Data Abstractions

Introduction to Computer

Yu-Ting Wu

Outline

- Arrays

- Lists

- Stacks

- Queues

- Trees

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

Outline

- Arrays

- Lists

- Stacks

- Queues

- Trees

Homogeneous Arrays

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

```
int scores[5];
scores[0] = 100;
scores[1] = 80;
scores[2] = 70;
scores[3] = 90;
scores[4] = 60;
```

```
int* ptrScores = scores;
std::cout << ptrScores << std::endl;              ──→ 00000001000FFE60
std::cout << ptrScores + 1 << std::endl;          ──→ 00000001000FFE64
std::cout << ptrScores + 2 << std::endl;          ──→ 00000001000FFE68
std::cout << std::endl;
std::cout << ptrScores << std::endl;              ──→ 00000001000FFE60
std::cout << &(scores[0]) << std::endl;           ──→ 00000001000FFE60
std::cout << &(scores[1]) << std::endl;           ──→ 00000001000FFE64
std::cout << &(scores[2]) << std::endl;           ──→ 00000001000FFE68
```

address 61270FF990   61270FF994   (sizeof(int) = 4)

memory
cell   • • • | 100 | 80 | 70 | 90 | 60 | • • •

scores[0]   scores[2]   scores[4]

scores[1]   scores[3]
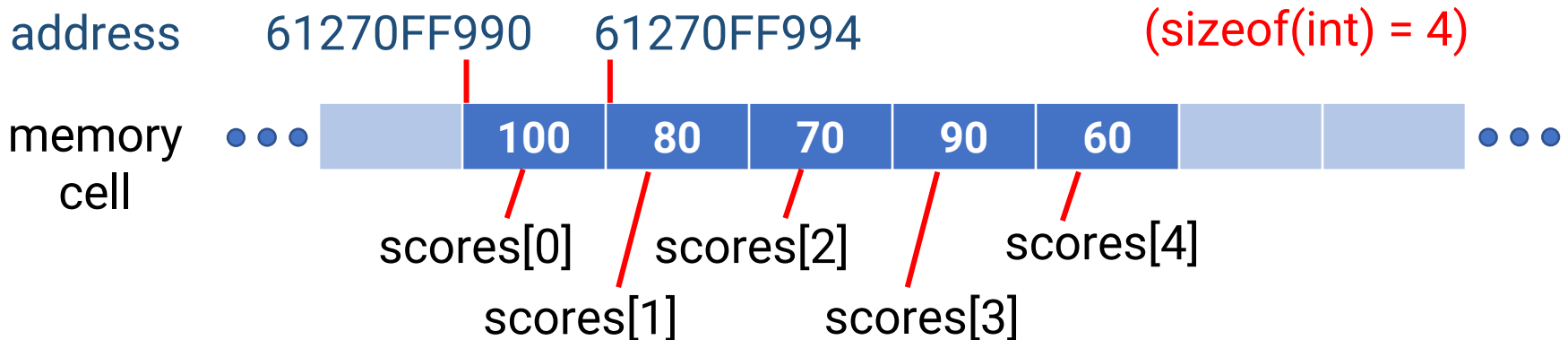
Homogeneous Arrays (cont.)

- 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]

image[2][1] (row2, column1)
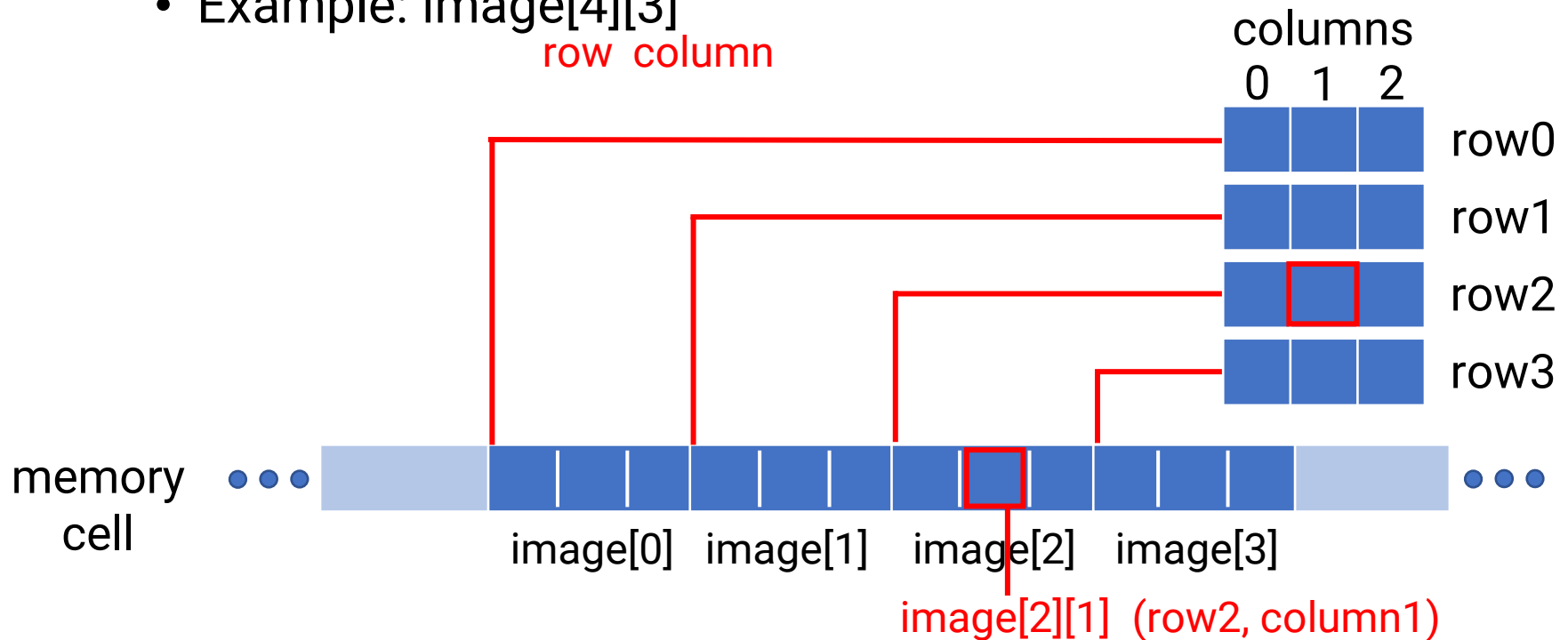
Homogeneous Arrays (cont.)

- **Array addresses**

x = 1000

x[0] = 1000

x[1] = 1016

x[2] = 1032

x[0][0] 1000	x[0][1] 1004	x[0][2] 1008	x[0][3] 1012
x[1][0] 1016	x[1][1] 1020	x[1][2] 1024	x[1][3] 1028
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** = 1016 + **2** * **4** = 1024

x[**1**]

Homogeneous Arrays (cont.)

- **Parameter passing**
  - Does it work?

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

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

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

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

abc	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

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;
```

```
0000007AF10FFAD0
0000007AF10FFAD0
0000007AF10FFAE4
0000007AF10FFAF8
0000007AF10FFAFC
```

student
student.name

student.ICMidtermScore

student.department   student.grade

7AF10FFAD0   7AF10FFAE4   7AF10FFAF8

7AF10FFAFC
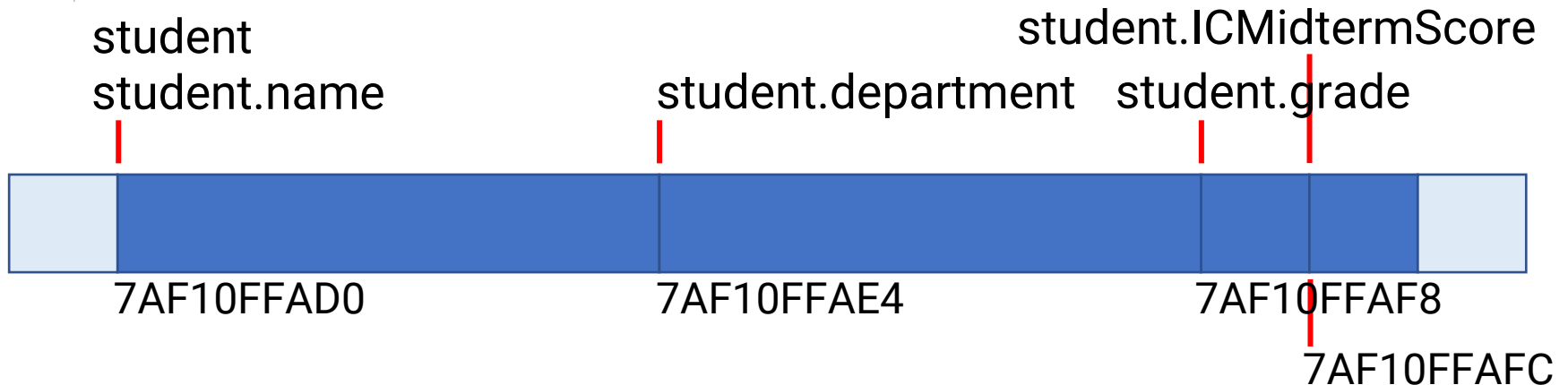
Outline

- Arrays

- Lists
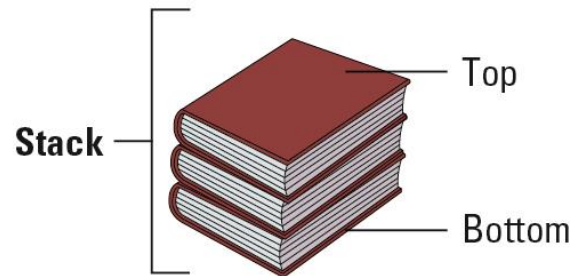
- Stacks

- Queues

- Trees

Lists, Stacks, and Queues

- List: a collection of data whose entries are arranged sequentially

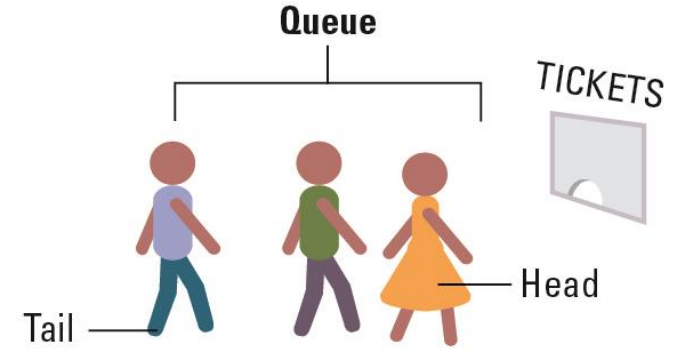- **Stacks** and **Queues** are specialized lists

List	**Stack**	**Queue**
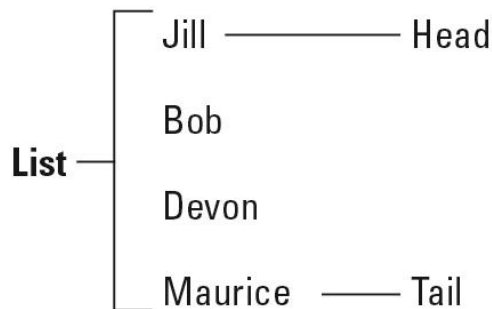
a. A list of names

b. A stack of books

c. A queue of people

First-In-Last-Out   **First-In-First-Out**

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

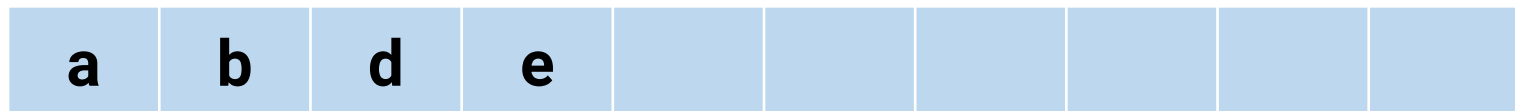# Storing Lists

- **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

a	b	c	d	e					

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

a	b		d	e					

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

a	b	d	e						

Storing Lists (cont.)

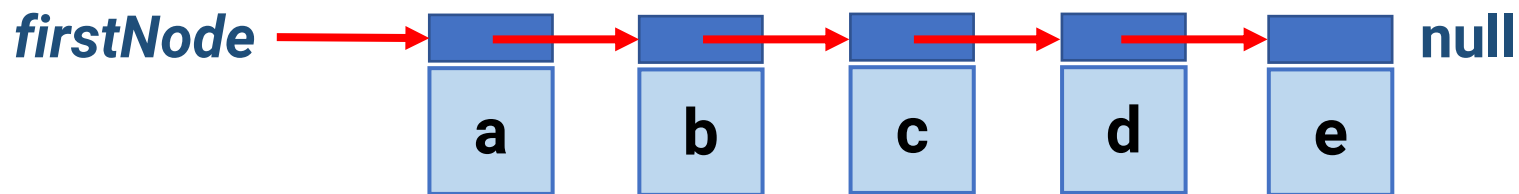- **Linked list**
    - Head pointer: indicate the start (*firstNode*)
    - NULL pointer: indicate the end
    - Example: *L = (a, b, c, d, e)* using a linked representation
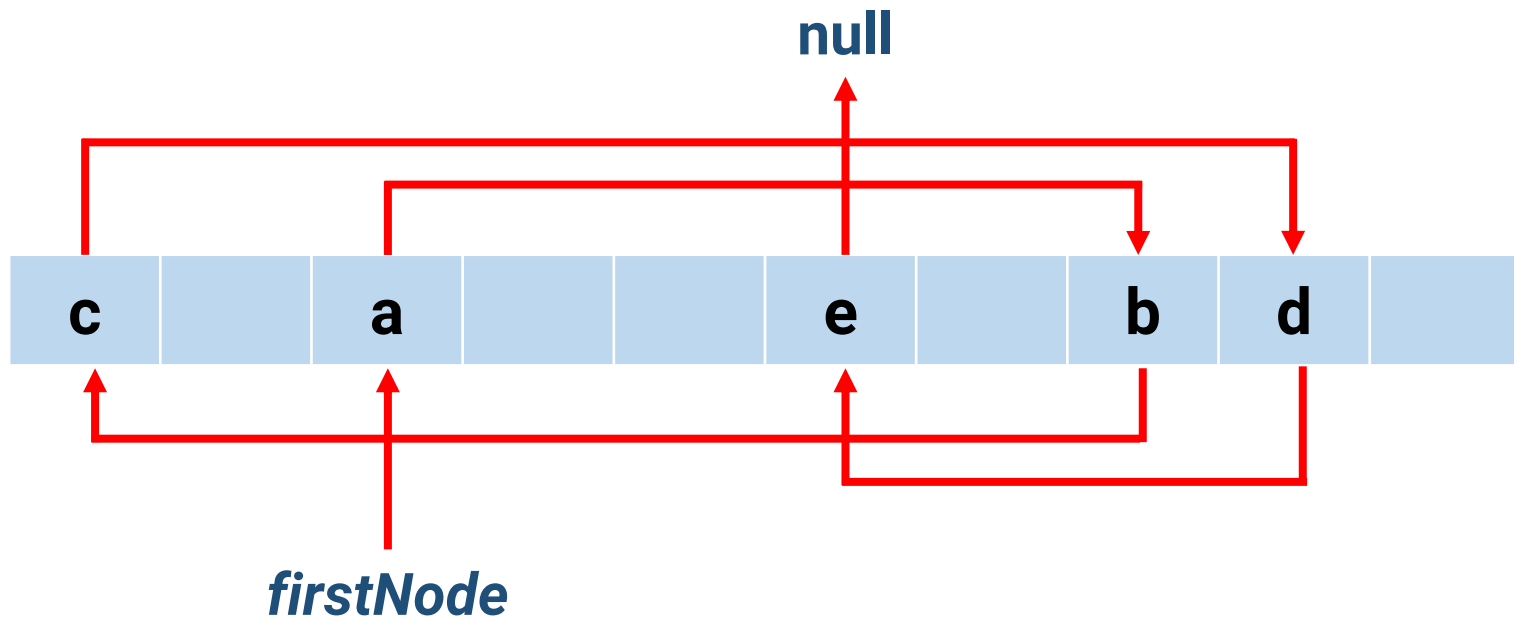
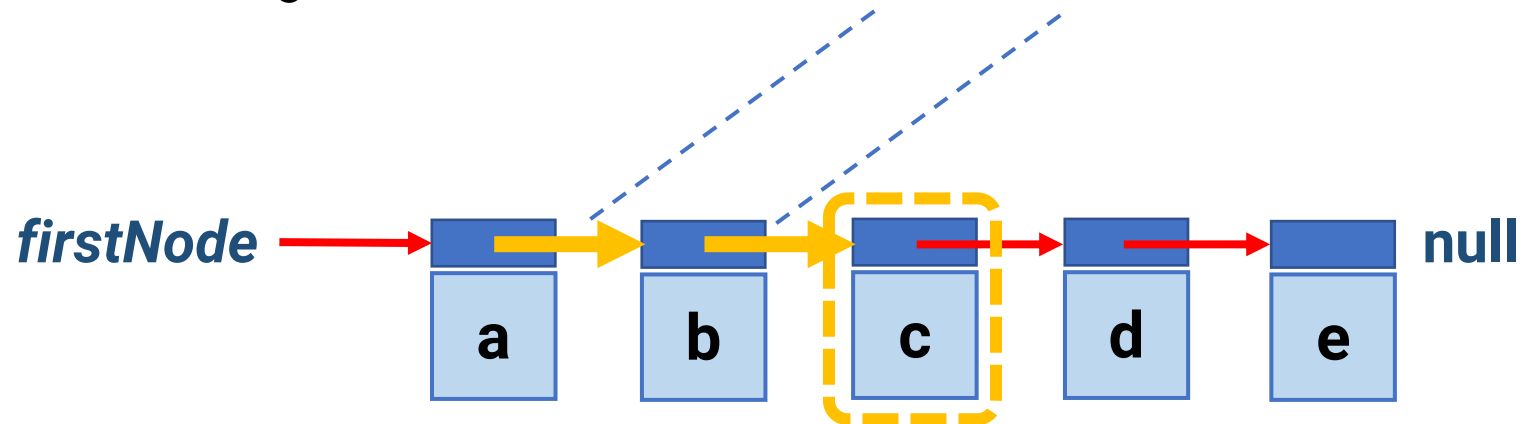# Storing Lists (cont.)

- **Linked list**
  - Head pointer: indicate the start
  - NULL pointer: indicate the end
  - Example: *L = (a, b, c, d, e)* using a linked representation
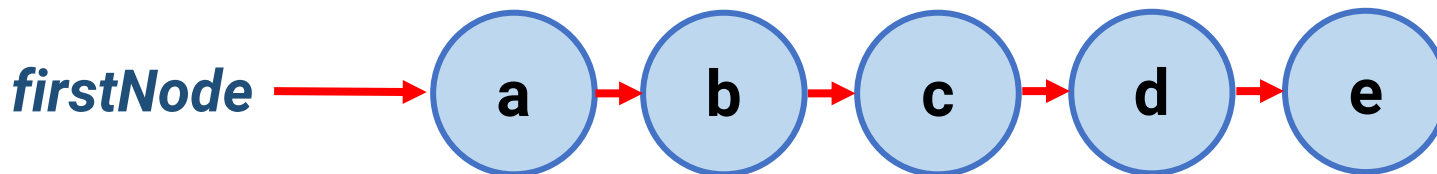
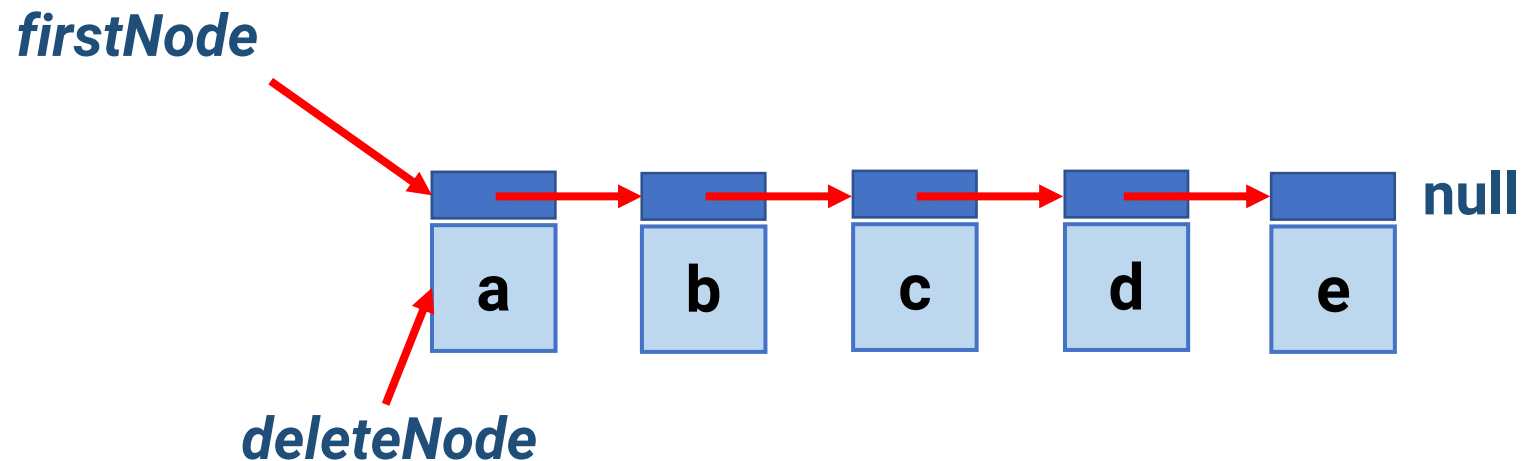# List: Get an Element with its Index

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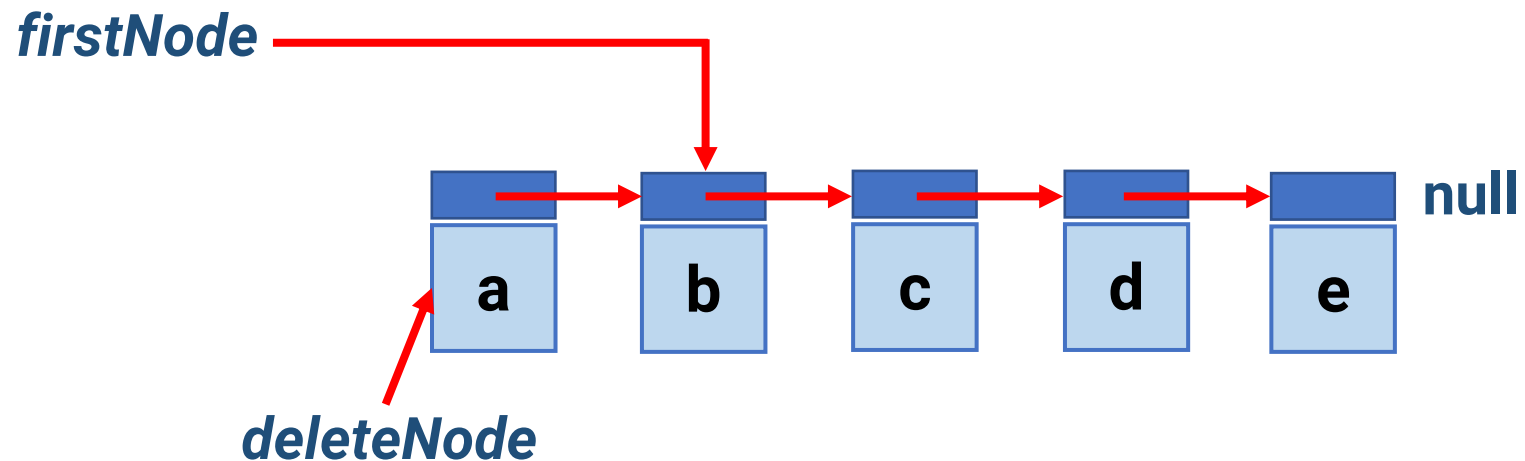   - Target node = *firstNode→next→next*;

firstNode → a → b → c → d → e → null

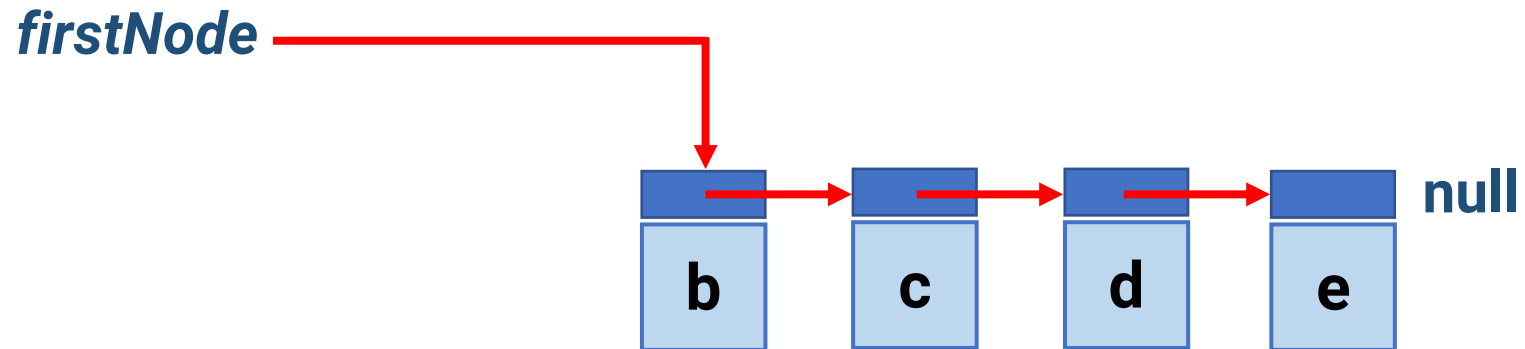# List: Erase an Element with its Index

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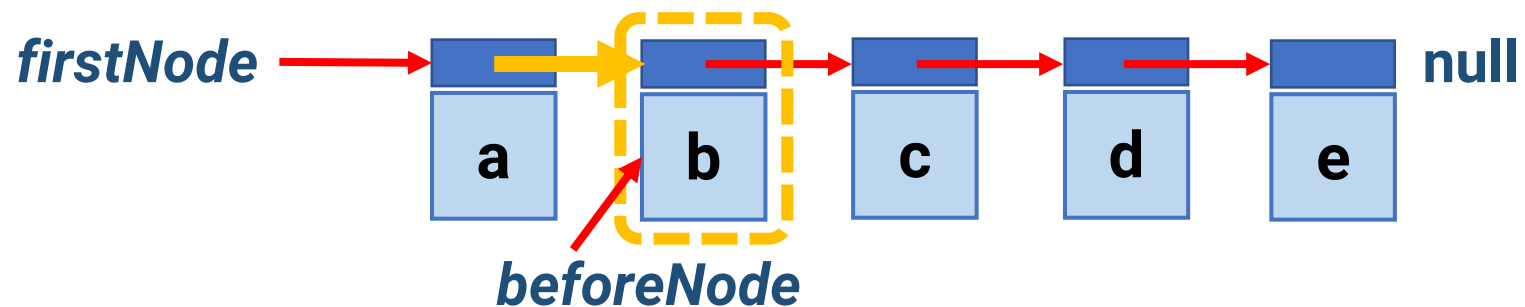
firstNode

deleteNode

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

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*

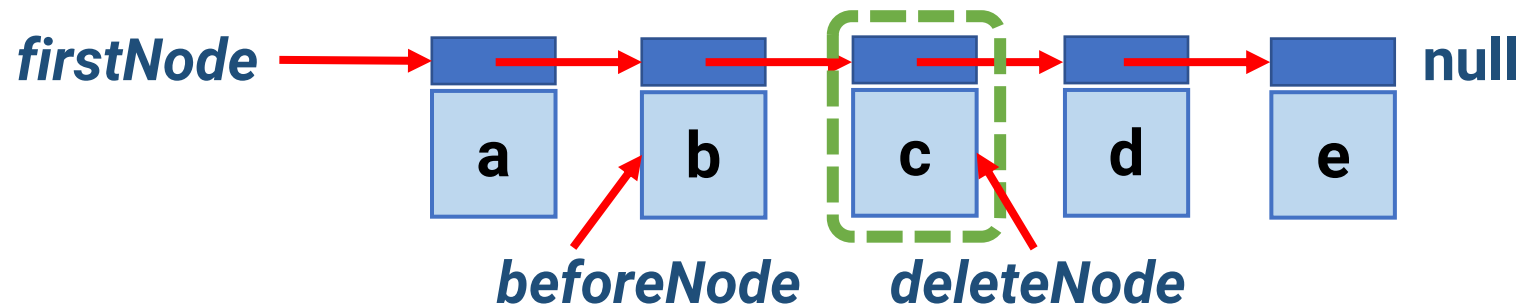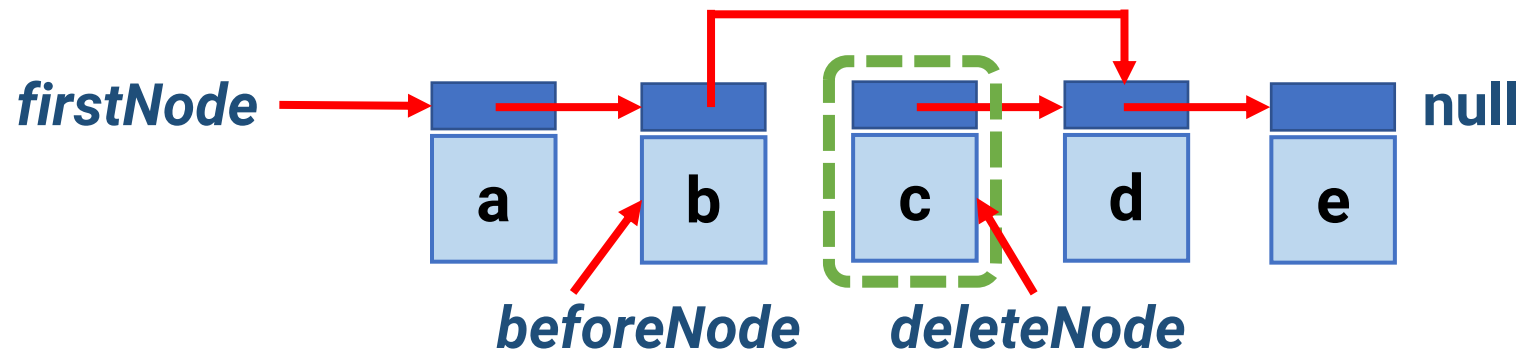*firstNode*

b   c   d   e   **null**

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)   **assume the index starts with 0**

firstNode → a b c d e **null**

beforeNode

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)

firstNode → a → b → c → d → e → **null**

beforeNode   *deleteNode*

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)

firstNode → a → b → c → d → e **null**

beforeNode   *deleteNode*

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)
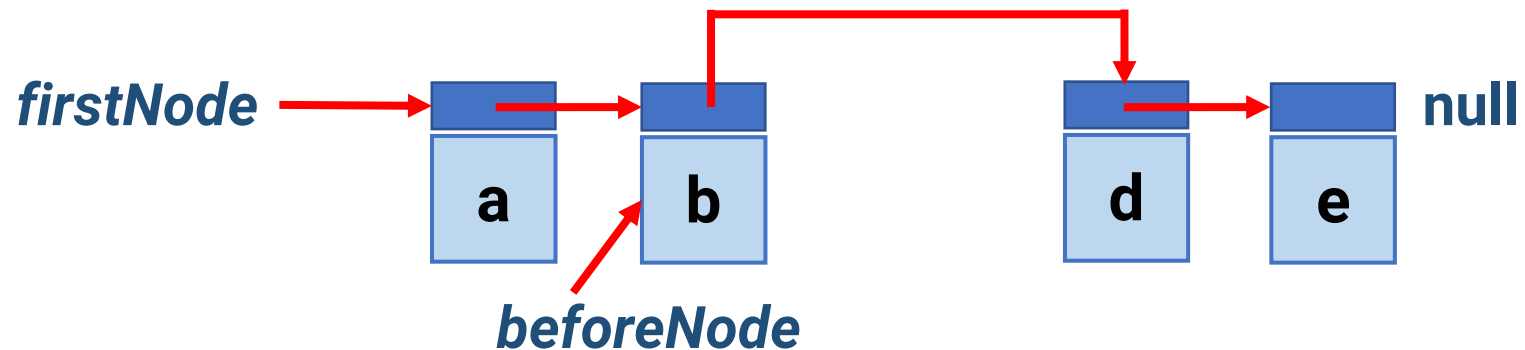
List: Insert a New Element

- Case 1: insert at front

- **Procedure**
    - Create a new node with the given data

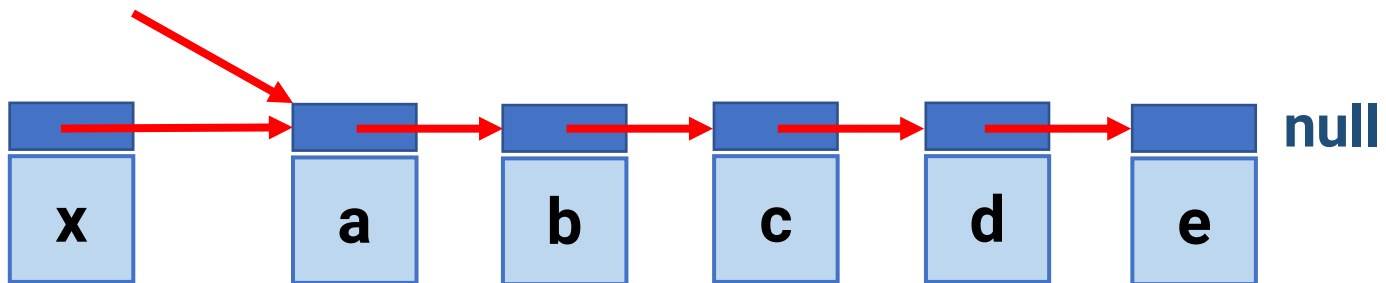- Example: Insert(0, x)
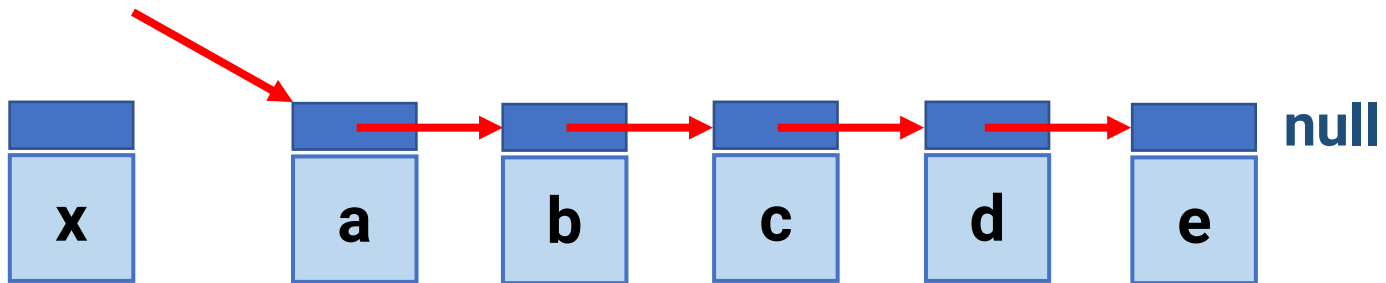
firstNode

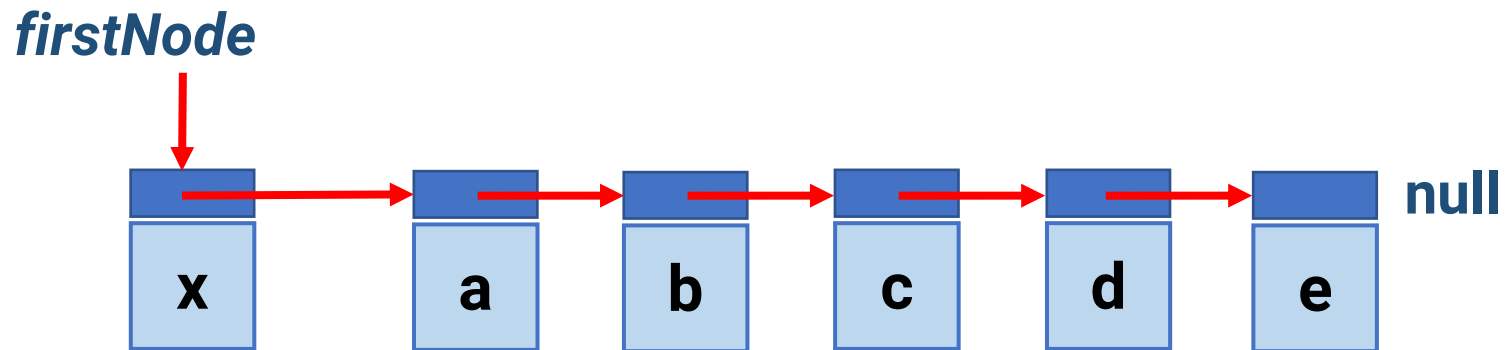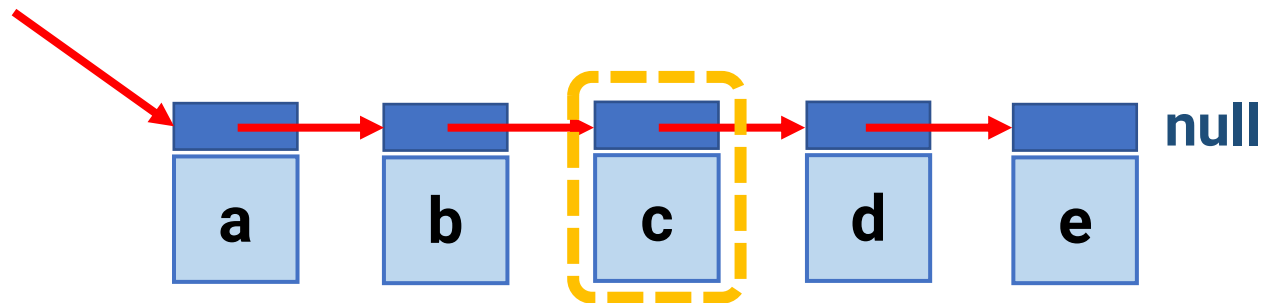# List: Insert a New Element (cont.)

- 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)

firstNode

null

x   a   b   c   d   e

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)

firstNode

x → a → b → c → d → e → null

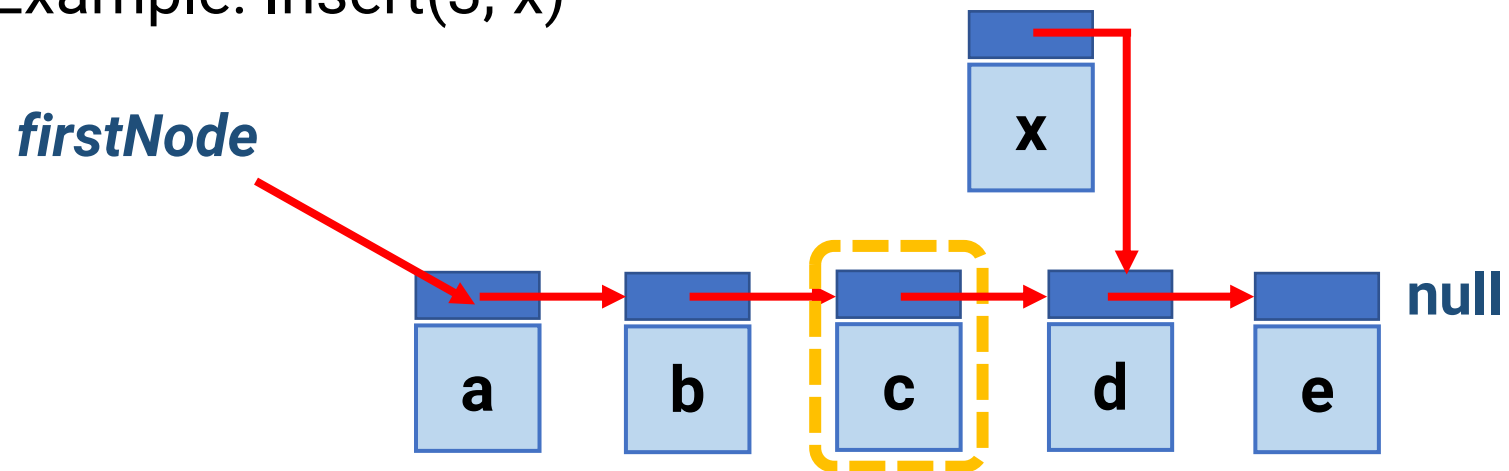# List: Insert a New Element (cont.)

- Case 2: insert in the middle

- **Procedure**
    - Find the node before the target (*beforeNode*)

- Example: Insert(3, x)   **assume the index starts with 0**

List: Insert a New Element (cont.)

- Case 2: insert in the middle

- **Procedure**
    - Find t he node before the target (*beforeNode*)
    - Create a new node with the given data and set its pointer
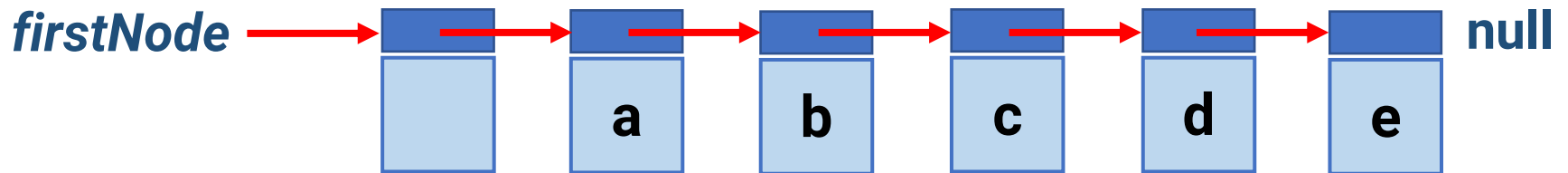
- Example: Insert(3, x)

List: Insert a New Element (cont.)

- Case 2: insert in the middle

- **Procedure**
  - Find t he 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
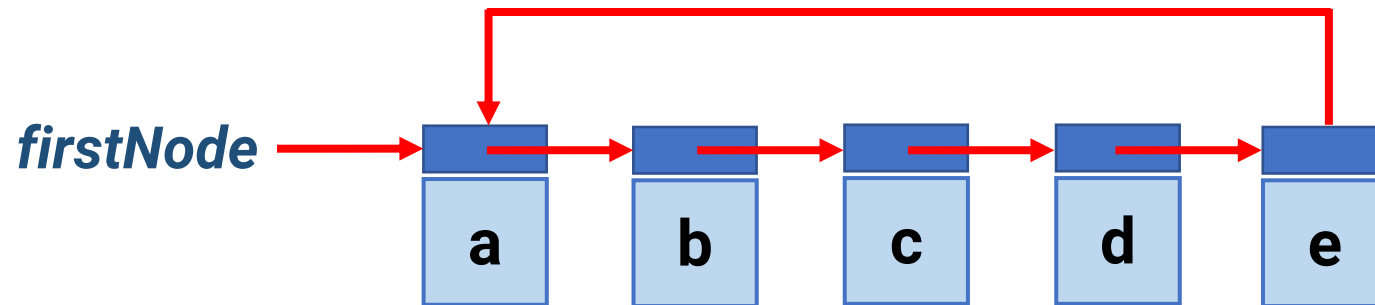
- Example: Insert(3, x)
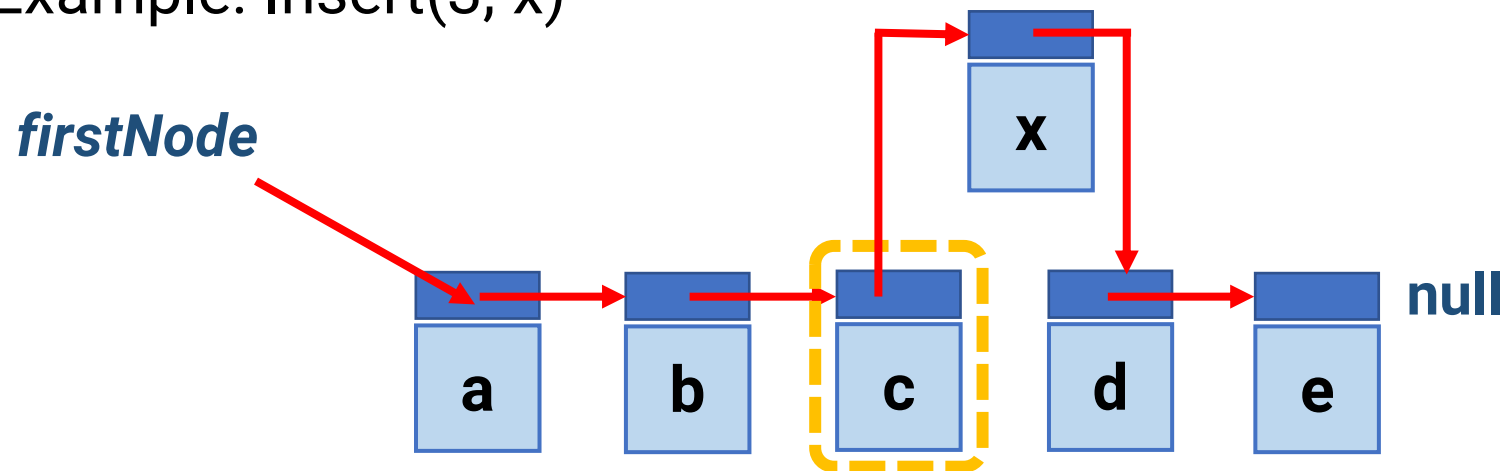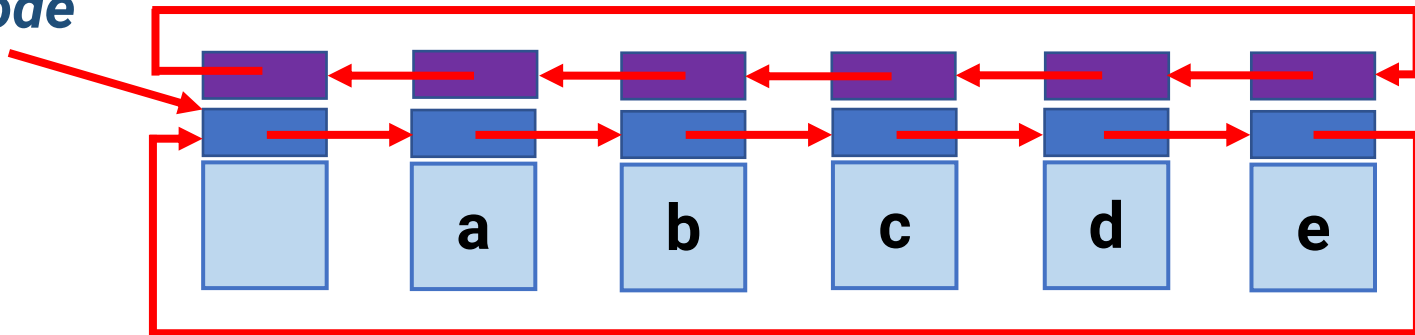
Variations

- List with a **dummy** header node

firstNode → [ ] → [a] → [b] → [c] → [d] → [e] → **null**

- Circular list

firstNode → [a] → [b] → [c] → [d] → [e]

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

firstNode

Outline

- Arrays

- Lists

- Stacks

- Queues

- Trees

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

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
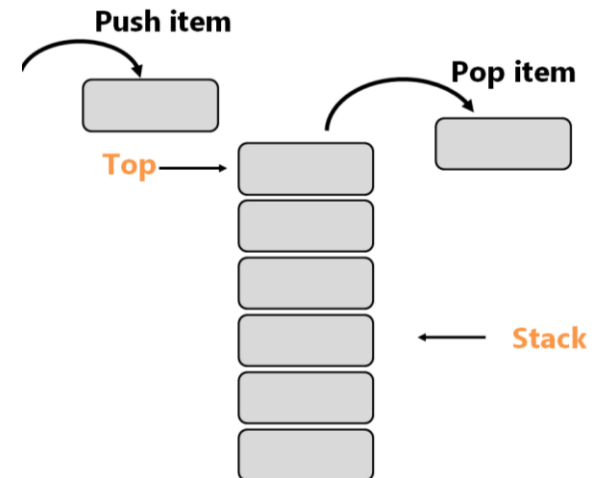
Storing Stacks

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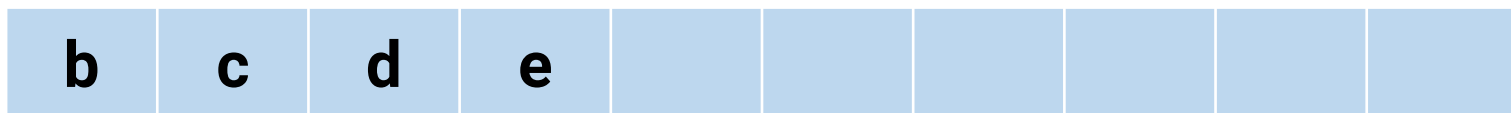 - When the top is left end

a	b	c	d	e					

stack top

  - Push: Θ(n)
    - Need to move all elements right

x	a	b	c	d	e				

  - Pop: Θ(n)
    - Need to move all elements left

b	c	d	e						

Storing Stacks (cont.)

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

| a | b | c | d | e | | | | | |

stack top

  - Push: Θ(1)
    - **No** need to move all elements right

| a | b | c | d | e | X | | | | |

  - Pop: Θ(1)
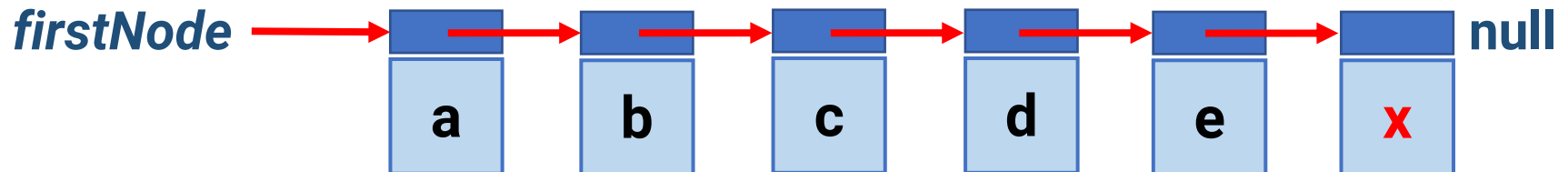   - **No** need to move all elements left

| a | b | c | d | | | | | | |

Storing Stacks (cont.)

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

stack top

firstNode → a → b → c → d → e null

- Push: Θ(n)
    - Need to traverse all nodes

firstNode → a → b → c → d → e → x null

Storing Stacks (cont.)

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

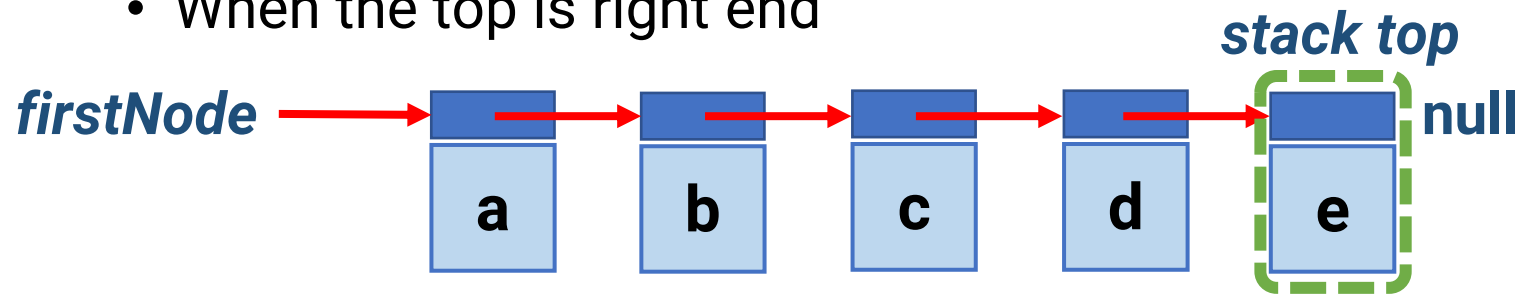*stack top*

firstNode → [a] → [b] → [c] → [d] → [e] null

  - Pop: Θ(n)
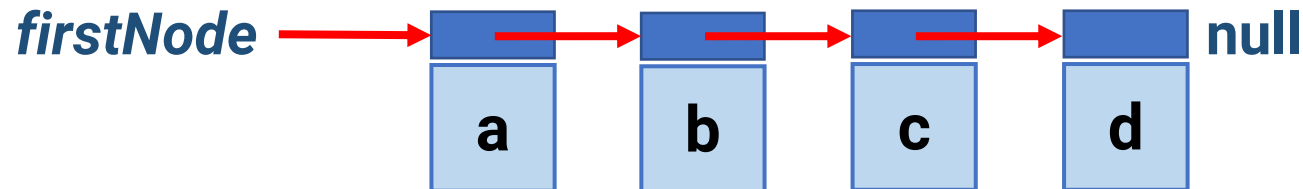   - Need to traverse all nodes

firstNode → [a] → [b] → [c] → [d] null

Storing Stacks (cont.)

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

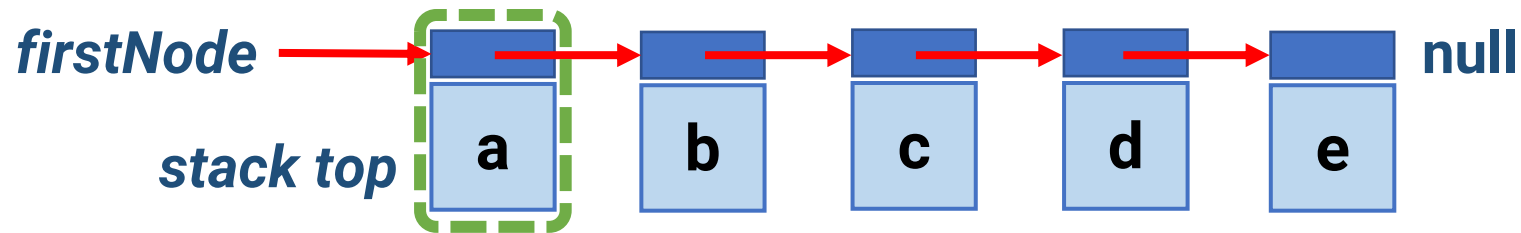*firstNode* → a → b → c → d → e **null**
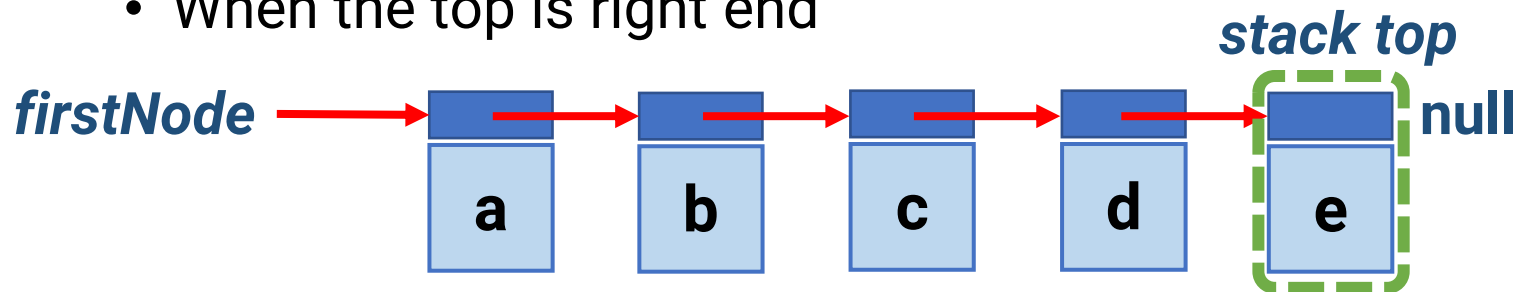
stack top

- Push: Θ(1)
    - Insert at front
- Pop: Θ(1)
    - EraseElement at front

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

- 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), (missing, 5), (8, 12), (7, missing)

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
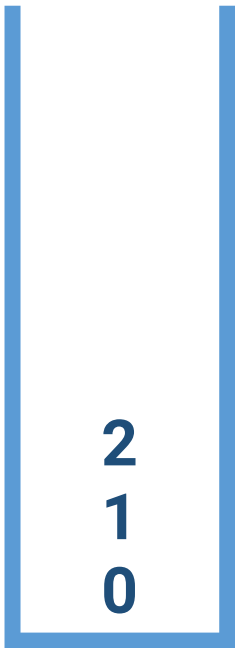
Parentheses Matching (cont.)

(	(	(	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

2
1
0

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)

1
0

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)

0

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

15
0

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

21

0

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)
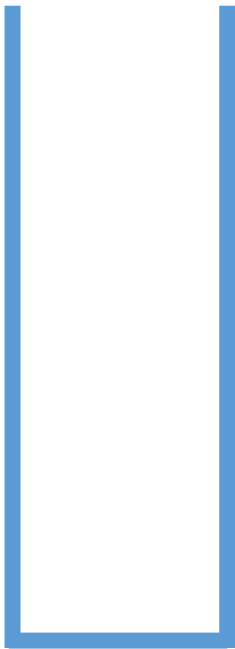
Parentheses Matching (cont.)

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

Actions

Push 0

0
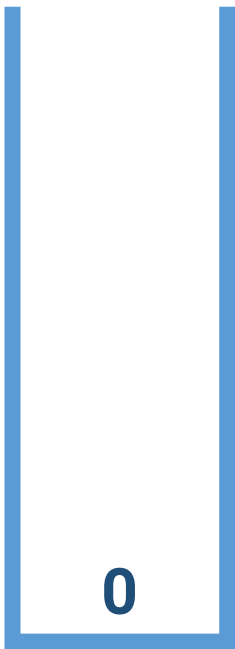
Parentheses Matching (cont.)

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

Actions
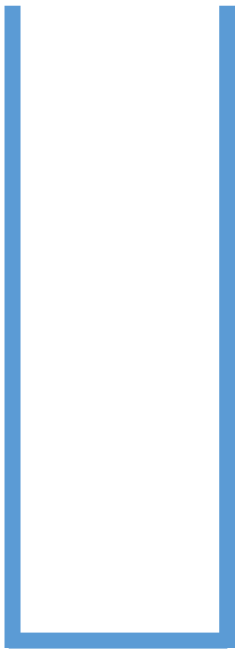
Push 0
Pop, output (0, 4)

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!

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

8
7
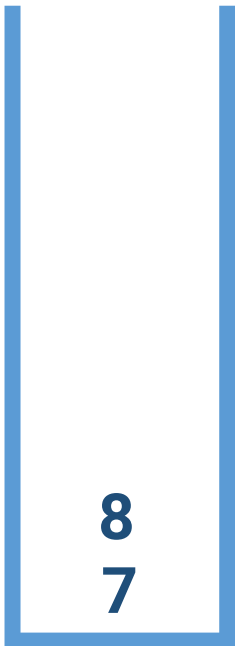
Parentheses Matching (cont.)

(	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

7
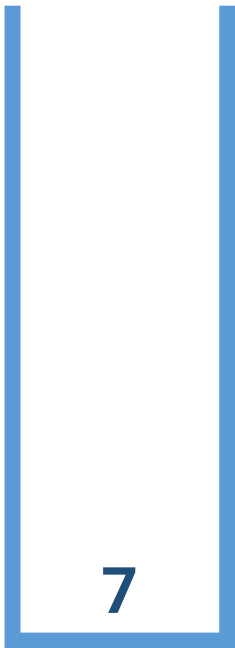
Outline

- Arrays

- Lists

- Stacks

- **Queues**

- Trees

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

pop

Dequeue

Front / Head

Back / Tail / Rear

| 3 | 4 | 5 | 6 | 7 | 8 |

9

Enqueue

push

2

Storing Queues

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

a	b	c	d	e				

rear   *front*

- Front: Θ(1)
- Back: Θ(1)
- Pop: Θ(1)
- Push: Θ(n)

x	a	b	c	d	e			

rear   *front*
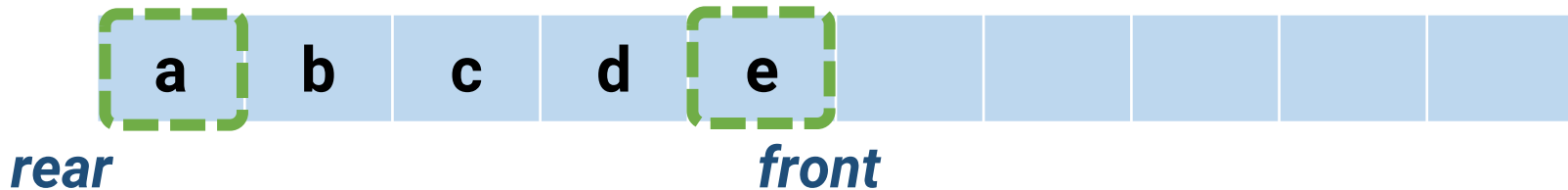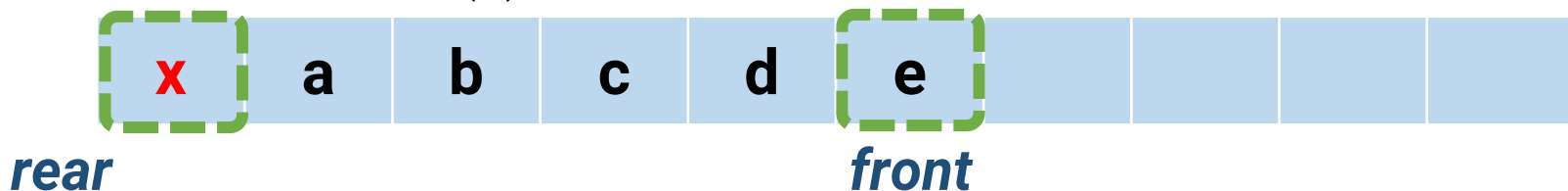
Storing Queues (cont.)

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

a	b	c	d	e				

front   *rear*

- Front: Θ(1)
- Back: Θ(1)
- Pop: Θ(n)
- Push: Θ(1)

b	c	d	e					

front   *rear*

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)
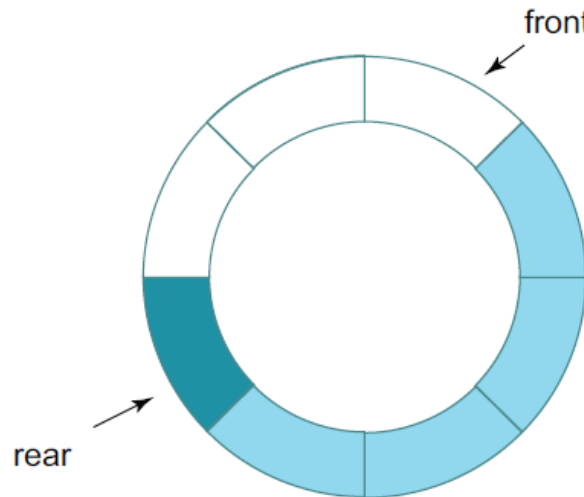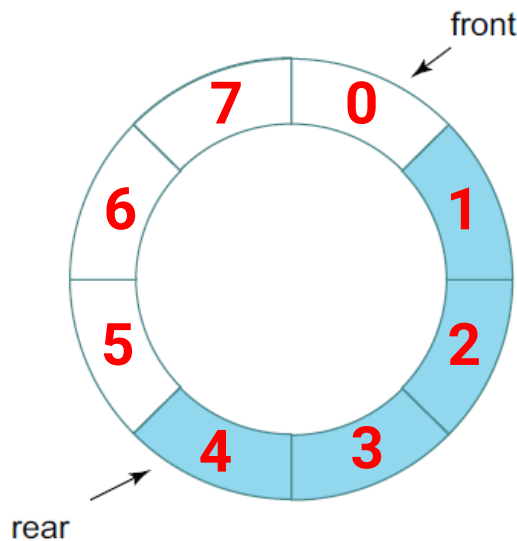
push
rear = (rear+1)%length
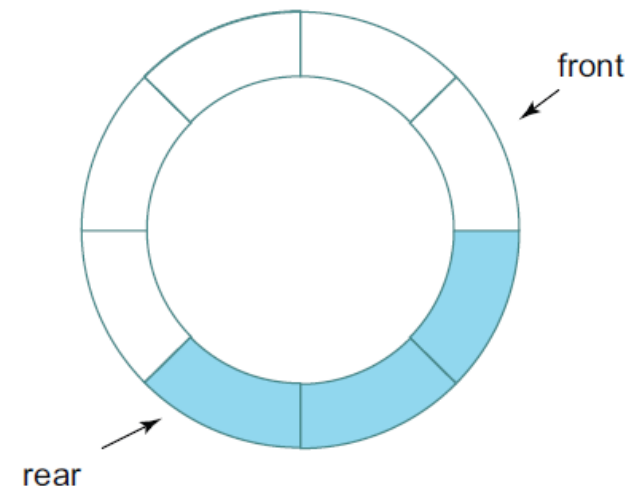
pop
front = (front+1)%length

Storing Queues (cont.)

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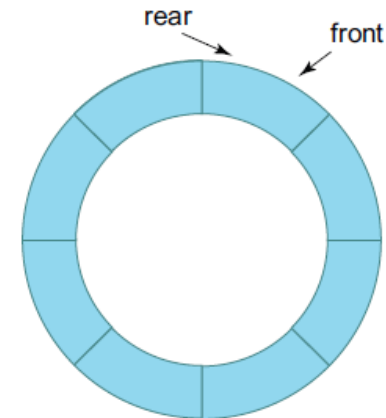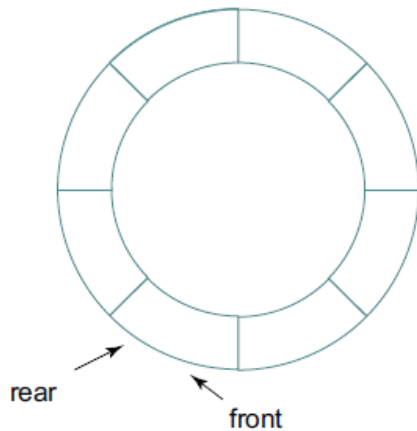
Outline

- Arrays

- Lists

- Stacks

- Queues

- Trees

Tree

- Lists are useful for serially ordered data

- Trees are useful for hierarchically ordered data

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)

Terminology for a Tree (cont.)
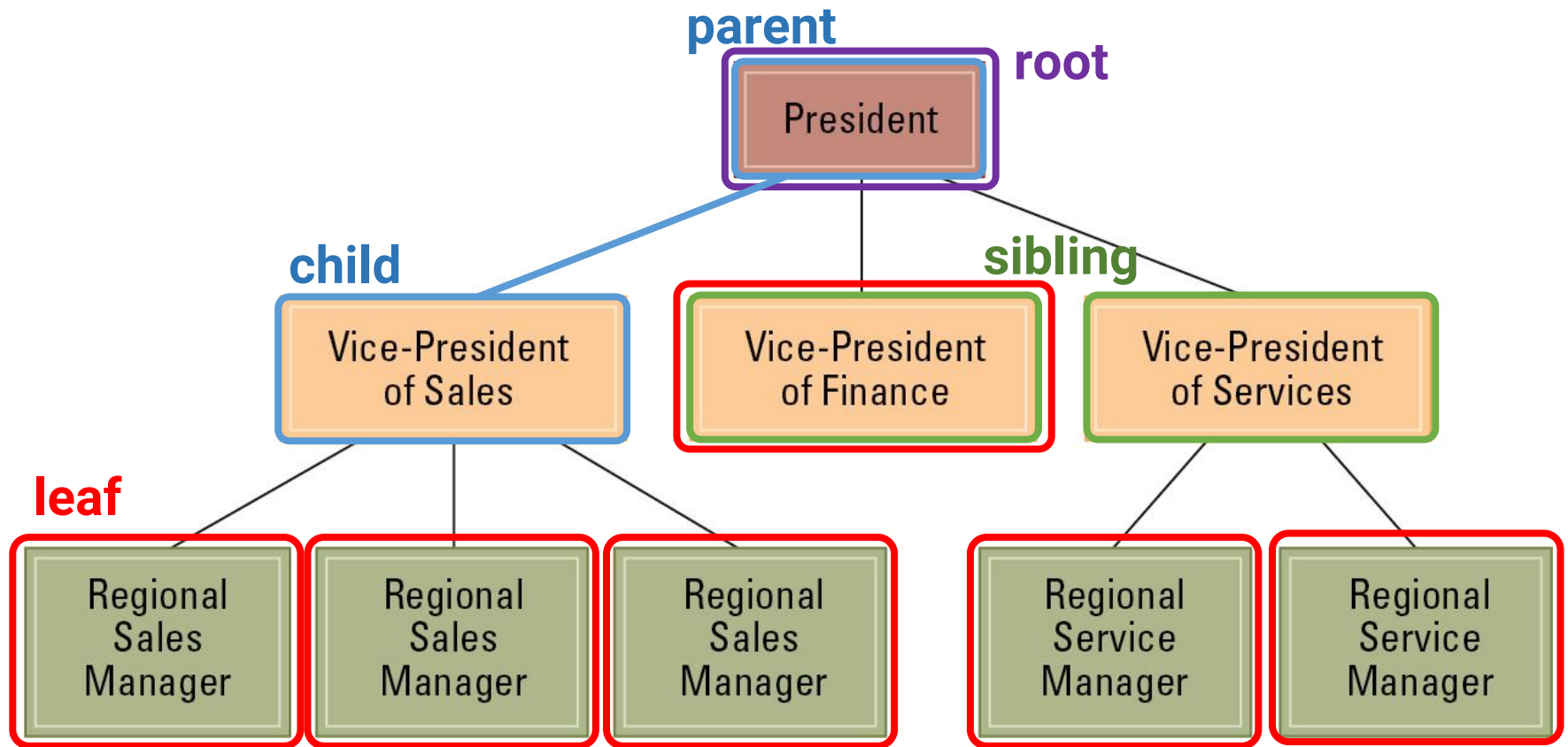
Terminology for a Tree (cont.)

- **Degree:** number of children
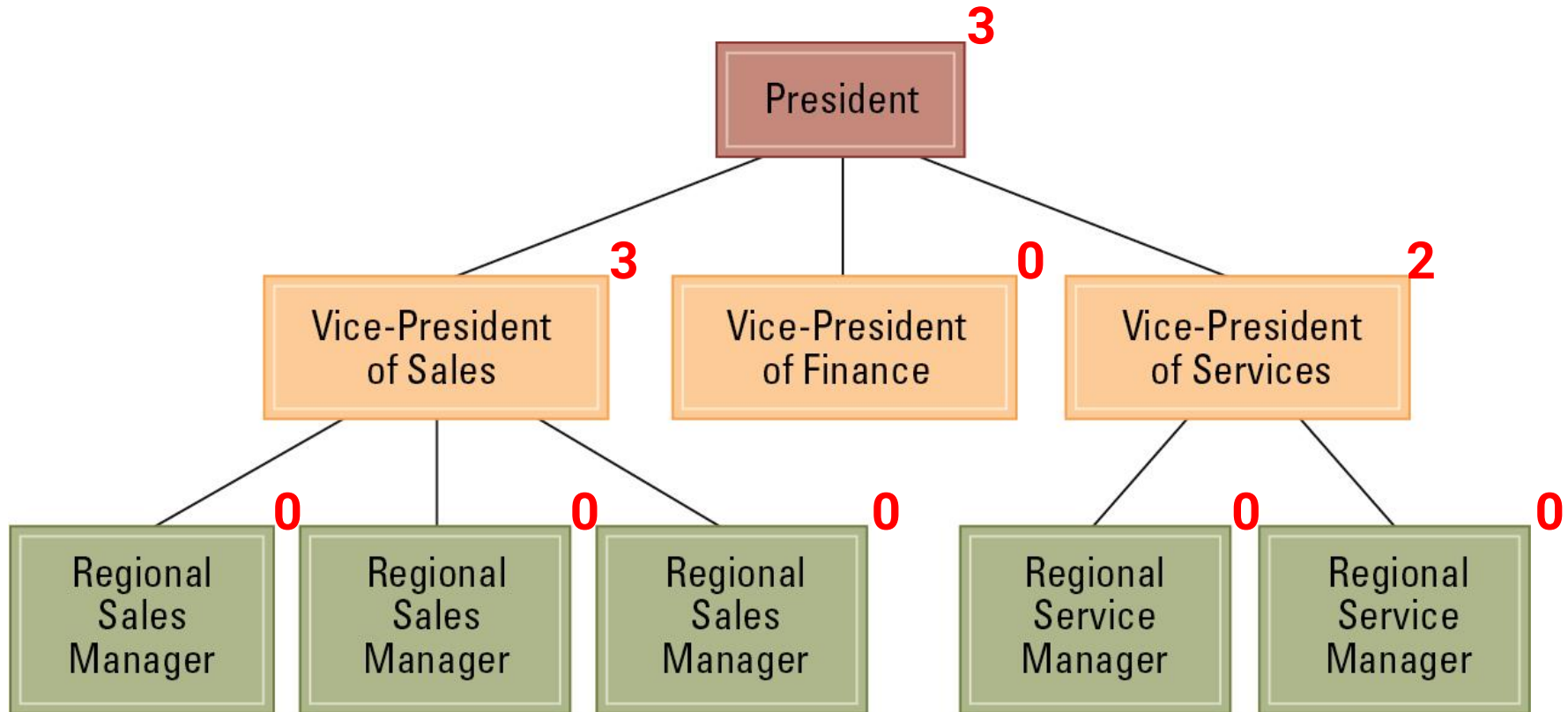
Definition of Tree

- **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*

Definition of Tree (cont.)
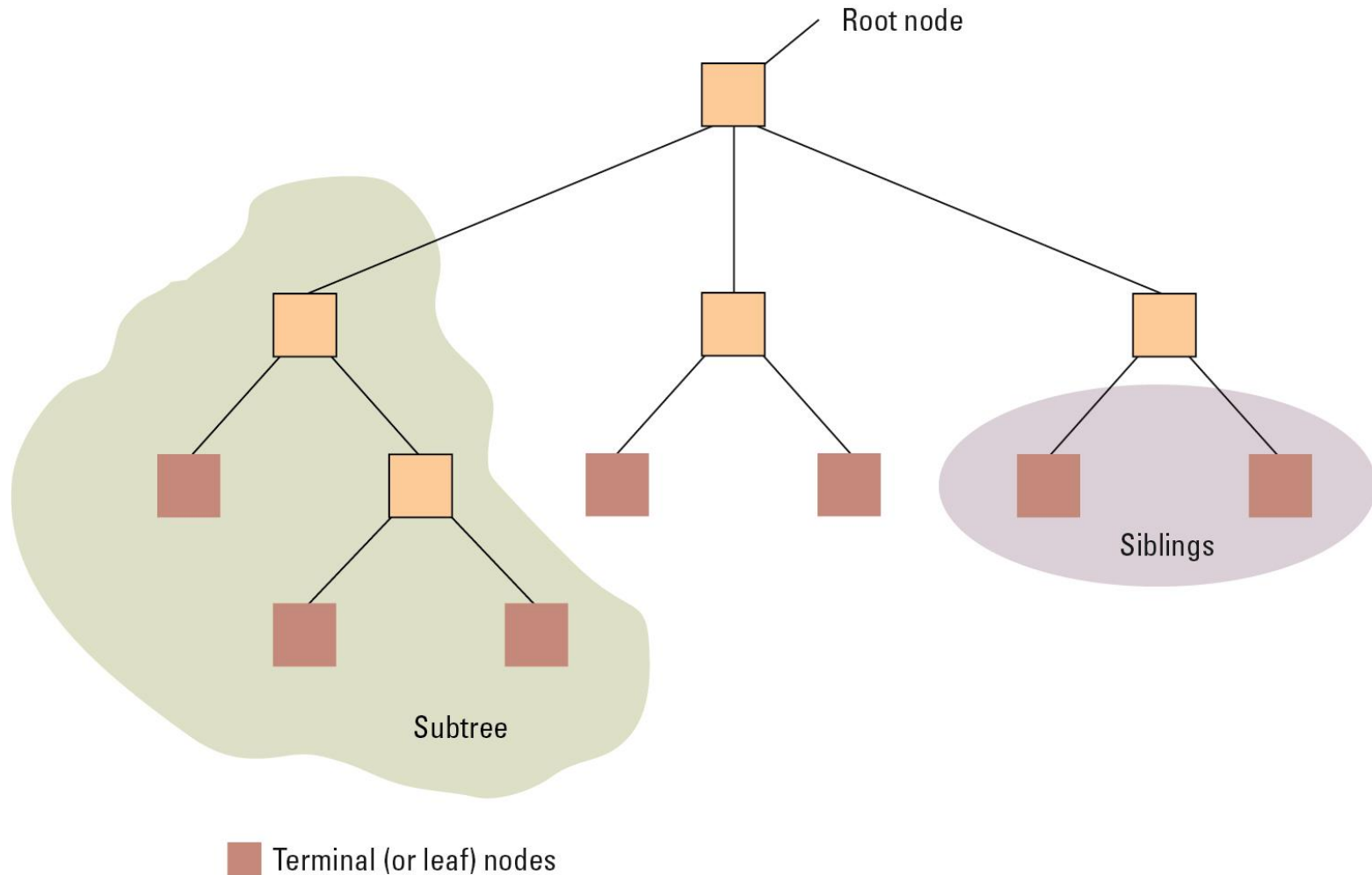
Binary Trees

- 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
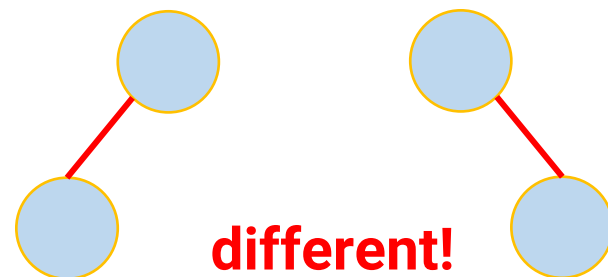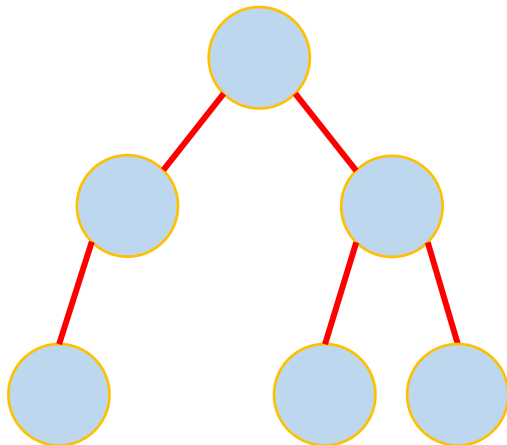
different!
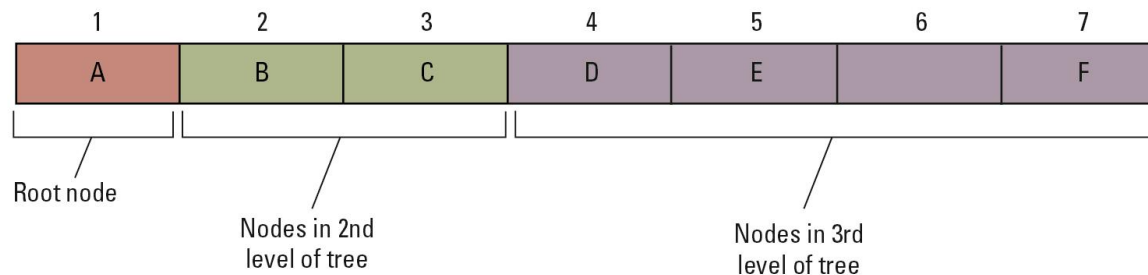
Storing Binary Trees

- **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]

Conceptual tree

Actual storage organization

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]

Conceptual tree

Actual storage organization

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C				D								E

Root | 2nd level | 3rd level | 4th level
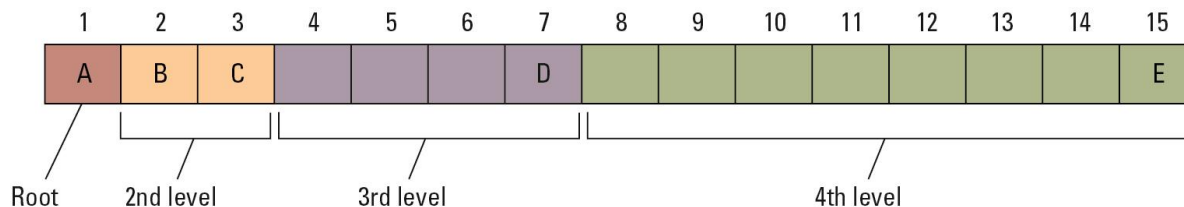
Storing Binary Trees

- **Linked structure**
  - Each node = data cells + two child pointers

  - Accessed via a pointer to the root node
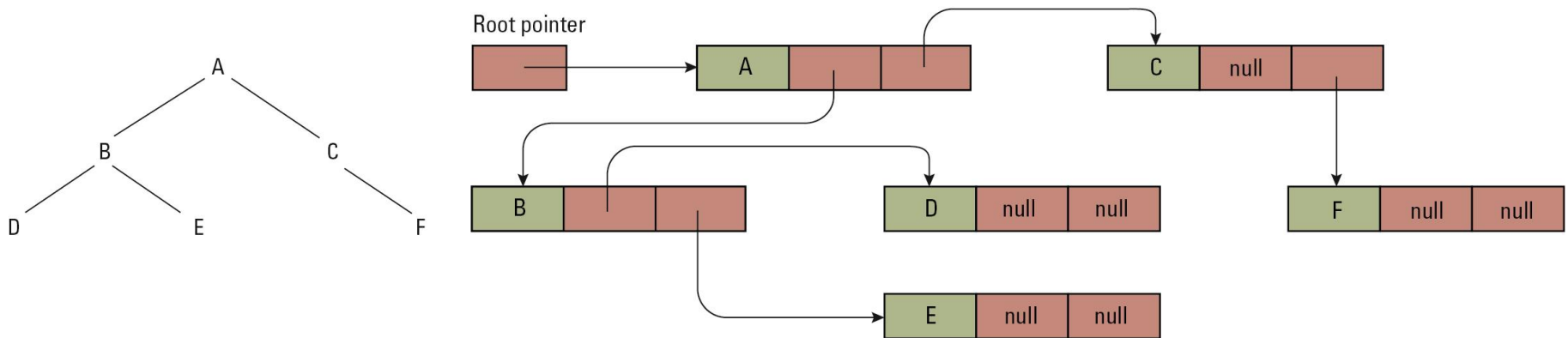
Traverse Binary Tree

- In-order
- Pre-order
- Post-order
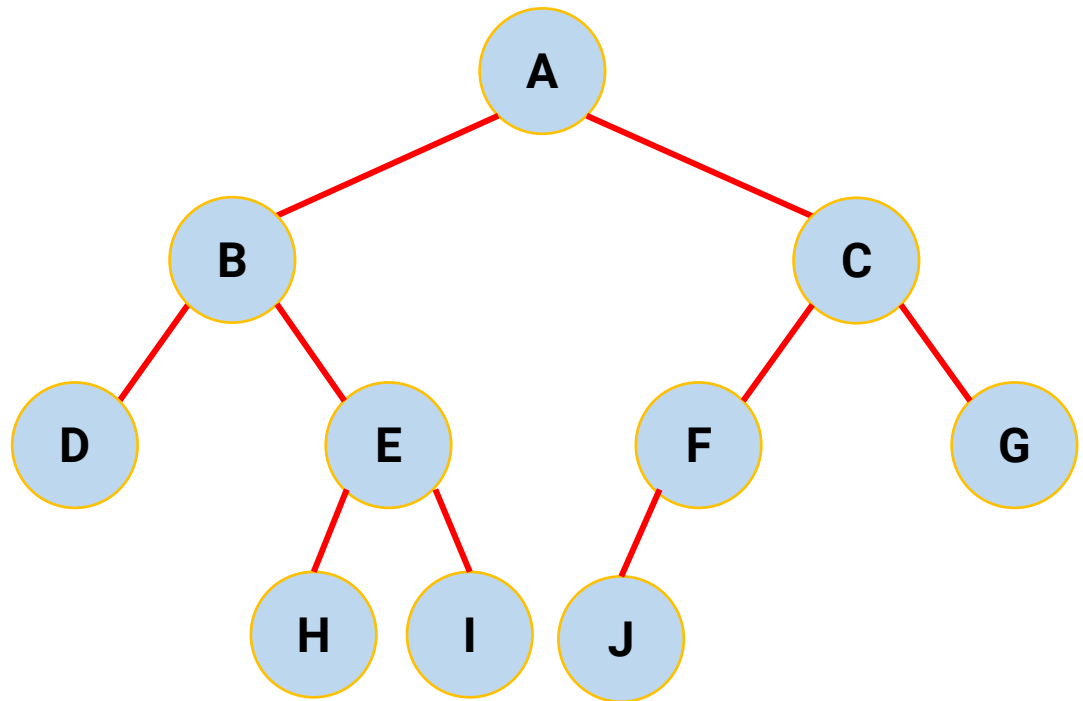
Traverse Binary Tree

- In-order
  - Visit the left branch
  - Visit the root node
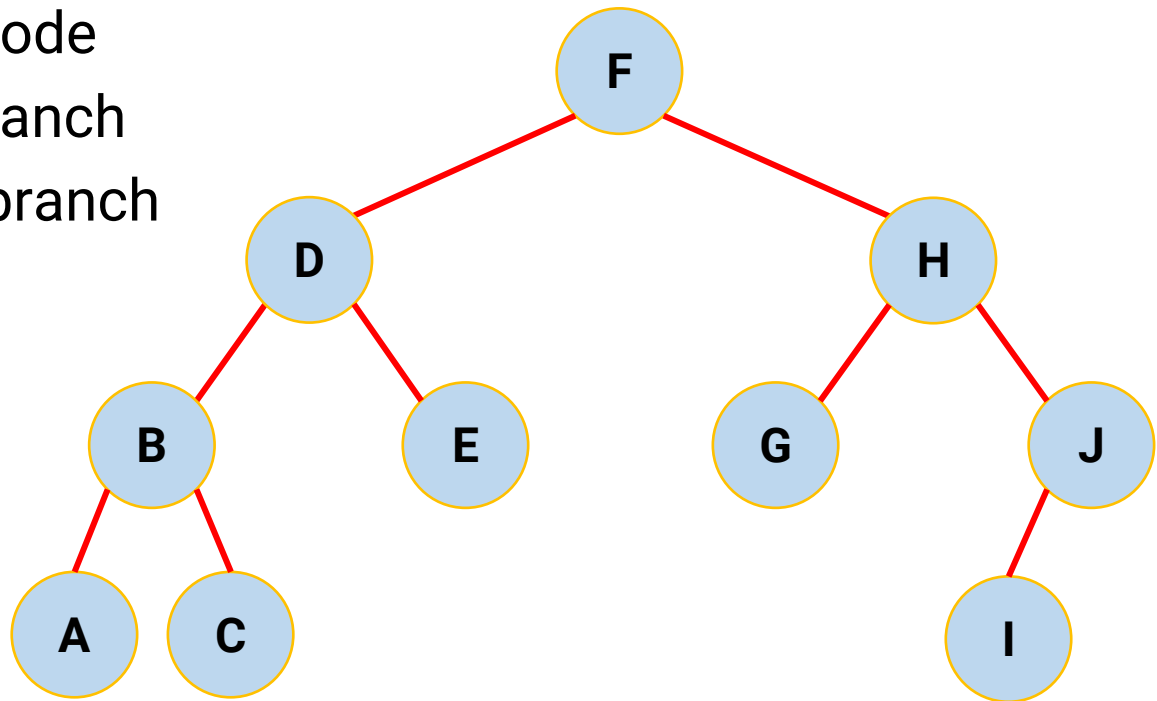 - Visit the right branch
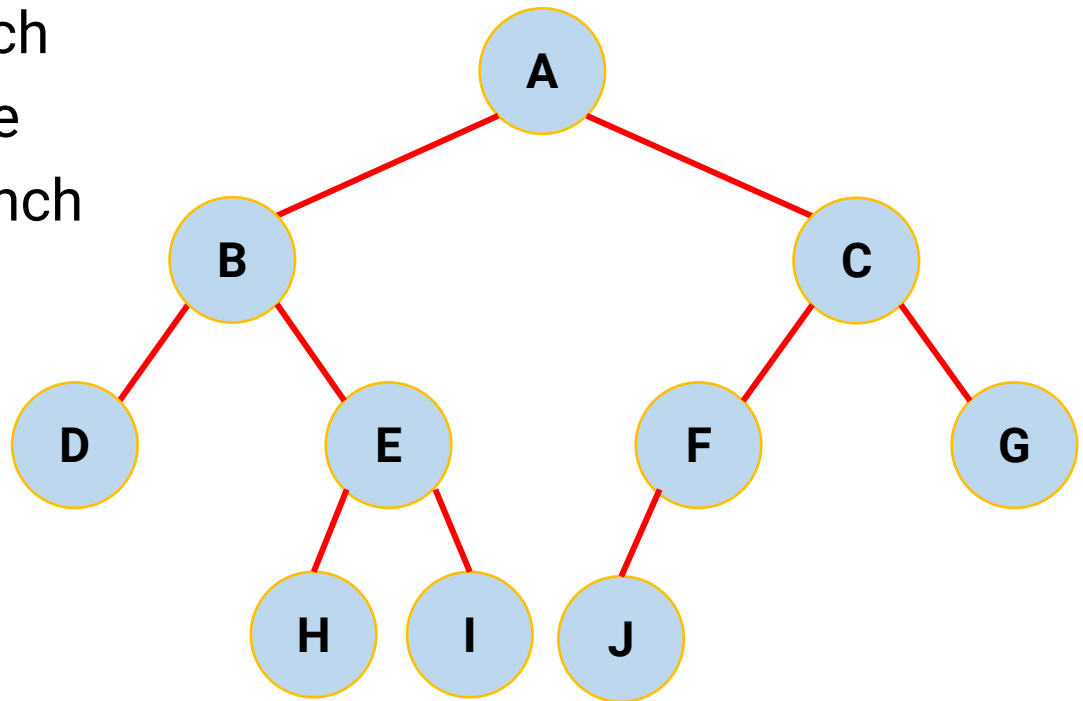
D ➜ B ➜ H ➜ E ➜ I ➜ A ➜ J ➜ F ➜ C ➜ G

Traverse Binary Tree (cont.)

- Pre-order
  - Visit the root node
  - Visit the left branch
  - Visit the right branch
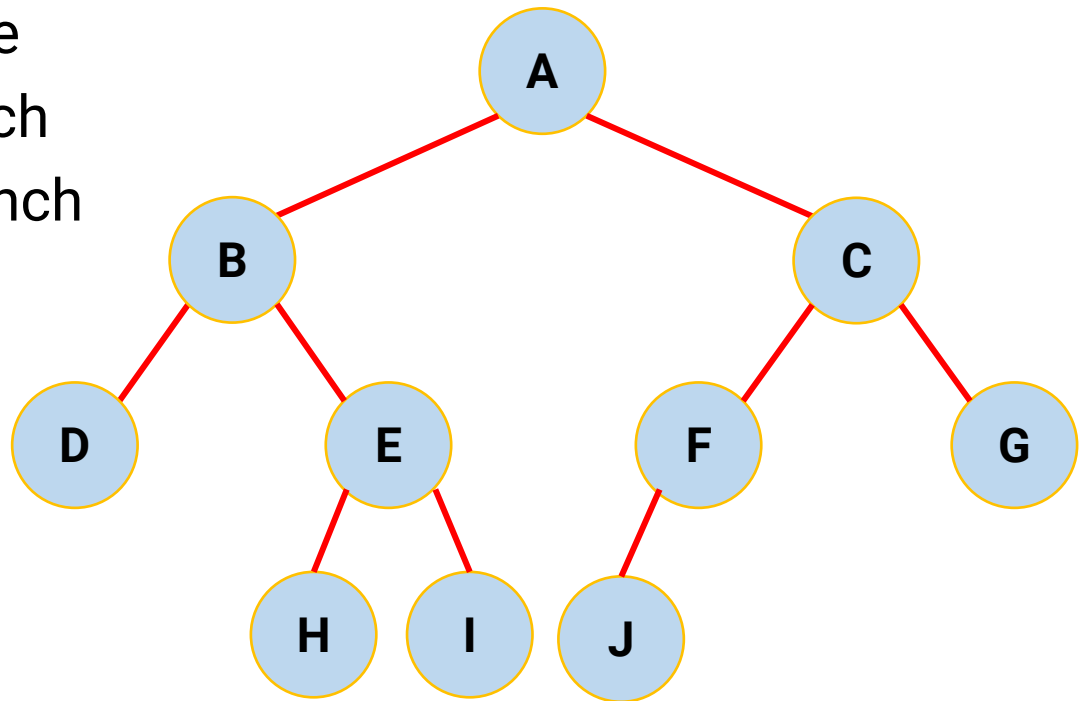
A ➔ B ➔ D ➔ E ➔ H ➔ I ➔ C ➔ F ➔ J ➔ G

Traverse Binary Tree (cont.)

- Post-order
  - Visit the left branch
  - Visit the right branch
  - Visit the root node
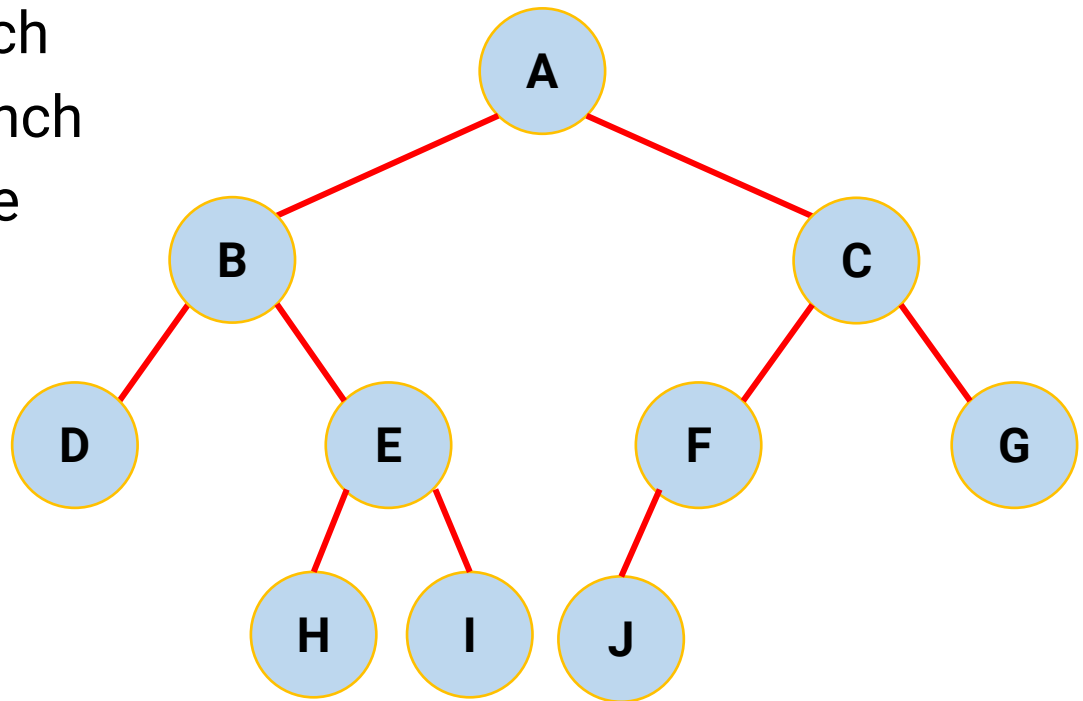
D ➜ H ➜ I ➜ E ➜ B ➜ J ➜ F ➜ G ➜ C ➜ A

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
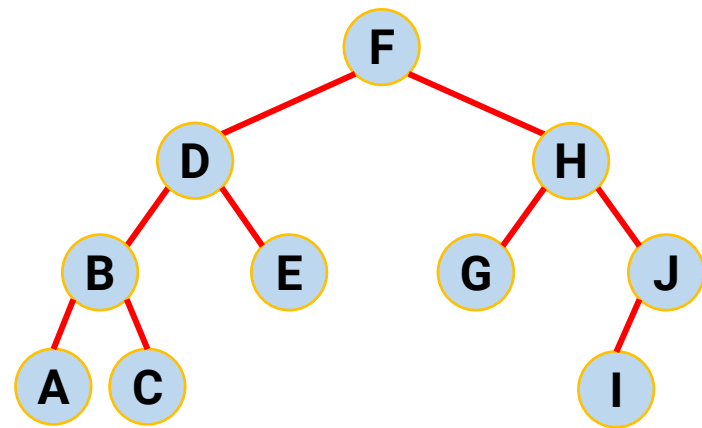
Traverse Binary Search Tree

- In-order
  - Visit the left branch
  - Visit the root node
  - Visit the right branch
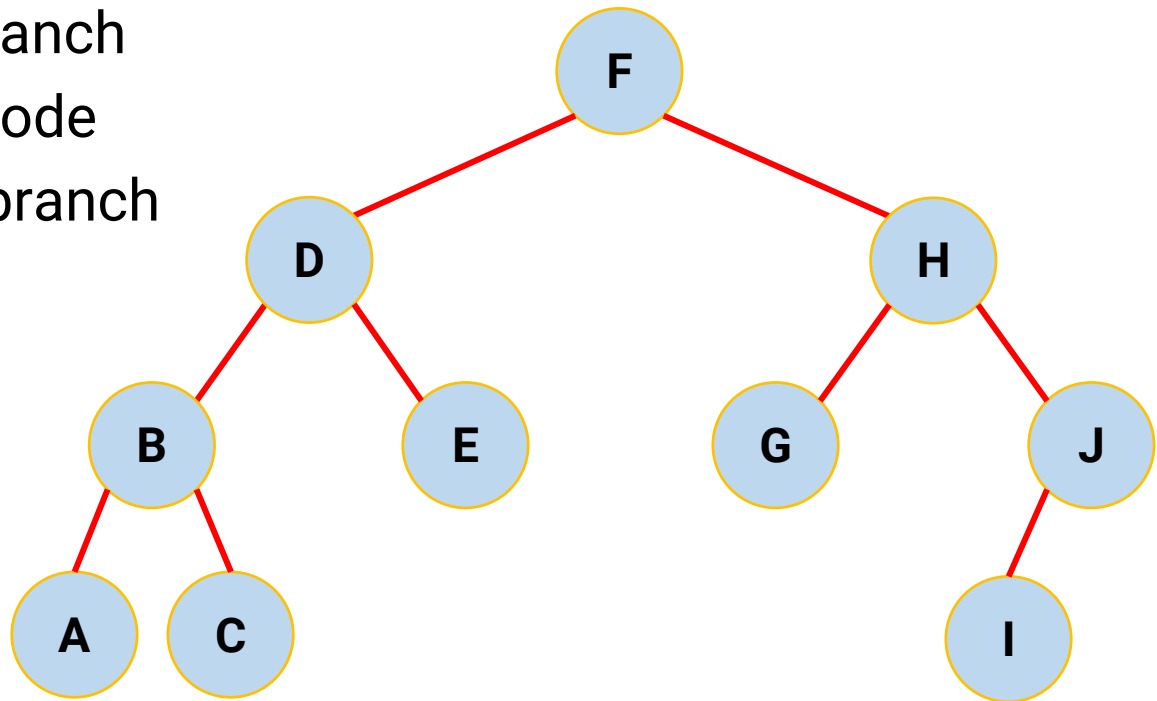
A ➔ B ➔ C ➔ D ➔ E ➔ F ➔ G ➔ H ➔ I ➔ J

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

Traverse Binary Search Tree (cont.)

- Post-order
    - Visit the left branch
    - Visit the right branch
    - Visit the root node
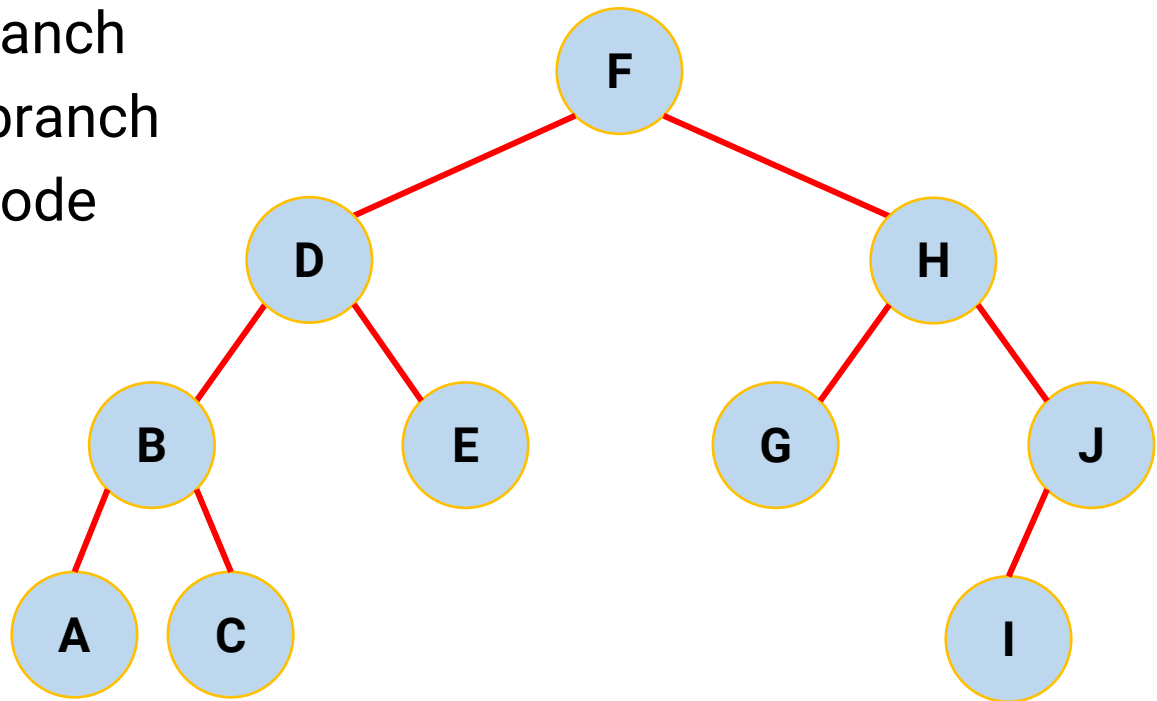
A ➜ C ➜ B ➜ E ➜ D ➜ G ➜ I ➜ J ➜ H ➜ F

Search Binary Search Tree

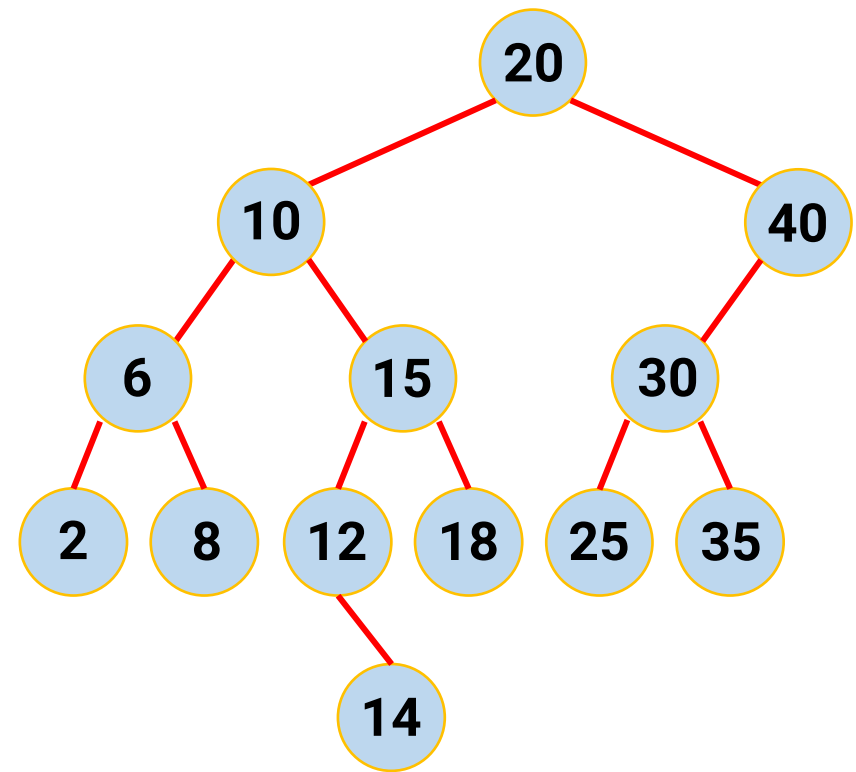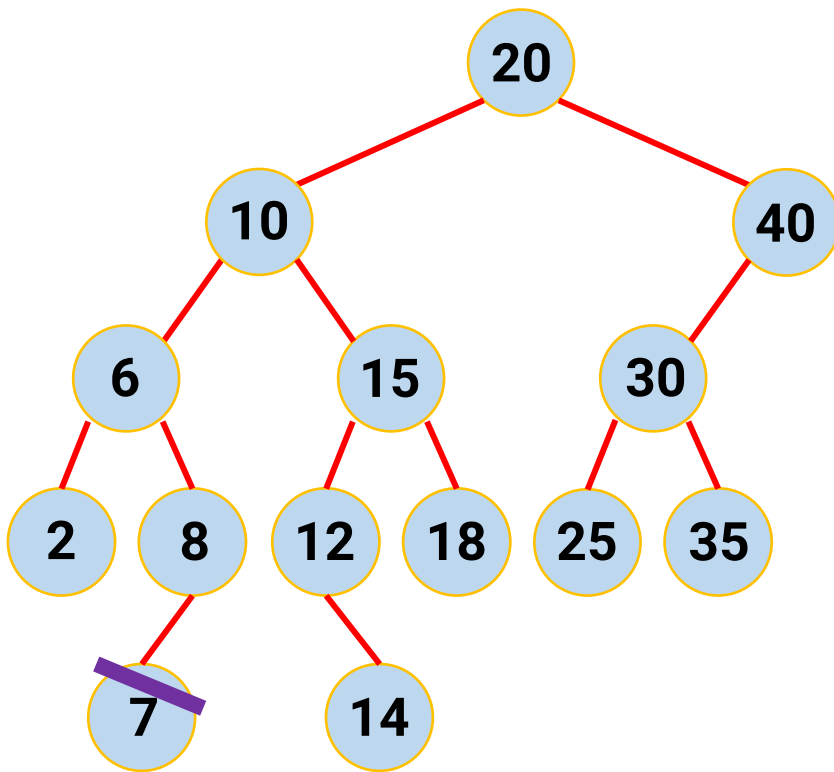- Similar to binary search (but may not be half-half)
- Example: find J
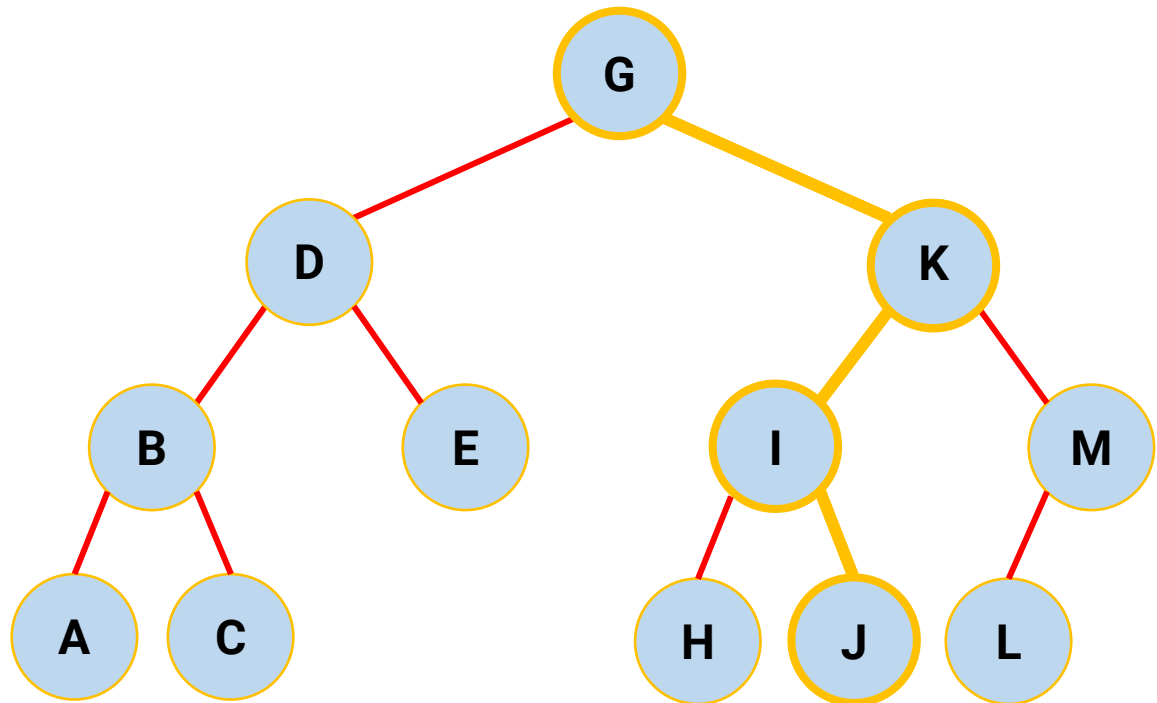
Insertion in Binary Search Tree

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

- This is the position for the insertion
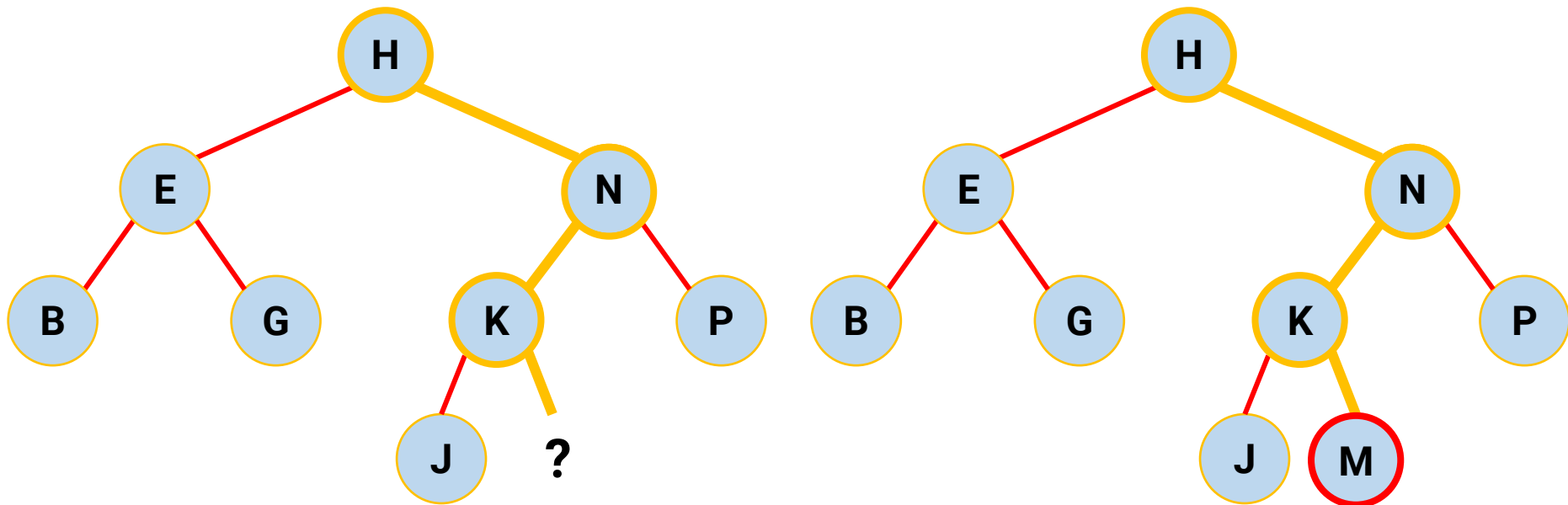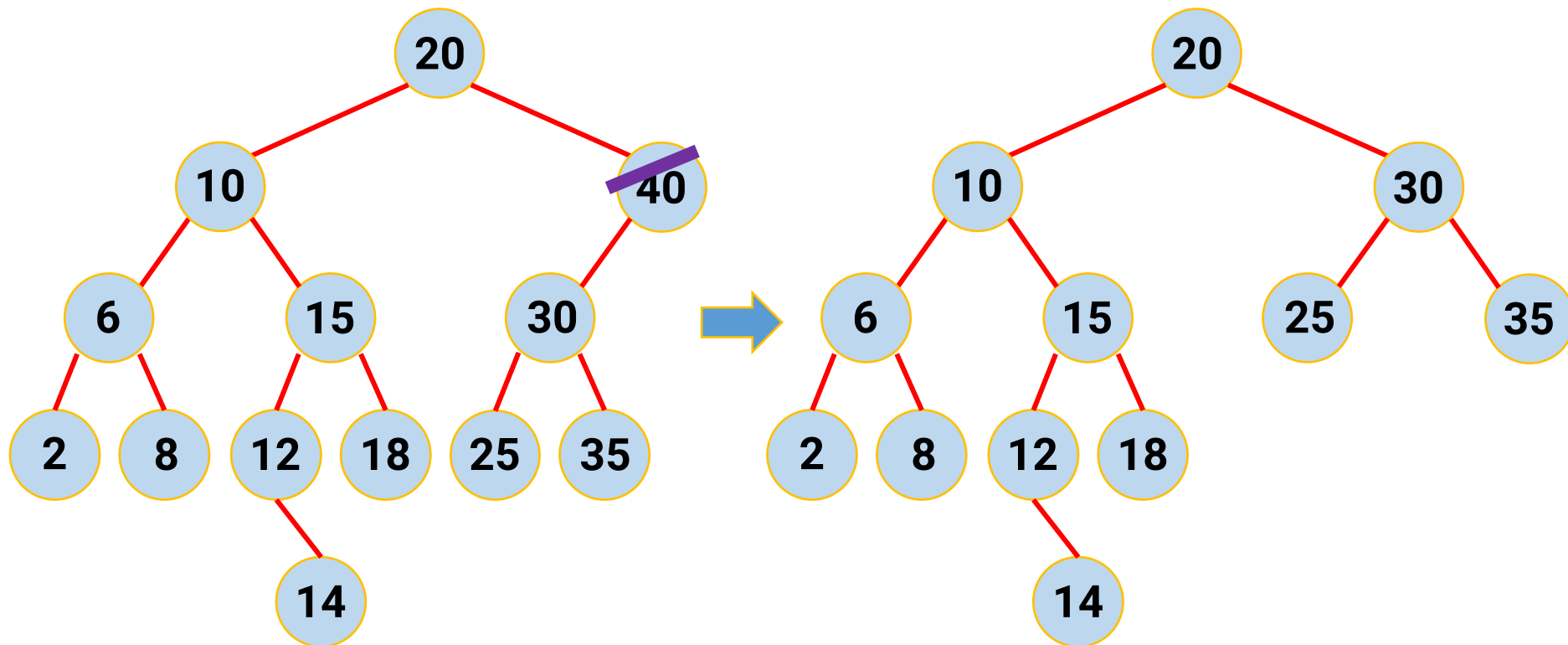
Deletion in Binary Search Tree

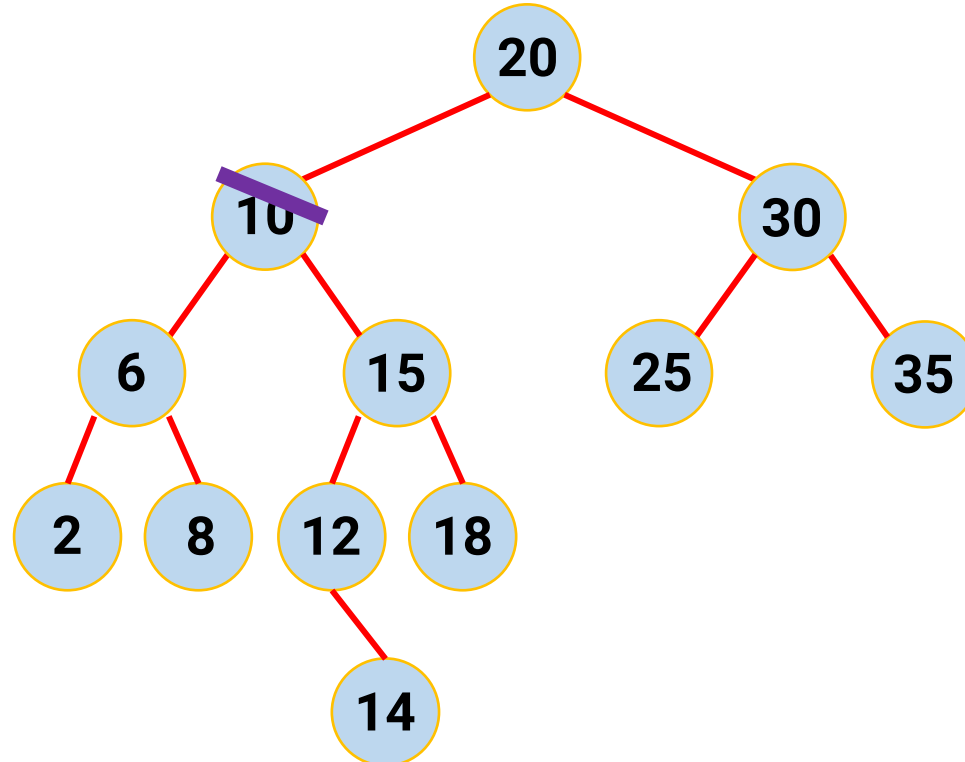- Erase a leaf **element** whose key is 7

Deletion in Binary Search Tree (cont.)
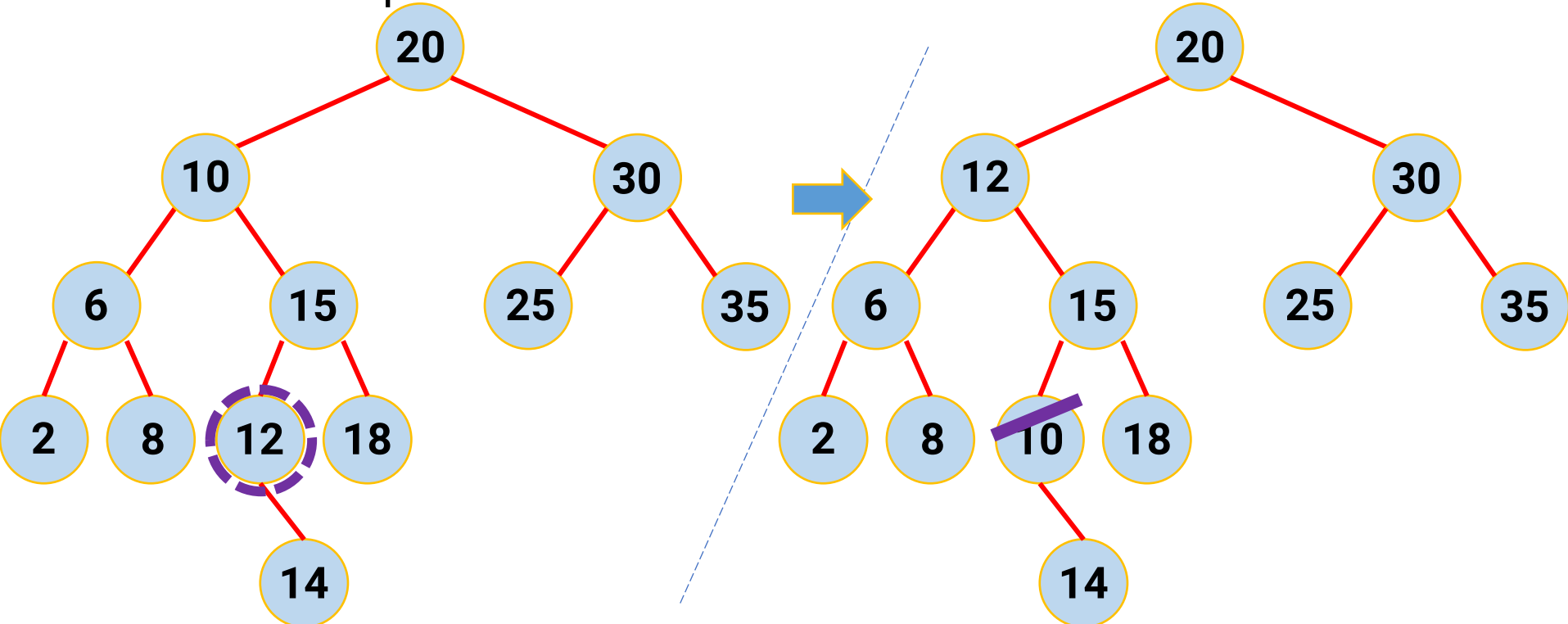
• Erase a degree-1 node whose key is 40

Deletion in Binary Search Tree (cont.)

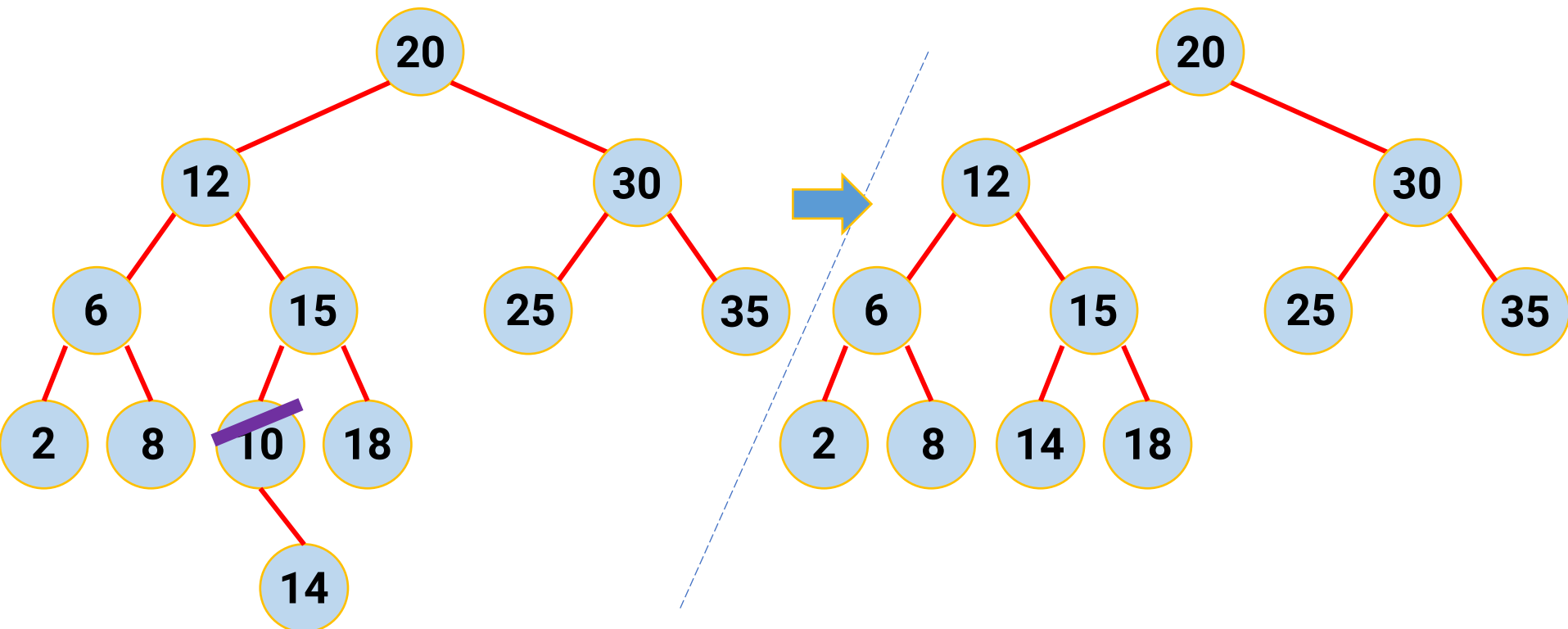- Erase a degree-2 node whose key is 10

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

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

Any Questions?