

Threaded Binary Tree & Heaps

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024



Outline

- ① Threaded Binary Trees (引線二元樹)
- ② Heaps

Outline

1 Threaded Binary Trees (引線二元樹)

2 Heaps

Threaded Binary Trees

Issue

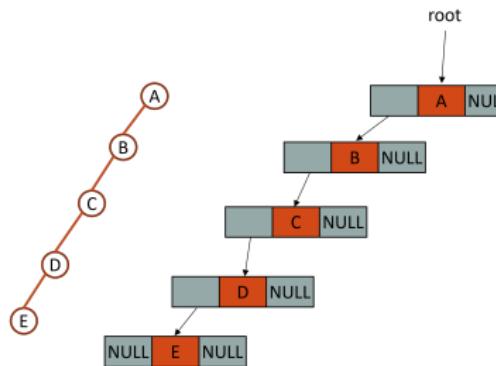
there are more null links than actual points.

Threaded Binary Trees

Issue

there are more null links than actual points.

- Number of nodes: n .
- Number of null non-null links: $n - 1$.
- Number of null links: $n + 1$.

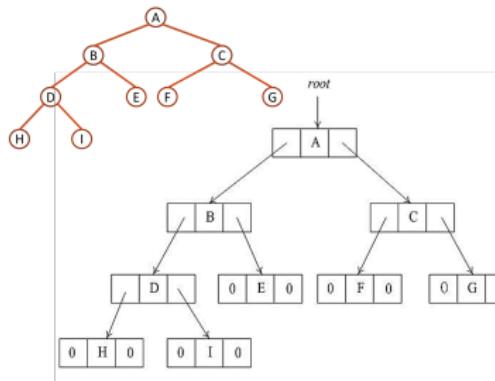


Threaded Binary Trees

Issue

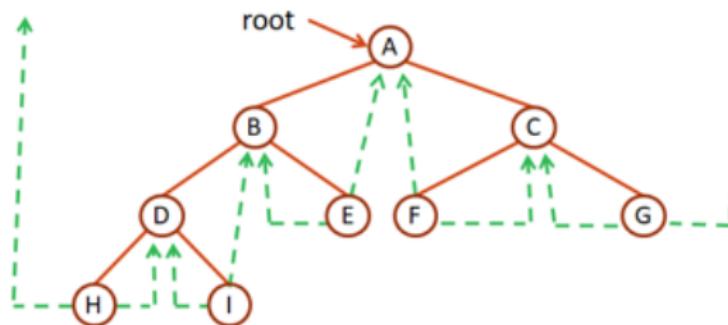
there are more null links than actual points.

- Number of nodes: n .
- Number of null non-null links: $n - 1$.
- Number of null links: $n + 1$.



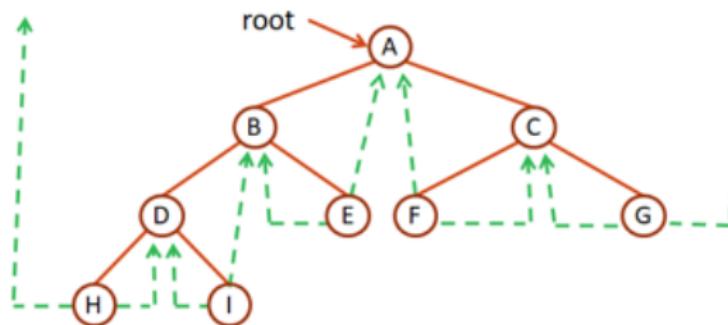
Solution

Replace the NULL Links by pointers, **threads**, pointing to other nodes.



Solution

Replace the NULL Links by pointers, **threads**, pointing to other nodes.



Threading Rules

- if $\text{ptr} \rightarrow \text{leftChild}$ is NULL, then $\text{ptr} \rightarrow \text{leftChild} = \text{inorder predecessor}$ (中序前行者) of ptr .
- if $\text{ptr} \rightarrow \text{rightChild}$ is NULL, then $\text{ptr} \rightarrow \text{rightChild} = \text{inorder successor}$ (中序後續者) of ptr .

To distinguish between normal pointers and threads

- Two **additional** fields of the node structure: **left-thread**, **right-thread**.

```
typedef struct threadedTree *threadedPointer;

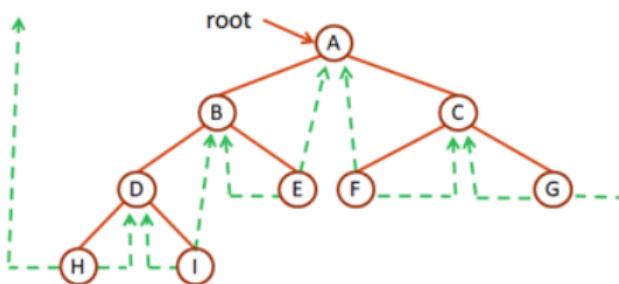
typedef struct threadedTree {
    short int leftThread;
    threadedPointer leftChild;
    char data;
    threadedPointer rightChild;
    short int rightThread;
};
```

leftThread	leftChild	data	rightChild	rightThread
------------	-----------	------	------------	-------------



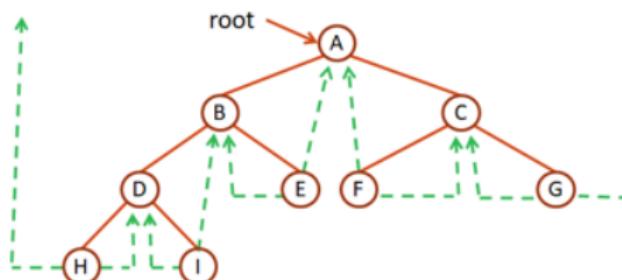
Rules of the Threading Fields

- If $\text{ptr} \rightarrow \text{leftThread} == \text{TRUE}$, $\text{ptr} \rightarrow \text{leftChild}$ contains a thread; Otherwise, the node contains a pointer to the left child.
- If $\text{ptr} \rightarrow \text{rightThread} == \text{TRUE}$, $\text{ptr} \rightarrow \text{rightChild}$ contains a thread; Otherwise, the node contains a pointer to the right child.



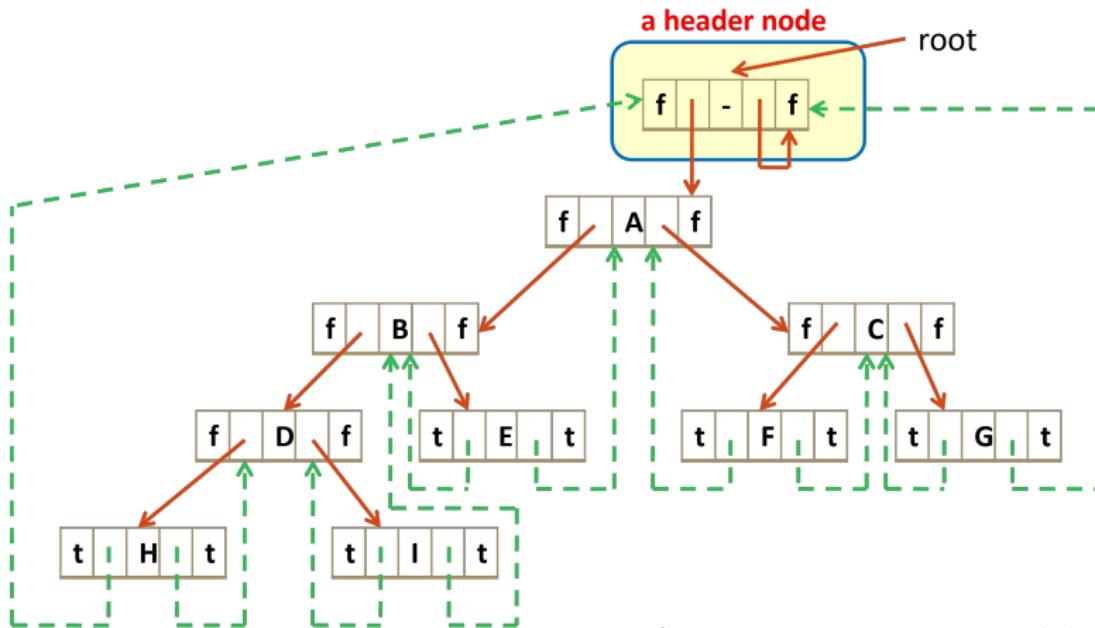
Rules of the Threading Fields

- If $\text{ptr} \rightarrow \text{leftThread} == \text{TRUE}$, $\text{ptr} \rightarrow \text{leftChild}$ contains a thread; Otherwise, the node contains a pointer to the left child.
- If $\text{ptr} \rightarrow \text{rightThread} == \text{TRUE}$, $\text{ptr} \rightarrow \text{rightChild}$ contains a thread; Otherwise, the node contains a pointer to the right child.



- Two **dangling** threads at node H and G .
 ⇒ Use a header node to collect them!

- The original tree becomes the left subtree of the head node.



Inorder sequence: H D I B E A F C G

Representing an Empty Binary Tree

leftThread	leftChild	data	rightChild	rightThread
true	:	-	.	false

```
graph LR; A[true] --- B[:]; B --- C["-"]; C --- D["."]; D --- E[false]; style A fill:#fff,stroke:#000,stroke-width:1px; style B fill:#fff,stroke:#000,stroke-width:1px; style C fill:#fff,stroke:#000,stroke-width:1px; style D fill:#fff,stroke:#000,stroke-width:1px; style E fill:#fff,stroke:#000,stroke-width:1px;
```

Finding the Inorder Successor of Node

```
threadedPointer insucc(threadedPointer tree) {  
    /* find the inorder successor of tree in a threaded  
       binary tree */  
    threadedPointer temp;  
    temp = tree->rightChild;  
    if (!tree->rightThread) // rightChild exists!  
        while (!temp->leftThread)  
            temp = temp->leftChild;  
    return temp;  
}
```

To perform an inorder traversal, we can simply make repeated calls to insucc!

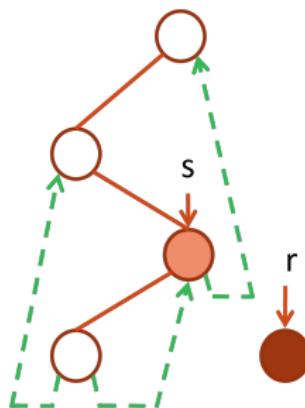


Inorder Traversal of a Threaded Binary Tree

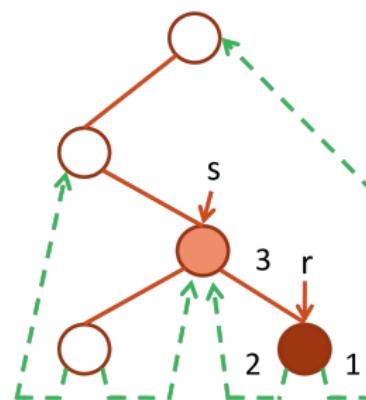
```
void traverseInorder(threadedPointer tree) {
    /* traverse the threaded binary tree inorder */
    threadedPointer temp = tree;
    while (1) {
        temp = insucc(temp);
        if (temp == tree)
            break;
        printf("%3c", temp->data);
    }
}
```

Inserting r as the rightChild of a node s

- Case I: $s \rightarrow \text{rightThread} == \text{False}$



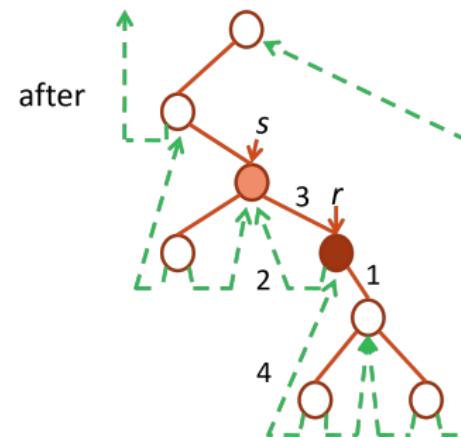
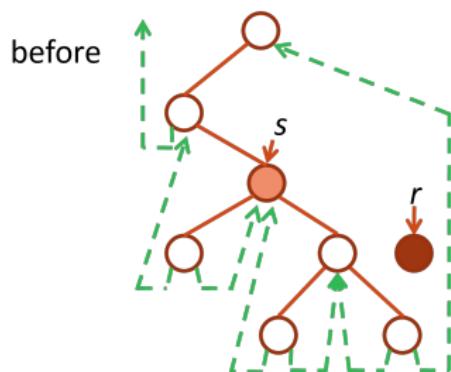
before



after

Inserting r as the rightChild of a node s

- Case II: $s \rightarrow \text{rightThread} \neq \text{False}$

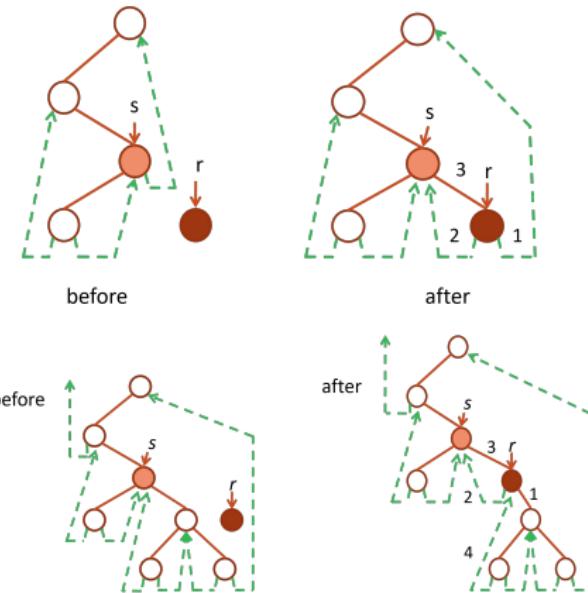


The Code for the Insertion

```

void insertRight (threadedPointer s,
                  threadedPointer r) {
/* insert r as the right child of s */
    threadedPointer temp;
    r->rightChild = s->rightChild;
    r->rightThread = s->rightThread;
    r->leftChild = s;
    r->leftThread = TRUE;
    s->rightChild = r;
    s->rightThread = FALSE;
    if (!r->rightThread){ // step 4
        temp = insucc(r);
        temp->leftChild = r;
    }
}

```



Outline

1 Threaded Binary Trees (引線二元樹)

2 Heaps



Heaps

Max Tree

A **max tree** is a tree in which

- the key value in each node \geq the key values in its children.

Heaps

Max Tree

A **max tree** is a tree in which

- the key value in each node \geq the key values in its children.

Min Tree

A **min tree** is a tree in which

- the key value in each node \leq the key values in its children.

Heaps

Max Tree

A **max tree** is a tree in which

- the key value in each node \geq the key values in its children.

Min Tree

A **min tree** is a tree in which

- the key value in each node \leq the key values in its children.

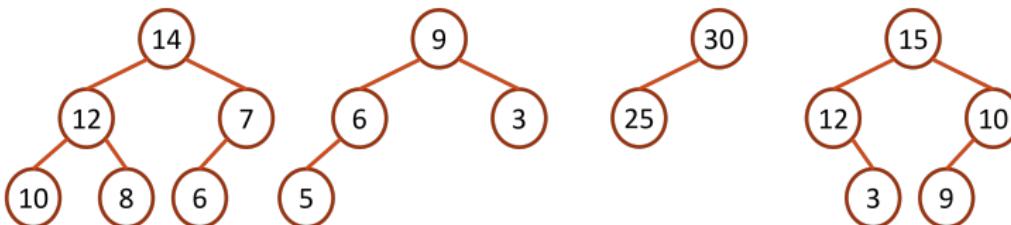
Max Heap

A complete binary tree that is also a max tree.

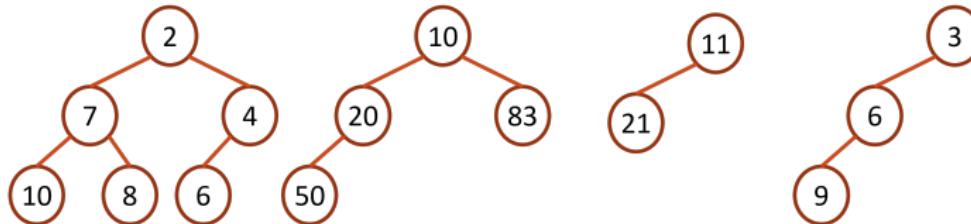
Min Heap

A complete binary tree that is also a min tree.

Examples: Max & Min Trees

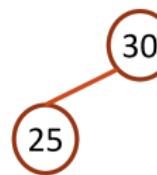
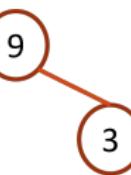
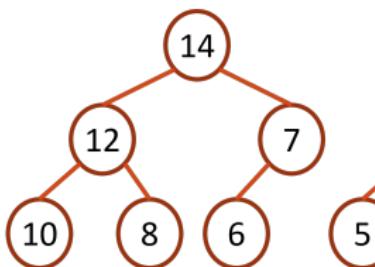


Max Trees

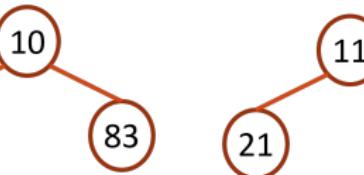
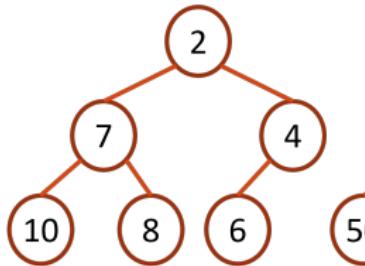


Min Trees

Examples: Max & Min Heaps



Max Heaps



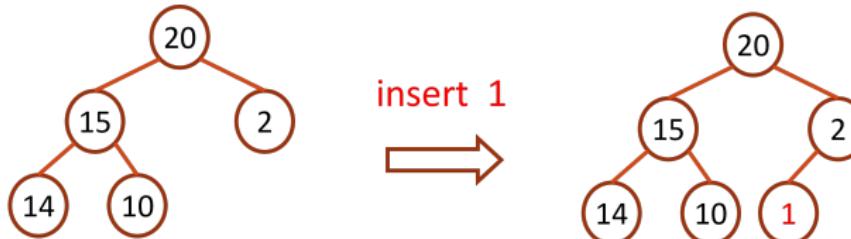
Min Heaps

The Key Application: Priority Queues

- Heaps are frequently used to implement **priority queues**.
- In this kind of queue,
 - the element to be **deleted** is the one with **highest** (or **lowest**) priority.
 - at **any time**, an element with **arbitrary priority** can be **inserted** into the queue.

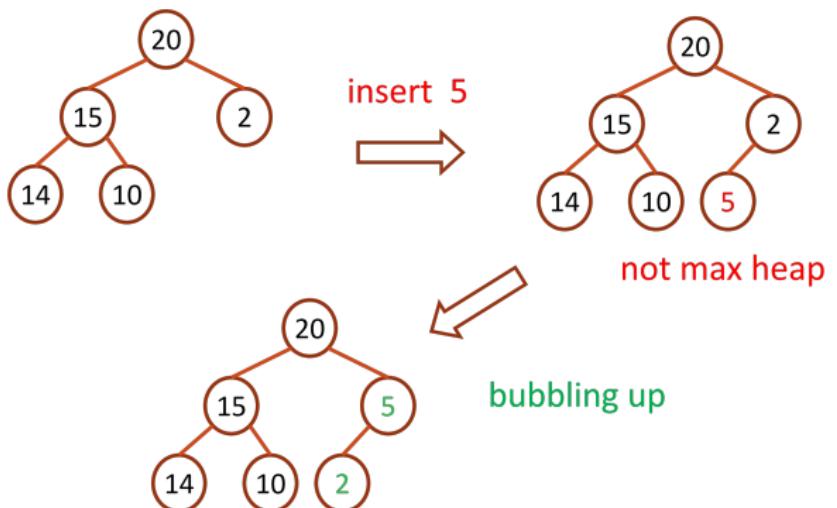
Insertion into a Max Heap

- The **bubbling process**.
 - It begins at the new node of the tree and moves toward the root.



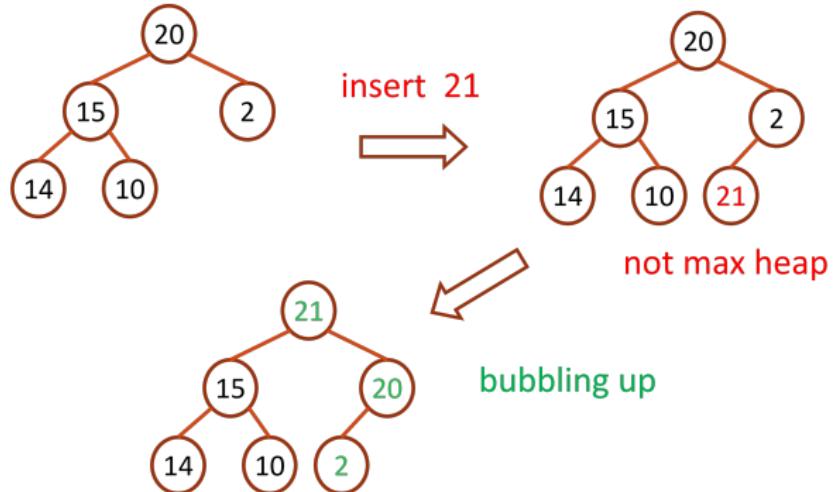
Insertion into a Max Heap

- The **bubbling process**.
 - It begins at the new node of the tree and moves toward the root.



Insertion into a Max Heap

- The **bubbling process**.
 - It begins at the new node of the tree and moves toward the root.



The Code for Insertion into a Max Heap

- Consider the following declarations:

```
#define MAX_ELEMENTS 200 /* maximum heap size+1 */  
#define HEAP_FULL (n) (n == MAX_ELEMENTS -1)  
#define HEAP_EMPTY (n) (!n)  
typedef struct {  
    int key;  
    /* other fields */  
} element;  
element heap[MAX_ELEMENTS];  
int n = 0;
```

The Code for Insertion into a Max Heap

```
void push (element item, int *n) {
    /* insert item into a max heap of current size *n */
    int i;
    if (HEAP_FULL(*n)) {
        printf("The heap is full.\n");
        exit(EXIT_FAILURE);
    } // O(1) time
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    } // O(lg n) time
    heap[i] = item; // O(1) time
}
```

- The time complexity of the insertion: $O(\lg n)$.



Discussions

