



# Data Abstractions

Introduction to Computer  
Yu-Ting Wu

1

## Outline

- Arrays
- Lists
- Stacks
- Queues
- Trees

2

2

## Data Structure Concepts

- Example: give an array, find the minimal element
  - How about doing this step 100 times
  - Sorting?
  - But if we need to insert a new element or update an old element?
- **Static** v.s. **dynamic** structures

3

3

## Outline

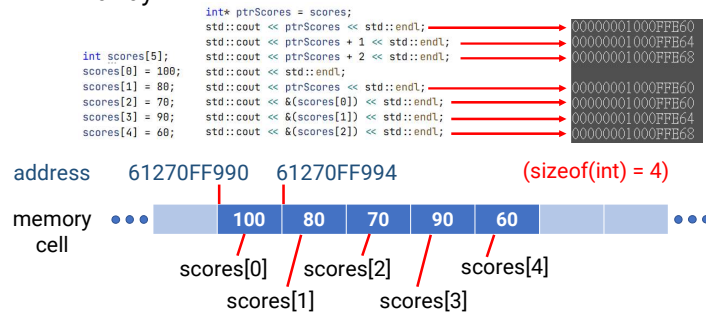
- Arrays
- Lists
- Stacks
- Queues
- Trees

4

4

## Homogeneous Arrays

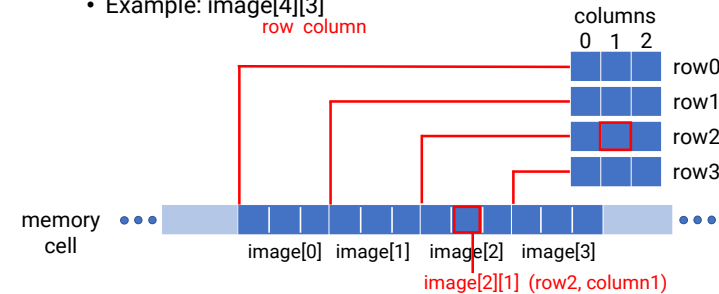
- Block of data whose entries are of the **same type (and size)**
- Indices are used to identify positions
- 1D array



5

## Homogeneous Arrays (cont.)

- Block of data whose entries are of the **same type (and size)**
- 2D array: array of arrays
  - Consist of rows and columns
  - Example: image[4][3]



6

## Homogeneous Arrays (cont.)

- Array addresses

x = 1000

x[0] = 1000	x[0][0] 1000	x[0][1] 1004	x[0][2] 1008	x[0][3] 1012
x[1] = 1016	x[1][0] 1016	x[1][1] 1020	x[1][2] 1024	x[1][3] 1028
x[2] = 1032	x[2][0] 1032	x[2][1] 1036	x[2][2] 1040	x[2][3] 1044

int x[3][4]

address polynomial

sizeof(int) = 4

int x[1][2] = 1000 + 1 \* 4 \* 4 + 2 \* 4 = 1024

x[1]

7

## Homogeneous Arrays (cont.)

- Parameter passing

- Does it work?

```
void UpdateArray2D(int **ptrX)
{
    ptrX[2][3] = 5;
}

void UpdateArray2D(int ptrX[3][4])
{
    ptrX[2][3] = 5;
}

int main()
{
    int x[3][4];
    UpdateArray2D(x);
}

void UpdateArray2D(int ptrX[][4])
{
    ptrX[2][3] = 5;
}
```

obs E0167 類型 "int (\*)[4]" 的引數與類型 "int \*\*\*" 的參數不相容

C2664 'void UpdateArray2D(int \*\*)': 無法將引數 1 從 'int [3][4]' 轉換為 'int \*\*'

Why? no enough information for address polynomial  
Need the number of elements per row

8

## Heterogeneous Arrays

- Structure: a block of data items that might be of different types or sizes
- Each data item is called a field (accessed by name)

```

struct Student
{
    char name[20];
    char department[20];
    int grade;
    int ICMidtermScore;
};

Student student;
std::cout << &(student) << std::endl;
std::cout << &(student.name) << std::endl;
std::cout << &(student.department) << std::endl;
std::cout << &(student.grade) << std::endl;
std::cout << &(student.ICMidtermScore) << std::endl;

```

9

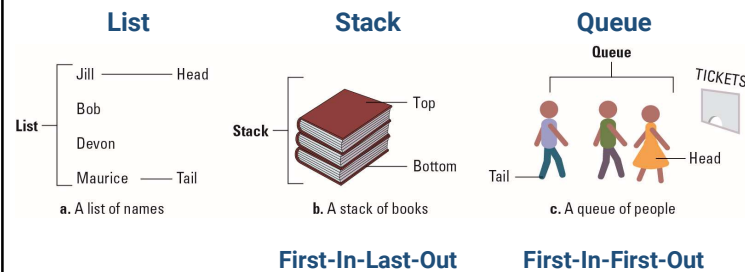
## Outline

- Arrays
- Lists
- Stacks
- Queues
- Trees

10

## Lists, Stacks, and Queues

- List: a collection of data whose entries are arranged sequentially
- **Stacks** and **Queues** are specialized lists



11

## Operations of a List

- **Empty()**
  - Return true if the list is empty
- **Size()**
  - Return the number of elements in the list
- **GetElement(index)**
  - Return the element with the given index
- **EraseElement(index)**
  - Remove the element at the index
- **Insert(index, data)**
  - Insert a new element with the given data at the given index

12

## Storing Lists

### • Contiguous list (array)

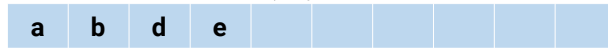
- **Pros:** easy to implement, an excellent choice for static use
- **Cons:** time-consuming for dynamic use, fragments may occur without careful implementation
- Example:  $L = (a, b, c, d, e)$  using an array representation



- After deleting the third element, choices are
  - Leave the holes (time-consuming for later use)



- Make it compact (easy for following OPs, but need to move)



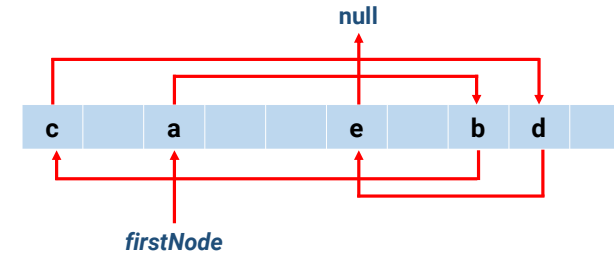
13

13

## Storing Lists (cont.)

### • Linked list

- Head pointer: indicate the start
- NULL pointer: indicate the end
- Example:  $L = (a, b, c, d, e)$  using a linked representation



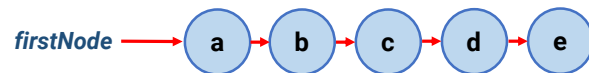
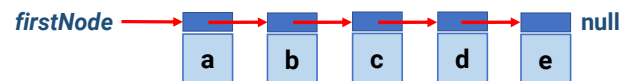
14

14

## Storing Lists (cont.)

### • Linked list

- Head pointer: indicate the start
- NULL pointer: indicate the end
- Example:  $L = (a, b, c, d, e)$  using a linked representation



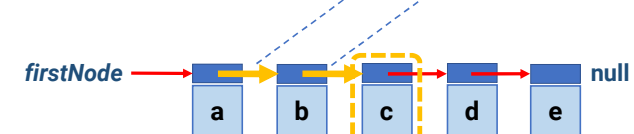
15

15

## List: Get an Element with its Index

### • Procedure

- Start from the first node
- List::GetElement(2) **assume the index starts with 0**
  - Target node =  $firstNode \rightarrow next \rightarrow next;$

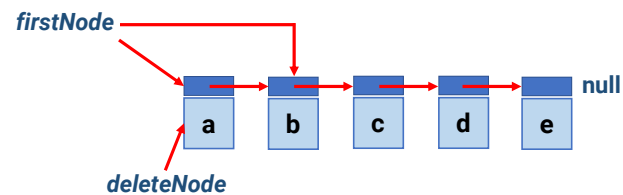


16

16

## List: Erase an Element with its Index

- Case 1: erase the first element
- **Procedure**
  - Use a pointer to identify the first node (*deleteNode*)

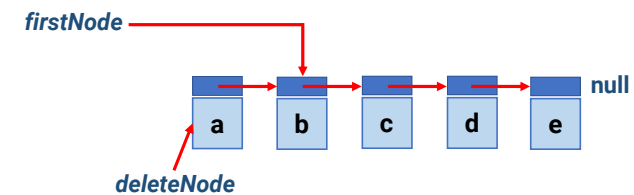


17

17

## List: Erase an Element with its Index

- Case 1: erase the first element
- **Procedure**
  - Use a pointer to identify the first node (*deleteNode*)
  - Change *firstNode* pointer to the second node

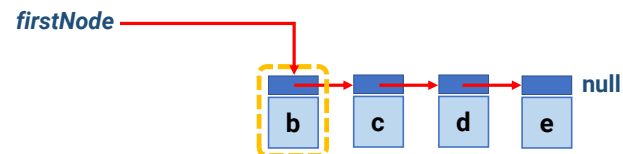


18

18

## List: Erase an Element with its Index

- Case 1: erase the first element
- **Procedure**
  - Use a pointer to identify the first node (*deleteNode*)
  - Change *firstNode* pointer to the second node
  - Delete the *deleteNode*

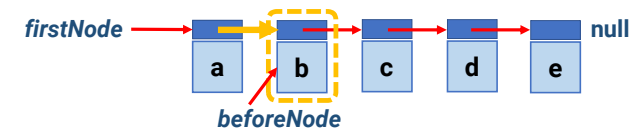


19

19

## List: Erase an Element with its Index (cont.)

- Case 2: an element that is not the first one
- **Procedure**
  - Get the node before the target index (*beforeNode*)
- Example: EraseElement(2)



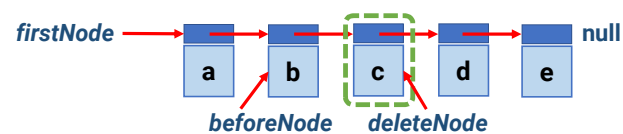
20

20

## List: Erase an Element with its Index (cont.)

- Case 2: an element that is not the first one
- **Procedure**
  - Get the node before the target index (*beforeNode*)
  - Identify the node to be deleted (*deleteNode*)

- Example: EraseElement(2)



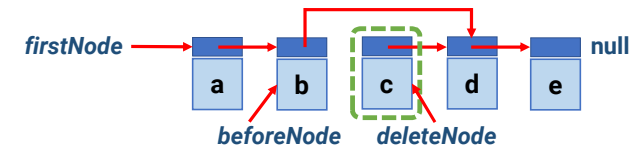
21

21

## List: Erase an Element with its Index (cont.)

- Case 2: an element that is not the first one
- **Procedure**
  - Get the node before the target index (*beforeNode*)
  - Identify the node to be deleted (*deleteNode*)
  - Change pointer in *beforeNode*

- Example: EraseElement(2)



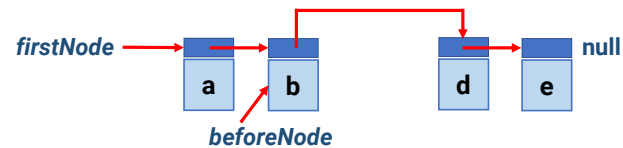
22

22

## List: Erase an Element with its Index (cont.)

- Case 2: an element that is not the first one
- **Procedure**
  - Get the node before the target index (*beforeNode*)
  - Identify the node to be deleted (*deleteNode*)
  - Change pointer in *beforeNode*
  - Delete the *deleteNode*

- Example: EraseElement(2)



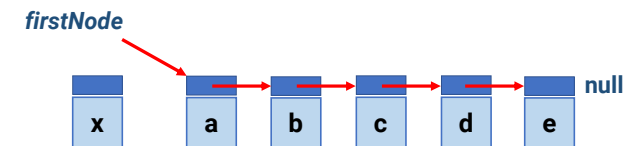
23

23

## List: Insert a New Element

- Case 1: insert at front
- **Procedure**
  - Create a new node with the given data

- Example: Insert(0, x)

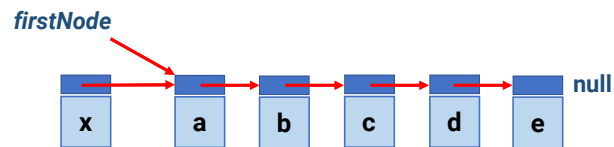


24

24

### List: Insert a New Element (cont.)

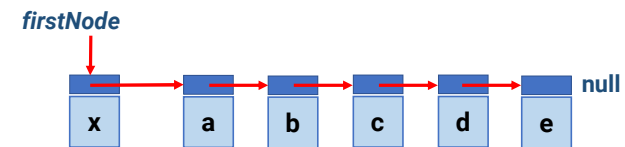
- Case 1: insert at front
- **Procedure**
  - Create a new node with the given data
  - Set the pointer of the new node to the original first node
- Example: Insert(0, x)



25

### List: Insert a New Element (cont.)

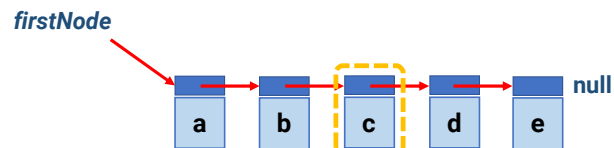
- Case 1: insert at front
- **Procedure**
  - Create a new node with the given data
  - Set the pointer of the new node to the original first node
  - Update the *firstNode* pointer
- Example: Insert(0, x)



26

### List: Insert a New Element (cont.)

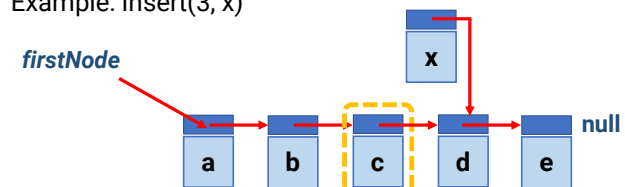
- Case 2: insert in the middle
- **Procedure**
  - Find the node before the target (*beforeNode*)
- Example: Insert(3, x)



27

### List: Insert a New Element (cont.)

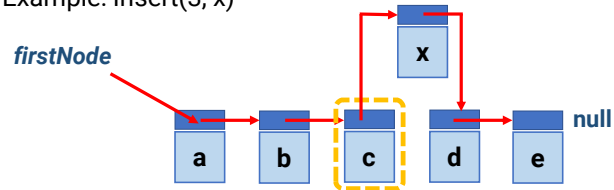
- Case 2: insert in the middle
- **Procedure**
  - Find the node before the target (*beforeNode*)
  - Create a new node with the given data and set its pointer
- Example: Insert(3, x)



28

## List: Insert a New Element (cont.)

- Case 2: insert in the middle
- **Procedure**
  - Find the node before the target (*beforeNode*)
  - Create a new node with the given data and set its pointer
  - Change the pointer of *beforeNode* to the new node
- Example: Insert(3, x)

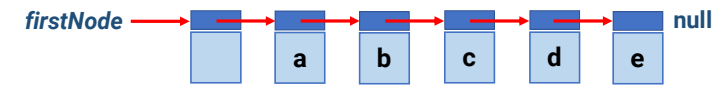


29

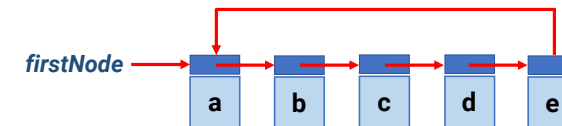
29

## Variations

- List with a **dummy** header node



- Circular list

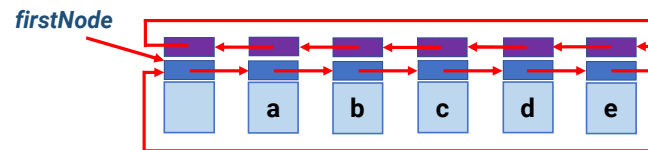


30

30

## Variations (cont.)

- Doubly linked circular list with header
- Efficient for inserting at the end
- C++ Standard Template Library (STL) adopts this implementation (std::list)
  - <https://en.cppreference.com/w/cpp/container/list>



31

31

## Outline

- Arrays
- Lists
- Stacks
- Queues
- Trees

32

32



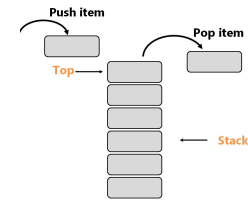
## Stacks and Queues

- Special cases of linked list
  - **Stack**: record the stack point
  - **Queue**: record head and tail
- Both can be implemented either using contiguous memory (array) or linked list
  - Contiguous (array) implementation is more common

33

## Stacks

- Stack: a list in which entries are removed and inserted only at the head
- **Last-in-first-out (LIFO)**
- Operations
  - **Top**: get the head of the list (stack)
  - **Pop**: to remove the entry at the top
  - **Push**: to insert an entry at the top



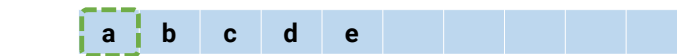
34

33

34

## Storing Stacks

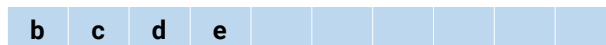
- Derive from array
  - Stack top is either left end or right end
  - When the top is left end



- Push:  $\Theta(n)$ 
  - Need to move all elements right



- Pop:  $\Theta(n)$ 
  - Need to move all elements left



35

35

## Storing Stacks (cont.)

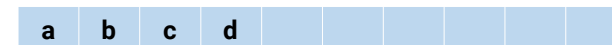
- Derive from array
  - Stack top is either left end or right end
  - When the top is right end



- Push:  $\Theta(1)$ 
  - **No** need to move all elements right



- Pop:  $\Theta(1)$ 
  - **No** need to move all elements left

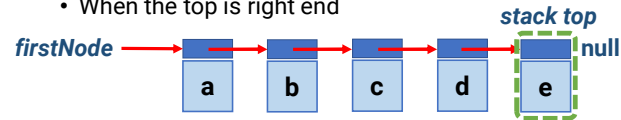


36

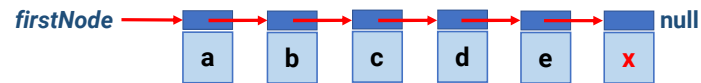
36

## Storing Stacks (cont.)

- Derive from linked list
  - Stack top is either left end or right end
  - When the top is right end



- Push:  $\Theta(n)$ 
  - Need to traverse all nodes

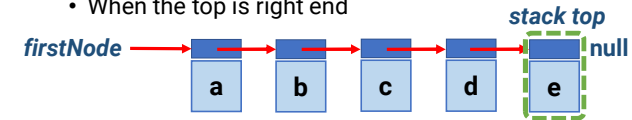


37

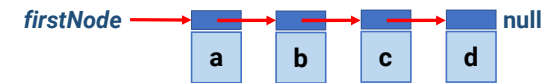
37

## Storing Stacks (cont.)

- Derive from linked list
  - Stack top is either left end or right end
  - When the top is right end



- Pop:  $\Theta(n)$ 
  - Need to traverse all nodes

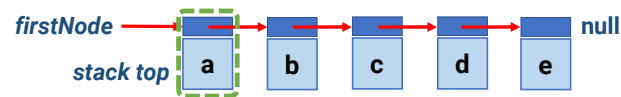


38

38

## Storing Stacks (cont.)

- Derive from linked list
  - Stack top is either left end or right end
  - When the top is left end



- Push:  $\Theta(1)$ 
  - Insert at front
- Pop:  $\Theta(1)$ 
  - EraseElement at front

39

39

## Parentheses Matching

(	(	(	a	+	b	)	*	c	+	d	-	e	)	/	(	f	+	g	)	-	(	h	+	j	)	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

- Output pairs  $(u, v)$  such that the left parenthesis at position  $u$  is matched with the right parenthesis at  $v$ 
  - $(2, 6), (1, 13), (15, 19), (21, 25), (0, 26)$

- Also report missing pair parentheses

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	10	11	12

- $(0, 4), (\text{missing}, 5), (8, 12), (7, \text{missing})$

40

40

## Parentheses Matching (cont.)

- Scan expression from left to right
- When a left parenthesis is encountered, **push** its position to the stack
- When a right parenthesis is encountered, **pop** matching position from the stack

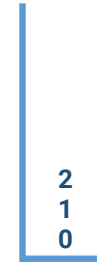
41

## Parentheses Matching (cont.)

(	(	(	a	+	b	)	*	c	+	d	-	e	)	/	(	f	+	g	)	-	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	
										0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6

### Actions

Push 0  
Push 1  
Push 2



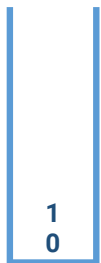
42

## Parentheses Matching (cont.)

(	(	(	a	+	b	)	*	c	+	d	-	e	)	/	(	f	+	g	)	-	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	
										0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6

### Actions

Push 0  
Push 1  
Push 2  
Pop, output (2, 6)



43

## Parentheses Matching (cont.)

(	(	(	a	+	b	)	*	c	+	d	-	e	)	/	(	f	+	g	)	-	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	
										0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6

### Actions

Push 0  
Push 1  
Push 2  
Pop, output (2, 6)  
Pop, output (1, 13)



44

Introduction to Computer 2022

### Parentheses Matching (cont.)

(	(	(	a	+	b	)	*	c	+	d	-	e	)	/	(	f	+	g	)	-	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

**Actions**

Push 0

Push 1

Push 2

Pop, output (2, 6)

Pop, output (1, 13)

Push 15

45

45

Introduction to Computer 2022

### Parentheses Matching (cont.)

(	(	(	a	+	b	)	*	c	+	d	-	e	)	/	(	f	+	g	)	-	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

**Actions**

Push 0

Push 1

Push 2

Pop, output (2, 6)

Pop, output (1, 13)

Push 15

Pop, output (15, 19)

Push 21

46

46

Introduction to Computer 2022

### Parentheses Matching (cont.)

(	(	(	a	+	b	)	*	c	+	d	-	e	)	/	(	f	+	g	)	-	(	h	+	j	)	)
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

**Actions**

Push 0

Push 1

Push 2

Pop, output (2, 6)

Pop, output (1, 13)

Push 15

Pop, output (15, 19)

Push 21

Pop, output (21, 25)

Pop, output (0, 26)

47

47

Introduction to Computer 2022

### Parentheses Matching (cont.)

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	10	11	12

**Actions**

Push 0

48

48

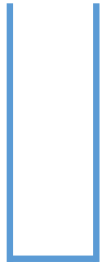
## Parentheses Matching (cont.)

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	10	11	12

### Actions

Push 0

Pop, output (0, 4)



49

## Parentheses Matching (cont.)

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	10	11	12

### Actions

Push 0

Pop, output (0, 4)

Pop, **error for stack is empty!**



50

## Parentheses Matching (cont.)

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	10	11	12

### Actions

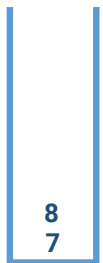
Push 0

Pop, output (0, 4)

Pop, **error for stack is empty!**

Push 7

Push 8



51

## Parentheses Matching (cont.)

(	a	+	b	)	)	*	(	(	c	+	d	)
0	1	2	3	4	5	6	7	8	9	10	11	12

### Actions

Push 0

Pop, output (0, 4)

Pop, **error for stack is empty!**

Push 7

Push 8

Pop, output (8, 12)

**error for the left parenthesis at 7 is not matched any right parenthesis**



52

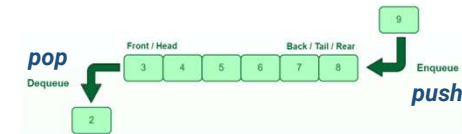
## Outline

- Arrays
- Lists
- Stacks
- Queues
- Trees

53

## Queues

- Queue: a list in which entries are removed at the head and are inserted at the tail
- First-in-first-out (FIFO)
- Operations
  - Front: get the value of the front element
  - Back: get the value of the back element
  - Pop: remove the front element
  - Push: add an element at the back of the queue



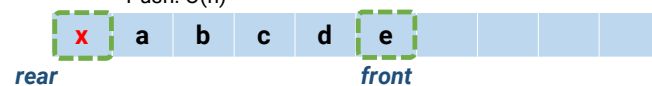
54

## Storing Queues

- Derive from array
  - Two choices for the front and rear
  - When the front is right end and rear is left end



- Front:  $\Theta(1)$
- Back:  $\Theta(1)$
- Pop:  $\Theta(1)$
- Push:  $\Theta(n)$



55

## Storing Queues (cont.)

- Derive from linked list
  - Two choices for the front and rear
  - When the front is right end and rear is left end



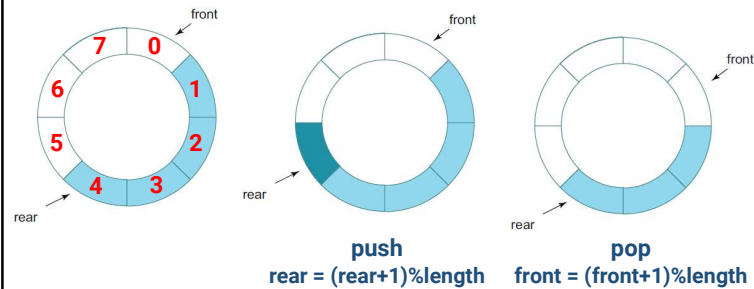
- Front:  $\Theta(1)$
- Back:  $\Theta(1)$
- Pop:  $\Theta(n)$
- Push:  $\Theta(1)$



56

## Storing Queues (cont.)

- Derive from linked list
  - We can do better ( $\Theta(1)$  for both pushing and pop) by using a customized array representation
  - Circular + **mod** operation (w.r.t the array length)



57

57

## Storing Queues (cont.)

- Derive from linked list
  - Handle **empty** and **full** queue (both  $\text{front} == \text{rear}$ )
  - Use a **size** variable
    - When pushing,  $++\text{size}$
    - When popping,  $--\text{size}$
    - Queue is empty iff ( $\text{size} == 0$ )
    - Queue is full iff ( $\text{size} == \text{length}$ )



58

58

## Outline

- Arrays
- Lists
- Stacks
- Queues
- Trees

59

59

## Tree

- Lists are useful for serially ordered data
- Trees are useful for hierarchically ordered data

60

60

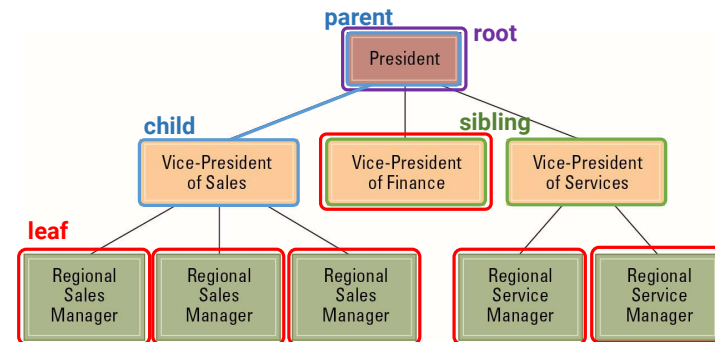
## Terminology for a Tree

- **Node:** an entry in a tree
- **Parent:** the node immediately above a specified node
- **Child:** a node immediately below a specified node
- **Ancestor:** parent, parent of the parent, etc.
- **Descendent:** child, child of a child, etc.
- **Siblings:** nodes sharing a common parent
- **Root node:** the node at the top
- **Leaf node:** the node at the bottom (thus has no children)

61

61

## Terminology for a Tree (cont.)

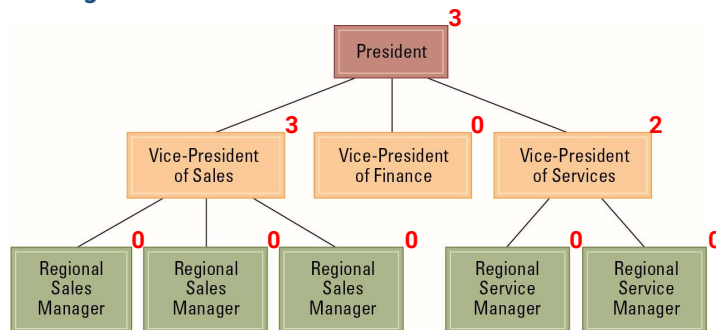


62

62

## Terminology for a Tree (cont.)

- **Degree:** number of children



63

63

## Definition of Tree

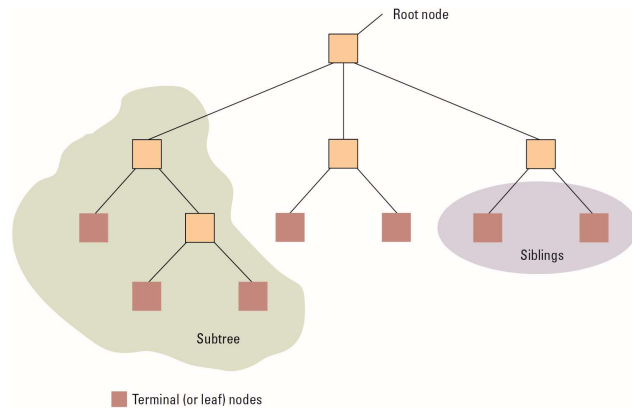
- **Recursive** definition
- A tree  $t$  is a finite non-empty set of elements
- One of these elements is called the **root**
- The remaining elements, if any, are partitioned into trees, which are called the subtrees of  $t$

64

64



## Definition of Tree (cont.)

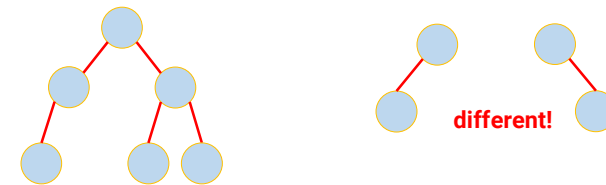


65

65

## Binary Trees

- Finite non-empty collection of elements
- A binary tree has a root element
- The remaining elements (if any) are partitioned into at most **two** binary trees
  - Called the **left** and **right** subtrees



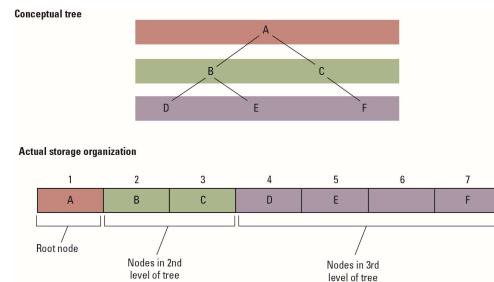
66

66

## Storing Binary Trees

### • Contiguous array structure

- Root node: A[1]
- Children of A[1]: A[2] and A[3]
- Children of A[2] and A[3]: A[4], A[5], A[6], and A[7]



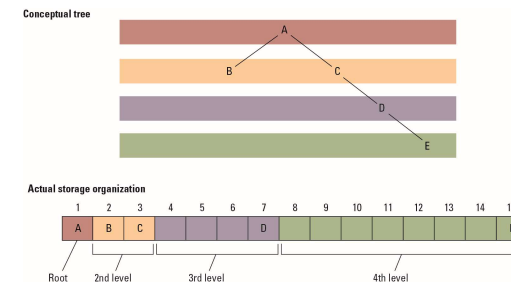
67

67

## Storing Binary Trees

### • Contiguous array structure

- Root node: A[1]
- Children of A[1]: A[2] and A[3]
- Children of A[2] and A[3]: A[4], A[5], A[6], and A[7]



68

68

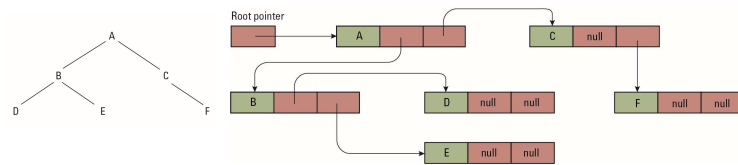
## Storing Binary Trees

### • Linked structure

- Each node = data cells + two child pointers



- Accessed via a pointer to the root node

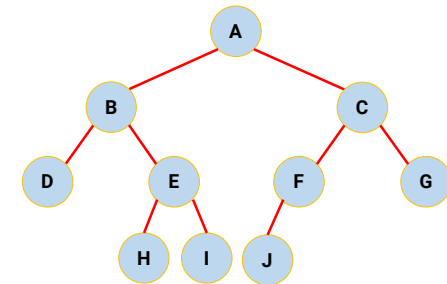


69

69

## Traverse Binary Tree

- In-order
- Pre-order
- Post-order



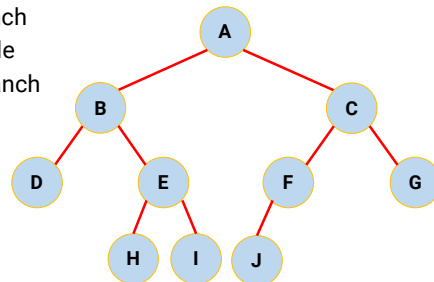
70

70

## Traverse Binary Tree

### • In-order

- Visit the left branch
- Visit the root node
- Visit the right branch



D → B → H → E → I → A → J → F → C → G

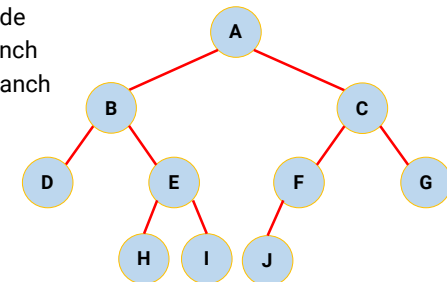
71

71

## Traverse Binary Tree (cont.)

### • Pre-order

- Visit the root node
- Visit the left branch
- Visit the right branch



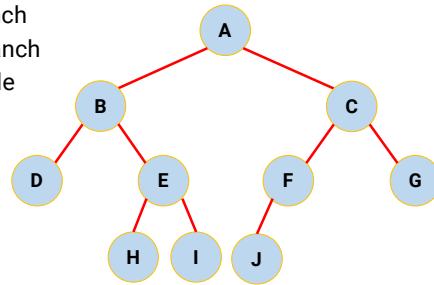
A → B → D → E → H → I → C → F → J → G

72

72

## Traverse Binary Tree (cont.)

- Post-order
  - Visit the left branch
  - Visit the right branch
  - Visit the root node



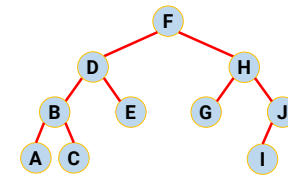
D → H → I → E → B → J → F → G → C → A

73

73

## Binary Search Tree (BST)

- A binary tree
- Each node has a (key, value) pair
- For every node  $x$ , all keys in the left subtree of  $x$  are smaller than that in  $x$
- For every node  $x$ , all keys in the right subtree of  $x$  are greater than that in  $x$
- Operations
  - Traversal
  - Search
  - Insertion
  - Deletion

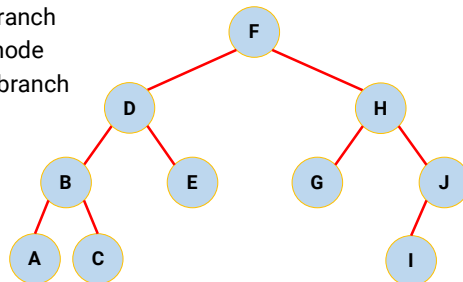


74

74

## Traverse Binary Search Tree

- In-order
  - Visit the left branch
  - Visit the root node
  - Visit the right branch



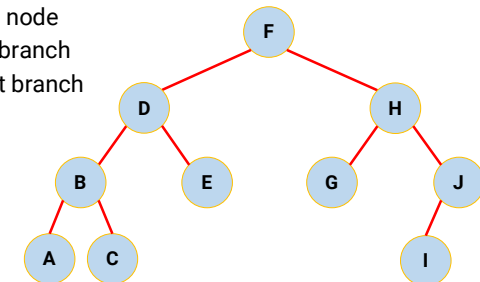
A → B → C → D → E → F → G → H → I → J

75

75

## Traverse Binary Search Tree (cont.)

- Pre-order
  - Visit the root node
  - Visit the left branch
  - Visit the right branch



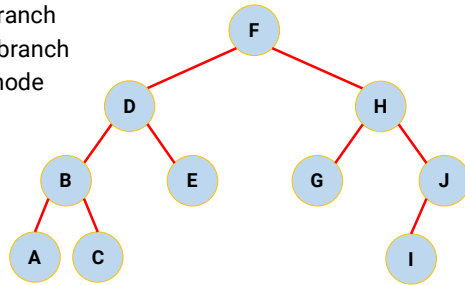
F → D → B → A → C → E → H → G → J → I

76

76

## Traverse Binary Search Tree (cont.)

- Post-order
  - Visit the left branch
  - Visit the right branch
  - Visit the root node



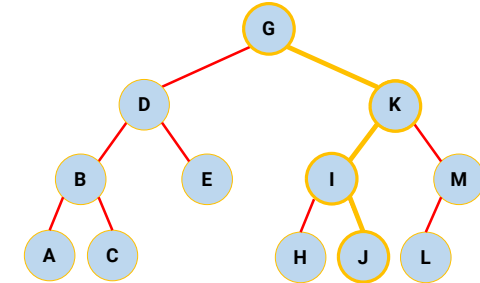
A → C → B → E → D → G → I → J → H → F

77

77

## Search Binary Search Tree

- Similar to binary search (but may not be half-half)
- Example: find J

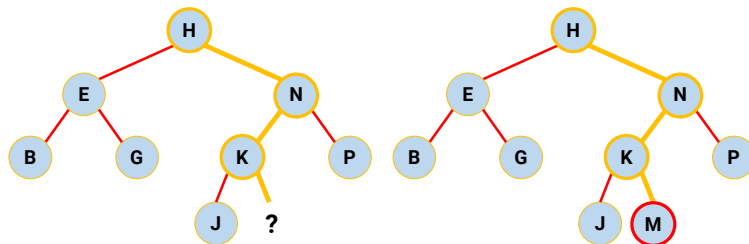


78

78

## Insertion in Binary Search Tree

- First search (e.g., M) for the new entry until its absence is detected
- This is the position for the insertion

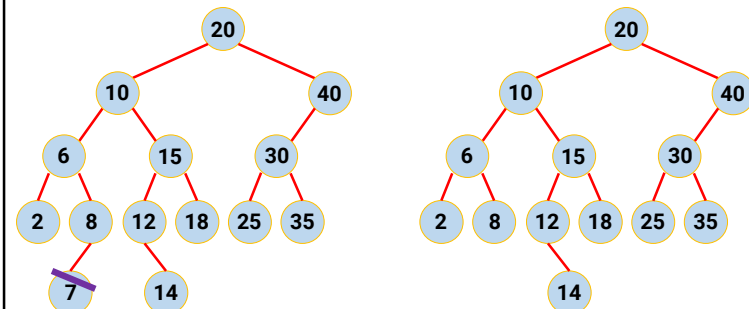


79

79

## Deletion in Binary Search Tree

- Erase a leaf **element** whose key is 7

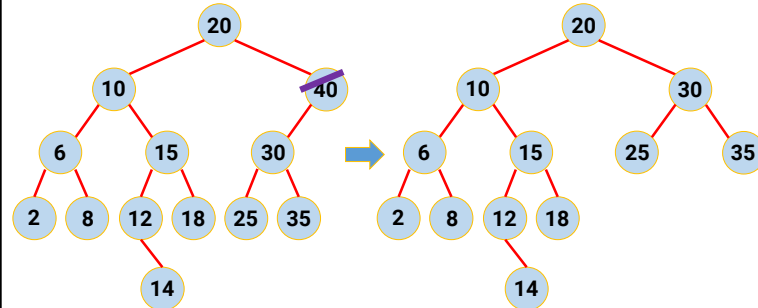


80

80

### Deletion in Binary Search Tree (cont.)

- Erase a degree-1 node whose key is 40

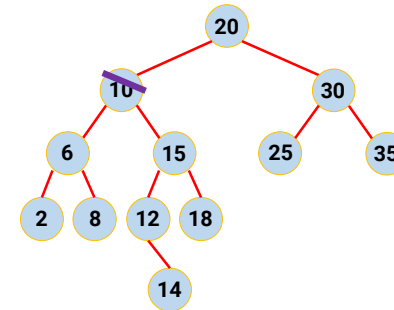


81

81

### Deletion in Binary Search Tree (cont.)

- Erase a degree-2 node whose key is 10

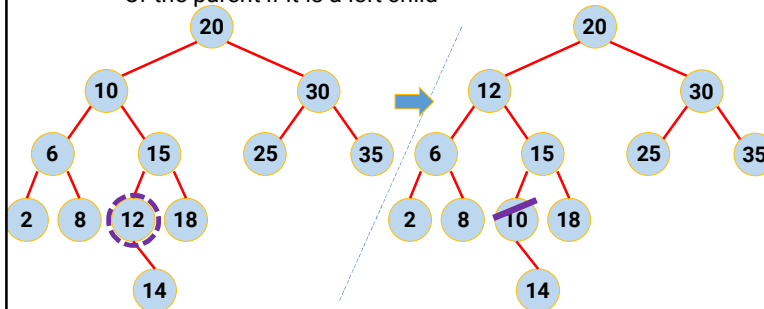


82

82

### Deletion in Binary Search Tree (cont.)

- Swap its with its successor
  - The minimum node of the right subtree (keep going left)
  - Or the parent if it is a left child

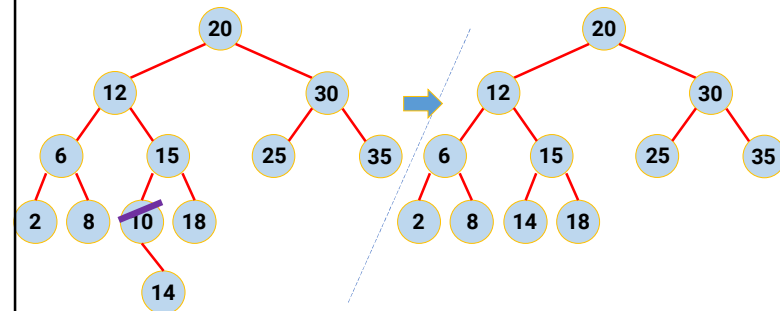


83

83

### Deletion in Binary Search Tree (cont.)

- Since Its successor has a degree of 1 or 0, we can simply cut and reconnect the rest of the tree



84

84

Introduction to Computer 2022

**Any Questions?**

85

85