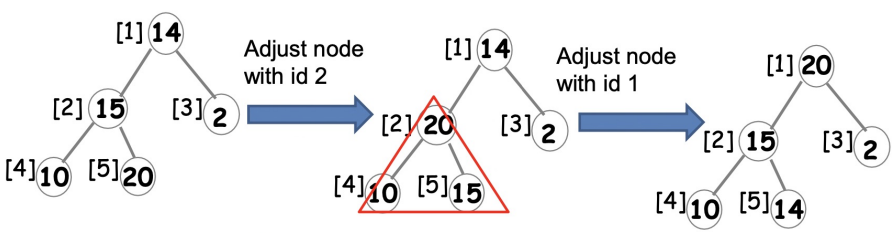
Heap Adjust: Given a node v, if its left-subtree and right-subtree are all max-tree, we can adjust the tree rooted at node v a max-tree

- Given node 6, both its left and right subtrees are max-trees (no violation). How to adjust to make the tree rooted as 6 a max-heap?
  - Hint: How we do the heap adjustment after the deletion?

During the adjustment, swap the node to be adjusted with the larger child




- For a complete binary tree, we adjust from node with id $\frac{n}{2}, \frac{n}{2}-1, \frac{n}{2}-2, \cdots 1$.
  - In the example of complete binary tree $C_1$, we only need to adjust from node with id 2 to id 1.




A complete binary tree $C_1$

- The implementation is quite similar to heap deletion

Heap Adjust Implementation

Algorithm 1: *HeapAdjust(heap, nodeid, heapsize)*

```
temp ← heap.arr[nodeid]
parent ← nodeid, child ← 2*nodeid
While child <= heapsize
    if child<heapsize and heap.arr[child].key<heap.arr[child+1].key
       child ← child + 1 //choose the larger child
    if temp.key >= heap.arr[child].key
        break // no violation if we put at position 'parent'
    heap.arr[parent]←heap.arr[child]//swap larger child with parent
    parent ← child //attempt to insert at the child position
    child ← 2*child
heap[parent] ← temp
```

## Heap Sort Implementation

- First adjust the nodes from $\frac{n}{2}, \frac{n}{2}-1, \frac{n}{2}-2, \cdots 1$ to make it a max-heap
  - Then, swap the largest number and the last element in the array, reduce the heap size by one and adjust the heap again.

Algorithm 1: ***HeapSort(arr, n)***

```
1  //sort arr[1..n]
2  //adjust the complete binary tree so that it becomes a max-heap
3  for i ← n/2 downto 1
4      adjust(arr, i, n)
5  //swap the root and last element, reduce the heap size, and adjsut
6  for i ← n − 1 downto 1 // i is the heap size
7      arr[i+1] ↔  arr[1] //swap the root and the last element
8      adjust(arr, 1, i) // adjust the heap of size i
```

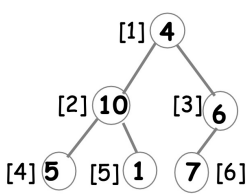
## Heap sort example

Sort the array $A[1 \cdots 6] = [4, 10, 6, 5, 1, 7]$ in ascending order by heap sort.

- Step 1: Adjust the heap to create a max heap. Given n nodes, we adjust from node n/2, then node n/2-1, …, and finally to node 1.

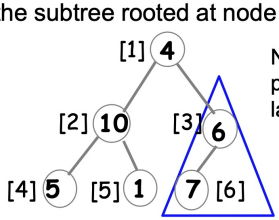
### Step 1

First, the complete binary tree corresponding to the array $A[1 \cdots 6] = [4,10,6,5,1,7]$ is:



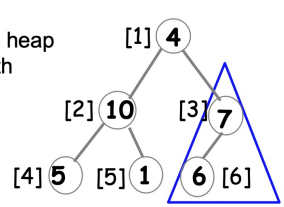

| Index | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Input data | 4 | 10 | 6 | 5 | 1 | 7 |

Adjust max heap from node id 3: Adjust the subtree rooted at node id 3




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 4 | 10 | 6 | 5 | 1 | 7 |

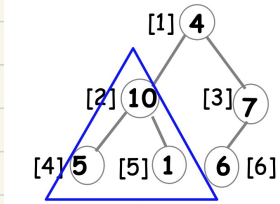
Node id 3 violates heap property, swap with larger child



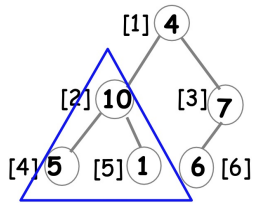

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 4 | 10 | 7 | 5 | 1 | 6 |

After adjust from node id 3: The subtree rooted at node 3 is a max-heap, the array becomes:

Adjust max heap from node id 2:
Adjust the subtree rooted at node 2



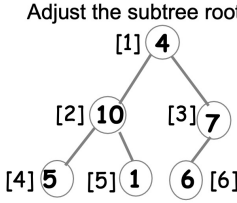

No violation




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 4 | 10 | 7 | 5 | 1 | 6 |

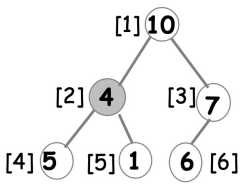
| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 4 | 10 | 7 | 5 | 1 | 6 |

Adjust max heap from node id 1:
Adjust the subtree rooted at node id 1



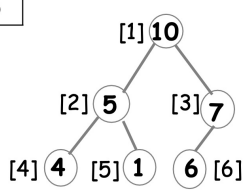

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 4 | 10 | 7 | 5 | 1 | 6 |

Node id 1 violates heap property, swap with larger child




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 10 | 4 | 7 | 5 | 1 | 6 |

After swapping, node id 2 violates the heap property, swap with larger child

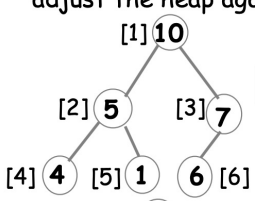



No violation. The array is now:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 10 | 5 | 7 | 4 | 1 | 6 |

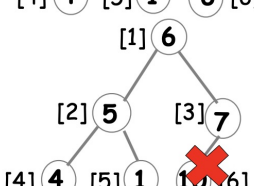
Step 2

- Step 2: sorting. Repeatedly swap the root (the first element) and the last element in the array, reduce the heap size by one and adjust the heap again.

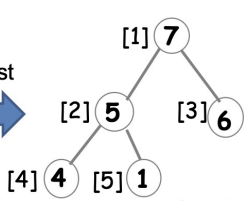



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 10 | 5 | 7 | 4 | 1 | 6 |

(i) Swap node id 6 and node id 1; reduce heap size by 1.

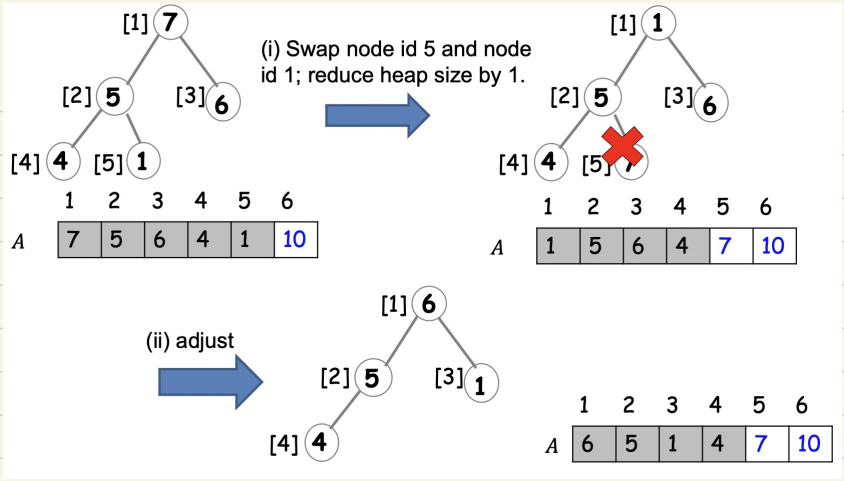


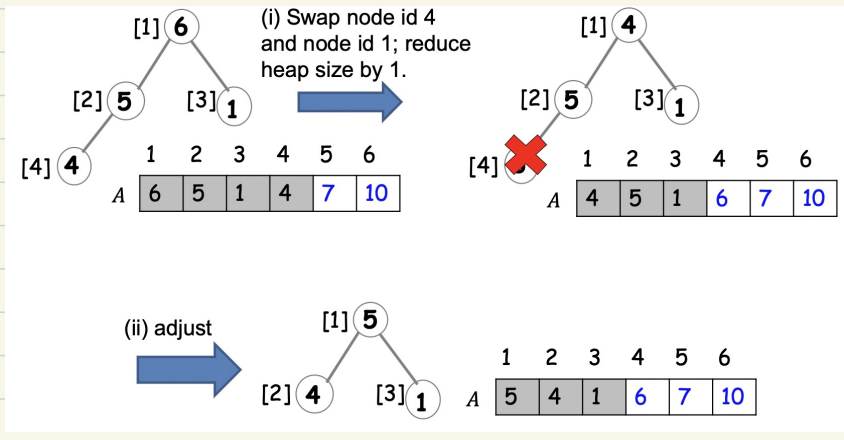

(ii) adjust

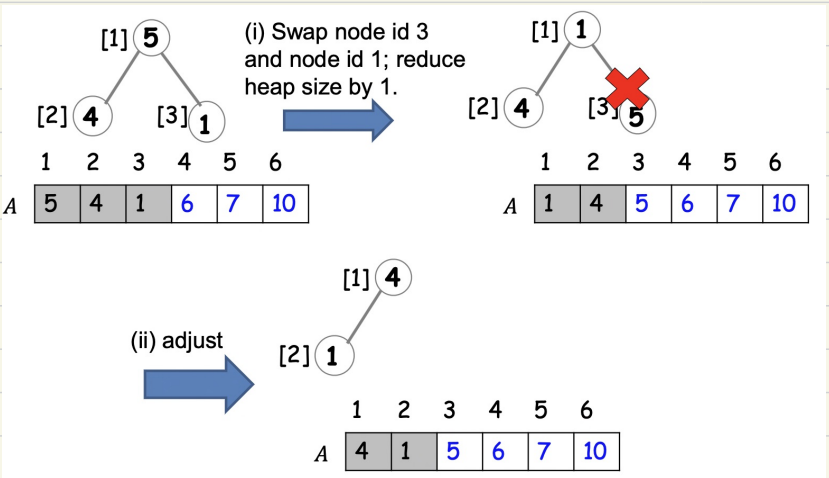


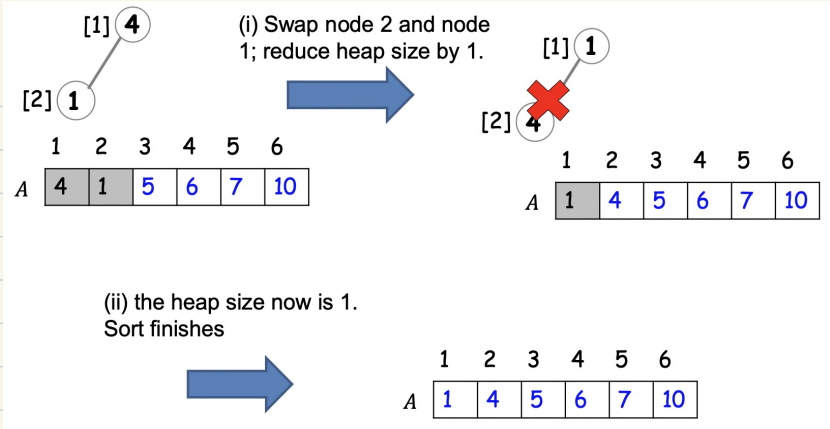

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 6 | 5 | 7 | 4 | 1 | 10 |

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A | 7 | 5 | 6 | 4 | 1 | 10 |


[1] 7
[2] 5
[3] 6
[4] 4
[5] 1
1 2 3 4 5 6
A 7 5 6 4 1 10
(i) Swap node id 5 and node id 1; reduce heap size by 1.
[1] 1
[2] 5
[3] 6
[4] 4
[5]
1 2 3 4 5 6
A 1 5 6 4 7 10
(ii) adjust
[1] 6
[2] 5
[3] 1
[4] 4
1 2 3 4 5 6
A 6 5 1 4 7 10


[1] 6
[2] 5
[3] 1
[4] 4
1 2 3 4 5 6
A 6 5 1 4 7 10
(i) Swap node id 4 and node id 1; reduce heap size by 1.
[1] 4
[2] 5
[3] 1
[4]
1 2 3 4 5 6
A 4 5 1 6 7 10
(ii) adjust
[1] 5
[2] 4
[3] 1
1 2 3 4 5 6
A 5 4 1 6 7 10


[1] 5
[2] 4
[3] 1
1 2 3 4 5 6
A 5 4 1 6 7 10
(i) Swap node id 3 and node id 1; reduce heap size by 1.
[1] 1
[2] 4
[3] 5
1 2 3 4 5 6
A 1 4 5 6 7 10
(ii) adjust
[1] 4
[2] 1
1 2 3 4 5 6
A 4 1 5 6 7 10




Heap Sort Complexity Analysis

adjust for $\frac{n}{2}$ nodes to make the complete binary tree a max-heap:

$C_{adjust} = O(n \cdot \log n)$

then repeated swap the root and the last element, and do the adjustment:

$C_{sort} = O(n \cdot \log n)$

total cost:

$C_{adjust} + C_{sort} = O(n \cdot \log n)$

# Hashing

Searching cost analysis : current best time complexity : $O(\log n)$

To preprocess the input ( to sort / build the binary search tree)

$O(n \log n)$ if we have $n$ records

To search the record

$O(\log n)$

What hashing achieves ?

To preprocess the input

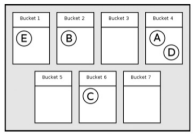
$O(n)$ if we have $n$ records

To search the record

$O(1)$ in expectation

Hash Tables : The main idea

- In direct-addressing, the record with key $k$ is stored in slot $k$ of the array $T$, i.e., $T[k]$
- In hashing, the element is stored in slot $h(k)$, i.e., $T[h(k)]$, where $h$ is a hash function.
  - Assume keys are integers in the range of $[0, U-1]$
  - Denote by $[x]$ the set of integers from 0 to $x-1$.
  - A hash function $h$ is a function from $[U]$ to $[m]$
    - For any integer k, $h(k)$ returns an integer in $[m]$
    - The value $h(k)$ is called the hash value of k
    - $U > m$



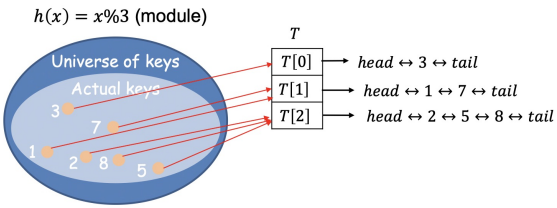
Problem: hashing collision

- If we have a set $S$ of keys $\{1,2,3,5,7,8\}$, and the hash function is $h(x) = x\%3$ (module)
  - What are the hash values of the integers in $S$?
  - $h(1) = 1\ \%3 = 1$
  - $h(2) = 2\%\ 3 = 2$
  - $h(3) = 3\ \%\ 3 = 0$
  - $h(5) = 5\ \%\ 3 = 2$ → Collision
  - $h(7) = 7\%\ 3\ = 1$
  - $h(8) = 8\ \%\ 3 = 2$ → Collision
  - What are $U$ and $m$?

## Hashing Collision Resolution: Chaining & Open Addressing

### Resolution 1: Chaining

- Collision: two keys hash to the same slot
- Chaining: We place all elements that hash to the same slot into the same linked list




Hash Table Initialization w. Chaining

insert(hashtable, key, record)

search(hashtable, key)

delete(hashtable, key)

- Assume that the hash function $h$ hashes keys to the range $[0, m-1]$
  - We then create an array of size $m$ where each entry stores the address of an empty linked list.

**Algorithm: *createTable(sizem)***

```
1 hashtable←allocate an array of size sizem
2 for i from 0 to sizem-1
3     hashtable[i] <- allocate an empty linkedlist
4 return hashtable
```

Analysis of Hashing w. Chaining

- Load factor $\alpha$: the average number of elements stored in a chain
  - If the hash table $T$ has size $m$ and we store $n$ elements in the hash table
    - $\alpha = \frac{n}{m}$
  - Assumption: *Uniform hashing* of $h(k)$
    - Elements are equally likely to hash into any of the $m$ slots.
    - $L_i$: the linked list containing the elements hashes to $i$.

> Theorem 1: In a hash table with collision resolved by chaining, the search/insertion/deletion takes $O(1+\alpha)$ time in expectation if $h(k)$ is uniform hashing.

  - By choosing $m$ so that $\frac{n}{m} = \alpha = O(1)$, the query time is $O(1)$.

## Resolution 2: Open Addressing

- All elements are stored in the hash table.
  - Unlike chaining, no elements are stored outside the table.
  - The hash table may "fill up" such no insertion can be made
    - Load factor $\alpha$ is always smaller than 1.
- For insertion, we examine, or probe, the hash table until an empty slot is found to put the key and records.
  - How to probe the hash table?
    - Linear probing
    - Quadratic Probing (Not discussed in the lecture)
    - Double Hashing
- Deletion: not efficiently supported. Think why?
- ADT Implementation : refer to our tutorial

## (a) Linear Probing

- For insertion, we probe $h(k)$, $h(k)+1$, $h(k)+2, \cdots m, 1, 2, \cdots, h(k)-1$ one by one until we find an empty slot, and insert the record to this slot.
  - More formally we probe $h(k,i) = (h(k)+i)\%m$ from $i=0$ to $i = m-1$ until we find an empty slot to insert.
- When searching for a record with a certain key,
  - Compute $h(k)$
  - Examine the hash table buckets in order $T[h(k,i)]$ for $0 \leq i \leq m-1$ until one of the following happens:
    - $T[h(k,i)]$ has the record with key equal to $k$.
    - $T[h(k,i)]$ is empty, then no record contains key $k$ in the hash table

## (b) Double Hashing

- Deficiency of Linear Probing
  - Long sequence of occupied slots, which degrades the query efficiency
- Double hashing
  - We have an additional hash function $h' > 0$.
  - Insertion: we probe $h(k,i) = \big(h(k) + i \cdot h'(k)\big)\%m$ one by one for $i$ from 0 to $m-1$ until an empty slot is found.
  - Search: we search $h(k,i)$ for $i$ from 0 to $m-1$ until one of the following happens:
    - $T[h(k,i)]$ has the record with key equal to $k$.
    - $T[h(k,i)]$ is empty, then no record contains key $k$ in the hash table

## Analysis of Open Addressing

- Load factor $\alpha$: If the hash table $T$ has size $m$ and we store $n$ elements in the hash table
  - $\alpha = \frac{n}{m}$
- Assumption: *Uniform hashing* of $h(k, i)$
  - The probing sequence $\langle h(k,0), h(k,1), h(k,2), \cdots, h(k, m-1)\rangle$ used to insert or search for each key $k$ is equally likely to be any permutation of $\langle 0,1,2, \cdots, m-1\rangle$.

> Theorem 2: In a hash table with collision resolved by open addressing, the search takes $O(\frac{1}{1-\alpha})$ time in expectation if $h(k,i)$ is uniform hashing.

  - If $\alpha = \Theta(1)$, the query time is $O(1)$

## Chaining vs. Open Addressing

- Pros of Chaining
  - Less sensitive to hash functions and load factors ($\alpha$ can be larger than 1), while open addressing requires to avoid long probes, and its load factor $\alpha < 1$.
  - Support deletion, while open addressing is difficult to support deletion (Why).
- Pros of Open addressing
  - Usually much faster than chaining

## Designing Hash Functions

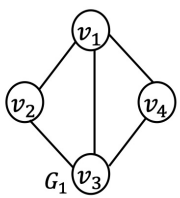
### Good Hash Function:

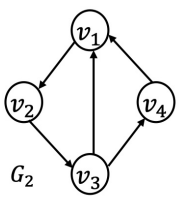
- A good hash function satisfies the uniform hashing property:
  - Each key is equally likely to hash to any of the $m$ slots, independent of where other keys will hash to.
- We learn two hashing functions: Division and universal hashing
  - Division is effective in practice without any theoretical guarantee on $O(1)$ query time.
  - Universal hashing provides theoretical guarantees on $O(1)$ query time

# Graph Basics & BFS Traversal

Graph def:

- A graph $G$ is defined as a pair of $(V, E)$ where:
  - $V$ is the set of objects, each of which is called a node (or a vertex)
  - $E$ is the set of edges, where each edge is a pair of two node $u$ and $v$
- Notations
  - $n$: the number of nodes, i.e., $|V|$
  - $m$: the number of edges, i.e., $|E|$




undirected / directed

Terms: neighbor and degree

- Neighbor: $v$ is called a neighbor of $u$ is there is an edge between $v$ and $u$.
  - In directed graphs, $v$ is called the out-neighbor of $u$ is there is an edge $\langle u, v\rangle$; $v$ is called the in-neighbor of $u$ is there is an edge $\langle v, u\rangle$.
- Degree $d(v)$ of a node $v$: the number of neighbors of this node $v$.
  - In directed graphs, the out-degree $d_{out}(v)$ of a node $v$ is the number of out-neighbors of this node; the in-degree $d_{in}(v)$ of a node $v$ is the number of in-neighbors of this node.

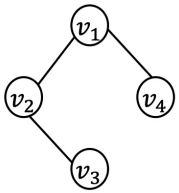
Theorem 1: In a undirected graph, the number $m$ of edges is equal to $\frac{1}{2}\sum_{v\in V} d(v)$. In a directed graph, $m = \sum_{v\in V} d_{out}(v) = \sum_{v\in V} d_{in}(v)$.

Path

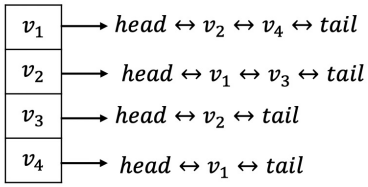
- A path from node $u$ to node $v$ in a graph $G$ is a sequence of nodes, $(u, v_1, \ldots, v_k, v)$ such that there exist a sequence of edges
  - $((u, v_1), (v_1, v_2), \ldots, (v_k, v))$ if $G$ is an undirected graph;
  - $(\langle u, v_1\rangle, \langle v_1, v_2\rangle, \ldots, \langle v_k, v\rangle)$ if $G$ is a directed graph.
- A simple path is a path in which all nodes except the first and last are distinct.
- A cycle is a simple path in which the first and the last nodes are the same.

Graph Representation
Adjacency List

- Adjacency list for undirected graph
  - Each node $v \in V$ is associated with a linked list that stores all neighbors of $v$, we map an ID for each node.



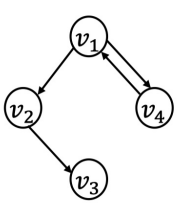

$G$



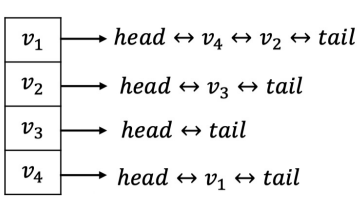

  - Space: $O(n + m)$, where $n$ is the number of nodes and $m$ is the number of edges

Adjacency list for directed graph

- Each node $v \in V$ is associated with a linked list that stores all out-neighbors of $v$




$G$

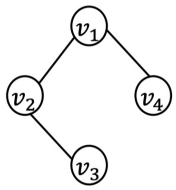



- Space: $O(n + m)$, where $n$ is the number of nodes and $m$ is the number of edges

Adjacency Matrix

## Adjacency matrix for undirected graph

- A $n \times n$ two dimensional matrix $A$ where $A[u][v] = 1$ if $(u, v) \in E$, or 0 otherwise.



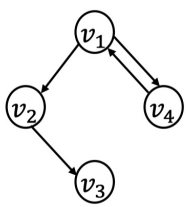

$G$

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | 1 | 0 | 1 |
| $v_2$ | 1 | 0 | 1 | 0 |
| $v_3$ | 0 | 1 | 0 | 0 |
| $v_4$ | 1 | 0 | 0 | 0 |

- $A$ is symmetric
- Space: $O(n^2)$ where $n$ is the number of nodes.

## Adjacency matrix for directed graph

- A $n \times n$ two dimensional matrix $A$ where $A[u][v] = 1$ if $\langle u, v \rangle \in E$, or 0 otherwise.




$G$

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | 1 | 0 | 1 |
| $v_2$ | 0 | 0 | 1 | 0 |
| $v_3$ | 0 | 0 | 0 | 0 |
| $v_4$ | 1 | 0 | 0 | 0 |

- $A$ may not be symmetric
- Space: $O(n^2)$ where $n$ is the number of nodes

Comparison:

Adjacency list:

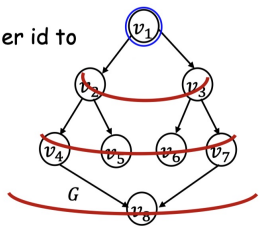
- Space: $O(n + m)$, save space if the graph is sparse, i.e., $m \ll n^2$
- Check the existence of an edge $(u, v)$: $O(k)$ time where $k$ is the number of neighbors of $v$
- Retrieve the neighbors of a node: $O(k)$ time.
- Add/delete a node: $O(n)$
- Add/delete an edge: $O(1)$

Adjacency matrix:

- Space consumption: $O(n^2)$
- Check the existence of an edge $(u, v)$: $O(1)$ time.
- Retrieve the neighbors of a node: $O(n)$ time
- Add/delete a node: $O(n^2)$, (create a new matrix)
- Add/delete an edge: $O(1)$

# Graph Traversal : Breadth-First Search (BFS)

- Intuition of BFS
  - Given a source node $s$, always visit nodes that are closer to the source $s$ first before visiting the others.
- The result is not unique, if we do not define an order among out-going edges from a node.
  - Possible results
    - $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$
    - $v_1, v_3, v_2, v_7, v_6, v_5, v_4, v_8$
  - We impose an order by going from smaller id to larger id. The result then will be unique.

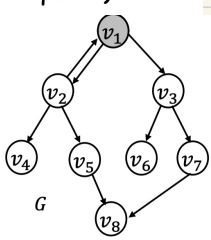



- At the beginning, color all nodes to be white
- Create a queue $Q$, enqueue the source $s$ to $Q$, and color the source to be gray (meaning $s$ is in the queue)

Example:

Assume the source is $v_1$.

$Q$: $(v_1)$



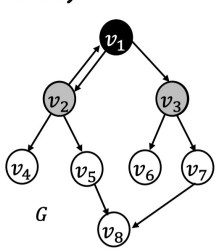

Repeat the following until queue $Q$ is empty

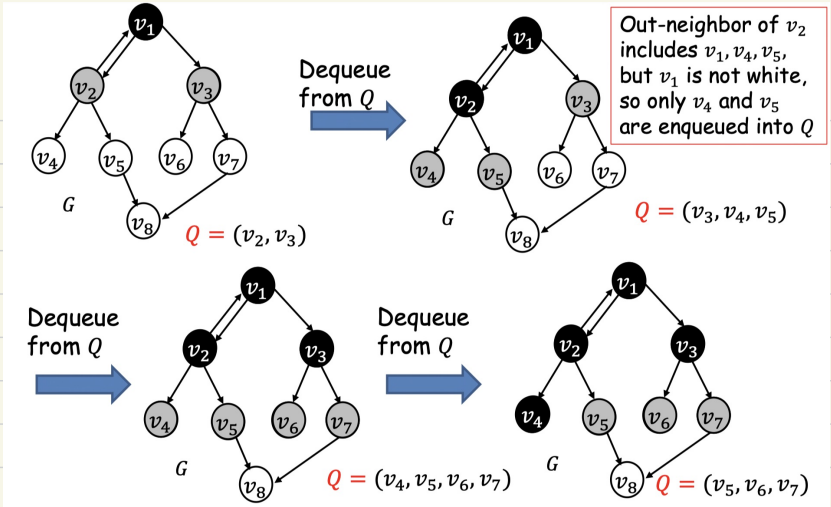
- Dequeue from $Q$, let the node be $v$.
- For every out-neighbor $u$ of $v$ that is still white
  - Enqueue $u$ into $Q$, and color $u$ to gray (to indicate $u$ is in queue)
- Color $v$ to be black (meaning $v$ has finished).

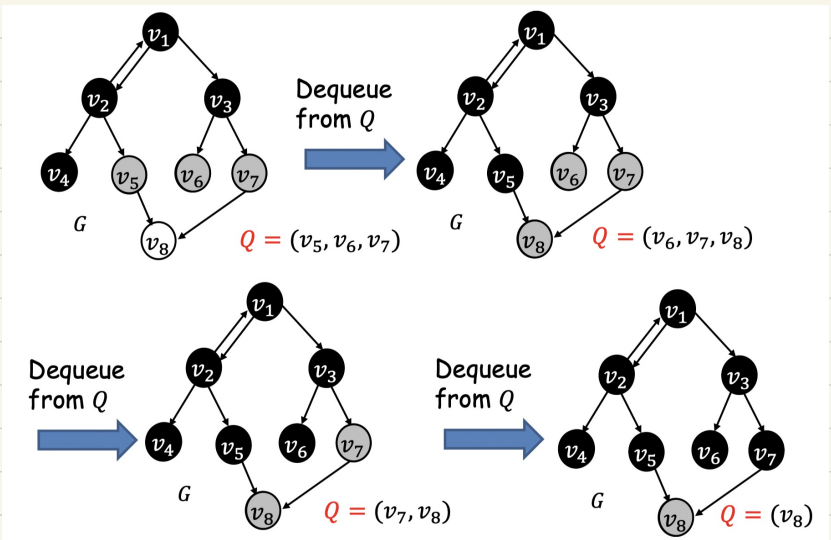
$Q = (v_1)$

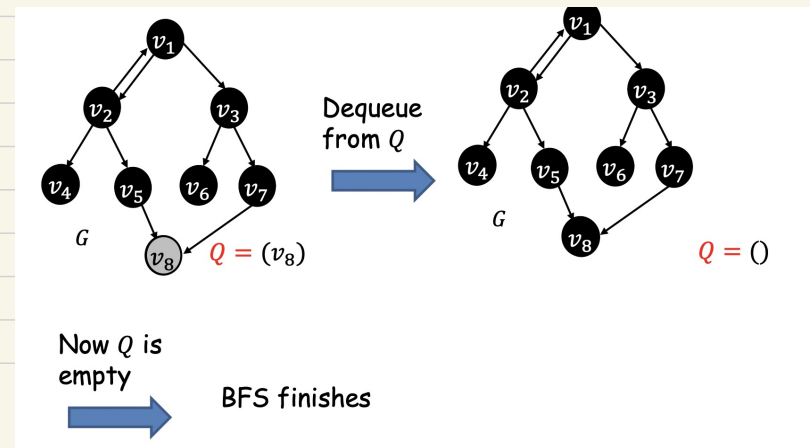
⬇ After dequeueing $v_1$

$Q = (v_2, v_3)$




Dequeue from $Q$
Out-neighbor of $v_2$ includes $v_1, v_4, v_5$, but $v_1$ is not white, so only $v_4$ and $v_5$ are enqueued into $Q$
$Q = (v_2, v_3)$
$Q = (v_3, v_4, v_5)$
Dequeue from $Q$
Dequeue from $Q$
$Q = (v_4, v_5, v_6, v_7)$
$Q = (v_5, v_6, v_7)$
$G$


Dequeue from $Q$
Dequeue from $Q$
$Q = (v_5, v_6, v_7)$
$Q = (v_6, v_7, v_8)$
Dequeue from $Q$
Dequeue from $Q$
$Q = (v_7, v_8)$
$Q = (v_8)$
$G$


Dequeue from $Q$
$Q = (v_8)$
$Q = ()$
Now $Q$ is empty
BFS finishes
$G$

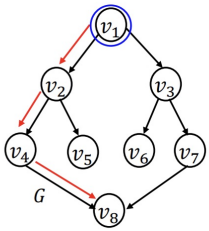
Time complexity

- When a node $u$ is dequeued
  - We examine all of its neighbors (check their color), enqueue them and color them to gray if they are white.
  - After that, we color $u$ as black
  - This incurs $c(1 + d_{out}(u))$ costs for node $u$.
- Each node is dequeued at most once.
  - Why?
  - We enqueue a node at most once. If it is in the queue, its color is gray and we will not further enqueue it.
- Therefore, the total running time is
  - $\sum_{u \in V} c(1 + d_{out}(u)) = c(n + m) = O(n + m)$

# Depth-First Search

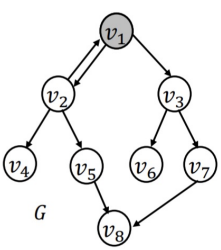
DFS:

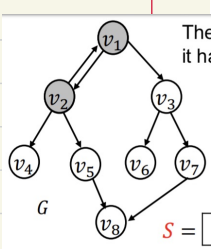
- Going along one path until we cannot go further
- Imposing an order to make the traversal unique: from smaller id to larger id
  - Visiting order: $v_1, v_2, v_4, v_8, v_5, v_3, v_6, v_7$
- We still focus on directed graph
  - Extension to undirected graph will be straightforward




- Initialization:
  - At the beginning, color all nodes to be white
  - Create a stack $S$, push the source $s$ to $S$, and color the source to be gray (meaning $s$ is in the stack)
- Example:
- Assume that $v_1$ is the source

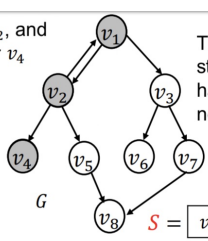
$S =$ | $v_1$ |



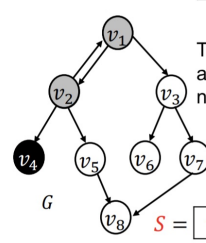

The top of the stack is $v_2$, and it has white out-neighbor $v_4$

$S =$ | $v_1$ | $v_2$ |



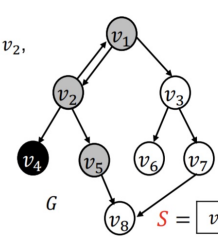

The top of the stack is $v_4$, and it has no white out-neighbor

$S =$ | $v_1$ | $v_2$ | $v_4$ |

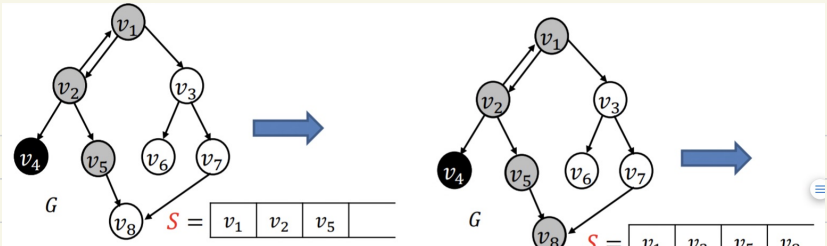


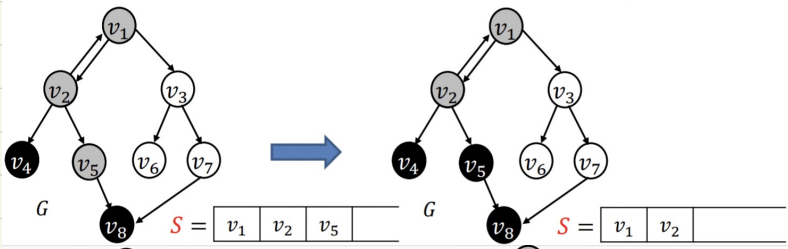

The top of the stack is $v_2$, and it has white out-neighbor $v_5$

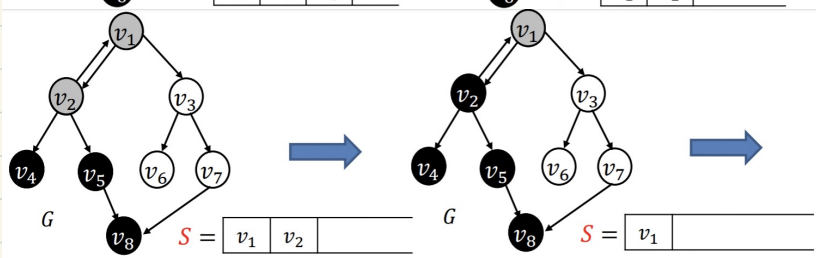
$S =$ | $v_1$ | $v_2$ |

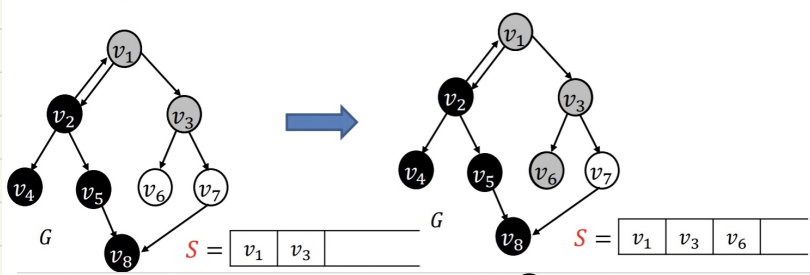


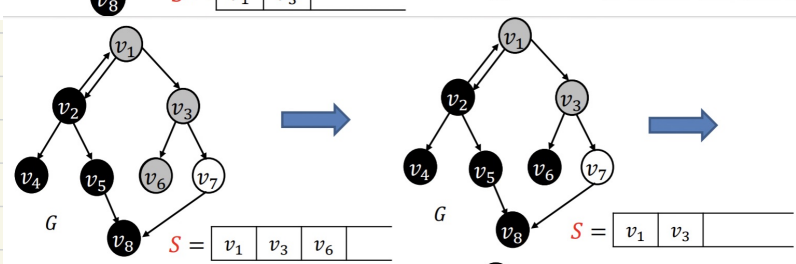

$S =$ | $v_1$ | $v_2$ | $v_5$ |

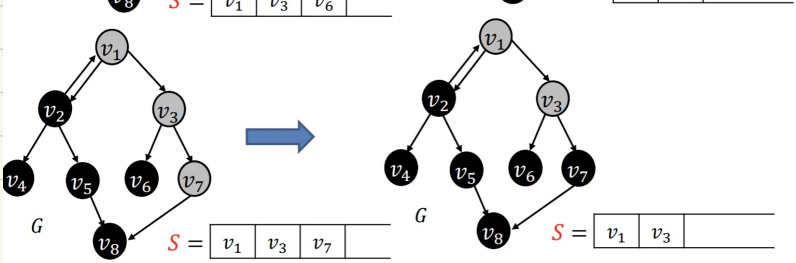

$v_1$ $v_2$ $v_3$ $v_4$ $v_5$ $v_6$ $v_7$ $v_8$
$G$
$S = \{v_1, v_2, v_5\}$
$S = \{v_1, v_2, v_5, v_8\}$

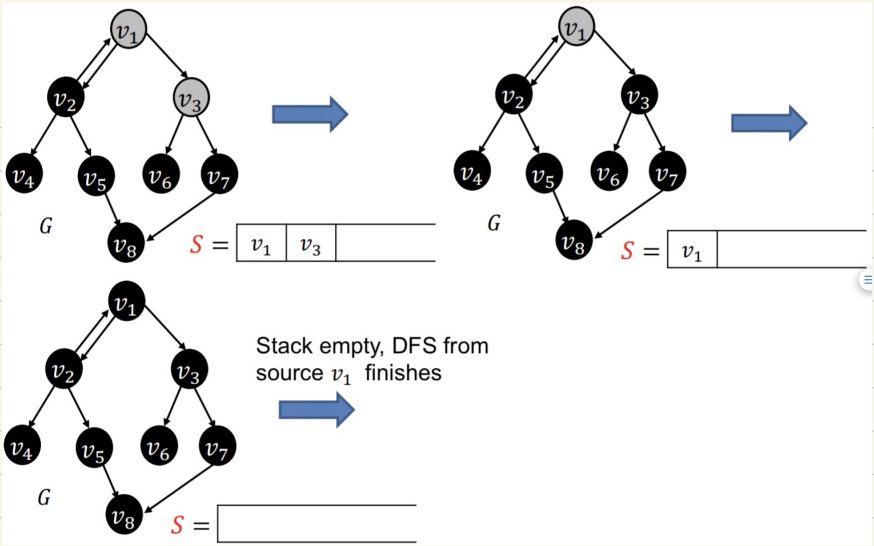

$v_1$ $v_2$ $v_3$ $v_4$ $v_5$ $v_6$ $v_7$ $v_8$
$G$
$S = \{v_1, v_2, v_5\}$
$S = \{v_1, v_2\}$


$v_1$ $v_2$ $v_3$ $v_4$ $v_5$ $v_6$ $v_7$ $v_8$
$G$
$S = \{v_1, v_2\}$
$S = \{v_1\}$


$v_1$ $v_2$ $v_3$ $v_4$ $v_5$ $v_6$ $v_7$ $v_8$
$G$
$S = \{v_1, v_3\}$
$S = \{v_1, v_3, v_6\}$


$v_1$ $v_2$ $v_3$ $v_4$ $v_5$ $v_6$ $v_7$ $v_8$
$G$
$S = \{v_1, v_3, v_6\}$
$S = \{v_1, v_3\}$


$v_1$ $v_2$ $v_3$ $v_4$ $v_5$ $v_6$ $v_7$ $v_8$
$G$
$S = \{v_1, v_3, v_7\}$
$S = \{v_1, v_3\}$




Complexity Analysis:

- When a node $v$ get popped from the stack?
  - None of its out-neighbors is a white node
  - We may need to repeated check if node $v$ has white out-neighbor. If $v$ has $d_{out}(v)$ out-neighbors, we need to check $d_{out}(v)$ times in the worst case.
    - Cost: $d_{out}(v) \cdot d_{out}(v)$?
    - Can we do better?
      - We record the position checked last time. All nodes in previous positions will not be white.
      - Cost: $O(d_{out}(v))$
- Each node is popped at most once.
- Therefore, the total running time is
  - $\sum_{u \in V} c(1 + d_{out}(u)) = c(n + m) = O(n + m)$