

# Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering

TGGS

KMUTNB

# Lecture 5: Searching and Sorting

# Binary Search Tree: Properties

## Properties:

- (1) Each node  $x$  in a binary search tree (BST) has a key  $x.key$ .
- (2) Nodes  $x$  other than the root have parents  $x.parent$ .
- (3) Nodes  $x$  may have a left child  $x.left$  or a right child  $x.right$  or both. (**Branching Factor** = 2)

# Binary Search Tree: Invariant

## Invariant:

For any node  $x$ , for all nodes  $y$  in the **left** subtree of node  $x$ ,  $y.key \leq x.key$  and for all nodes  $z$  in the **right** subtree of node  $x$ ,  $z.key \geq x.key$ .

# Binary Search Tree: Search

The running time of the **search operation** is  $O(h)$ , where  $h$  is the height of the tree.

---

```
1: procedure BST-SEARCH(root, key)
2:   if root == NULL then
3:     return NULL
4:   else
5:     if key < root.key then
6:       return BST-SEARCH(root.left, key)
7:     else
8:       if key > root.key then
9:         return BST-SEARCH(root.right, key)
10:      else
11:        return root
12:
```

---

# Binary Search Tree: Min and Max

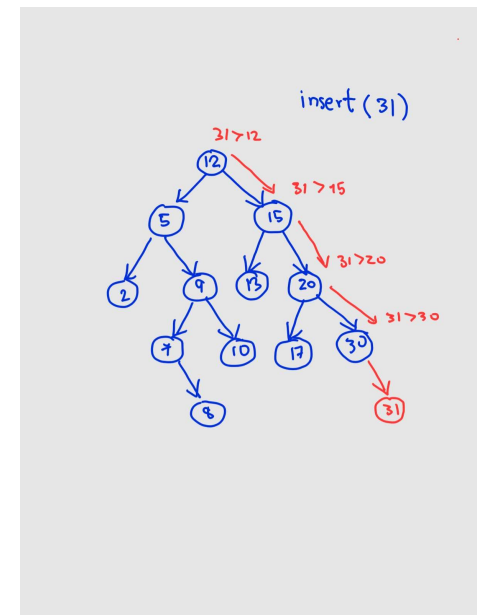
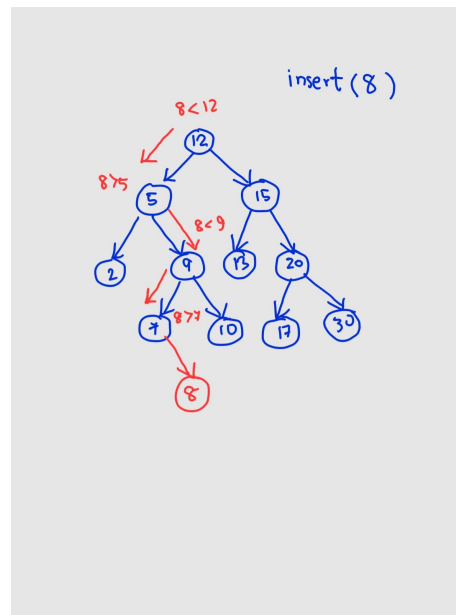
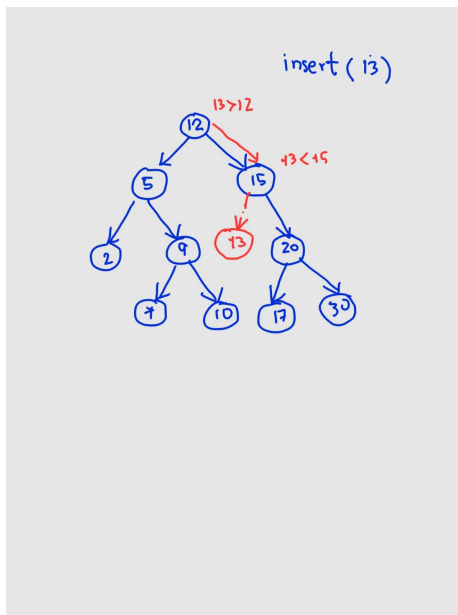
To find a *minimum key*,  
keep going *left*.

To find a *maximum key*,  
keep going *right*.

The running time of the *Min* and *Max* operations is the length of the downward path from root to the leftmost leaf and rightmost leaf, respectively.

# Binary Search Tree: Insert

Follow the left and right pointers until the right position for the key being inserted is found.



# Binary Search Tree: Delete

There are *four* cases.

*Case 1:* The node  $x$  is a leaf.

- Follow the path until node  $x$  is reached.
- Remove  $x$  by modifying its parent to replace  $x$  with NULL as its child.



# Binary Search Tree: Delete

There are *four* cases.

*Case II:* The node  $x$  has one child.

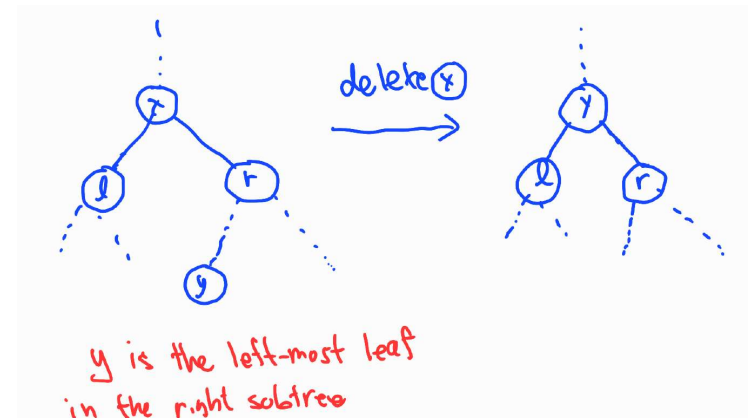
- Follow the path until  $x$  is reached.
- Remove  $x$  by elevating its child to take  $x$ 's position and modifying  $x$ 's parent to point to  $x$ 's child.

# Binary Search Tree: Delete

There are **four** cases.

**Case III:** The node  $x$  has two children and  $x$ 's successor ( $y$ ) is a **leaf** node in the right subtree.

- Locate the leftmost leaf node  $y$  in the right subtree of  $x$
- Replace  $x$  with  $y$
- Remove the now duplicated leaf node  $y$

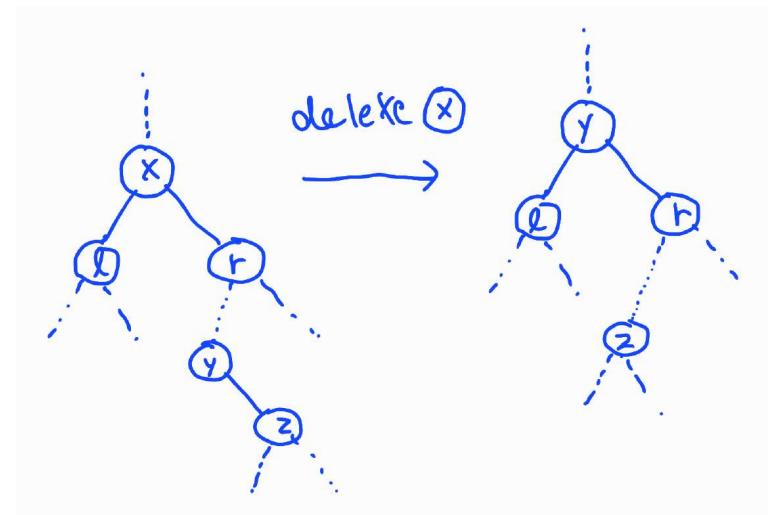


# Binary Search Tree: Delete

There are **four** cases.

**Case IV:** The node  $x$  has two children and  $x$ 's **successor** ( $y$ ) is a **non-leaf** node in the right subtree.

- Locate node  $y$  in the right subtree of  $x$
- Replace  $x$  with  $y$
- Replace  $y$  with its right subtree



# The Sorting Problem

The sorting problem can be stated as follows:

**Input:** a sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$ ,

**Output:** a permutation  $\langle a_1', a_2', \dots, a_n' \rangle$  of the input sequence  
 $a_1' \leq a_2' \leq \dots \leq a_n'$ .

The numbers that we are sorting are also known as **keys**.

# Insertion Sort

## Idea:

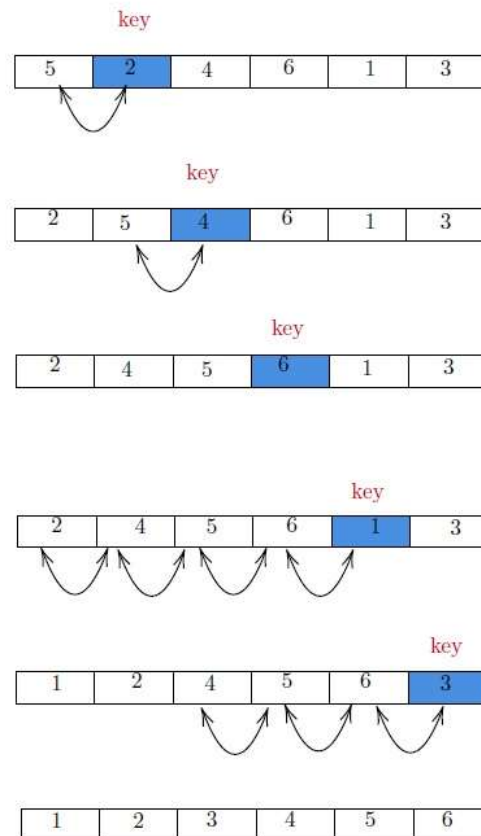
Imagine the way we sort a hand of playing cards.

- We start with an empty hand with the cards facing down on the table
- We pick up one card at a time from the table
- We insert the new card into the correct position in the left hand
  - To find the correct position, we compare the new card with the existing cards in the hand, from left to right.



*Illustration taken  
from CLRS*

# Insertion Sort



---

```
1: procedure INSERTION-SORT( $A$ )
2:   for  $j = 2 \rightarrow A.length$  do
3:      $key = A[j]$ 
4:      $i = j - 1$ 
5:     while  $i > 0 \wedge A[i] > key$  do
6:        $A[i + 1] = A[i]$ 
7:        $i = i - 1$ 
8:      $A[i + 1] = key$ 
```

---

# Insertion Sort

The index  $j$  indicates the **current card** being inserted into the left hand.

## **Observation:**

At the beginning of each iteration of the for loop,

- the subarray  $A[1 \dots j - 1]$  constitutes the currently sorted hand
- the subarray  $A[j + 1, \dots, n]$  corresponds to the pile of cards still on the table
- the elements  $A[1 \dots j - 1]$  are the elements originally in positions 1 through  $j - 1$ , but now in sorted order.

# Insertion Sort: Correctness Proof

Based on the *observation*,  
we propose the following *loop invariant*:

At the start of each iteration of the for loop, the subarray  $A[1 \dots j - 1]$  consists of the elements *originally* in  $A[1 \dots j - 1]$  but in *sorted* order.

We will use this *loop invariant* to prove the *correctness* of insertion sort.



# Loop Invariant

**Initialization:** The invariant is true prior to the first iteration of the loop.

**Maintenance:** If the invariant is true before an iteration of the loop, it remains true before the next iteration.

**Termination:** When the loop terminates, the invariant provides a useful property that can be used to show that the algorithm is correct.

# Loop Invariant

When the **Initialization** & **Maintenance** properties are true, the loop invariant is true prior to every iteration.

This is the concept of **mathematical induction**:

- **Initialization** corresponds to the **base case**
- **Maintenance** corresponds to the **inductive step**
  - Assume true for an iteration  $i$  and show true for the next iteration  $i'$
  - This is to show that the loop is inductive

The **Termination** property differs from how we use **mathematical induction** where we apply the inductive step **infinitely**.

At termination, we stop the induction and use the invariant to show that the algorithm is correct.

# Insertion Sort: Loop Invariant

**Initialization:** Before the first iteration,  $j = 2$ .

The subarray  $A[1 \dots 1] \equiv A[1]$  is ***trivially sorted*** and is also the original element in  $A[1]$  so the invariant holds prior to the first iteration.

**Maintenance:** Informally, the body of the for loop works by moving  $A[j - 1], A[j - 2], A[j - 3]$  and so on by one position to the right until it finds the proper position for  $A[j]$ , at which point  $A[j]$  is inserted.

The subarray  $A[1 \dots j]$  consists of the elements originally in  $A[1 \dots j]$ , but in sorted order.

When the loop counter  $j$  is incremented by one, the loop invariant still holds.

# Insertion Sort: Termination

**Termination:** Finally, we check what happens when the loop terminates.

Because each iteration increments  $j$  by one, we have  $j = n + 1$  on termination.

Substituting  $j = n + 1$  into the wording of the loop invariant, we have that:

at the start of each iteration of the for loop, the subarray  $A[1 \dots n]$ , which is now the **entire array**, consists of the elements **originally** in  $A[1 \dots n]$  but in **sorted** order. ■

# Insertion Sort: Running Time Analysis

## **Barometer Instruction:**

We count the total number of comparisons

- This corresponds to **Line 5**

---

```
1: procedure INSERTION-SORT( $A$ )
2:   for  $j = 2 \rightarrow A.length$  do
3:      $key = A[j]$ 
4:      $i = j - 1$ 
5:     while  $i > 0 \wedge A[i] > key$  do
6:        $A[i + 1] = A[i]$ 
7:        $i = i - 1$ 
8:      $A[i + 1] = key$ 
```

---

# Insertion Sort: Worst-Case Analysis

At each iteration  $i$  of the *inner while loop*, the elements in  $A[1 \dots i]$  are all moved one position to the right *in the worst case*.

- During each iteration,  $i = j - 1$  (Line 4)
- There can be at most  $\sum_{j=2}^n (j - 1)$  comparisons. (Line 5)

Summing up the number of comparisons:

$$\begin{aligned} T(n) &= \sum_{j=2}^n (j - 1) \\ &= \frac{n^2}{2} - \frac{n}{2} \blacksquare \end{aligned}$$

---

```
1: procedure INSERTION-SORT( $A$ )
2:   for  $j = 2 \rightarrow A.length$  do
3:      $key = A[j]$ 
4:      $i = j - 1$ 
5:     while  $i > 0 \wedge A[i] > key$  do
6:        $A[i + 1] = A[i]$ 
7:        $i = i - 1$ 
8:      $A[i + 1] = key$ 
```

---

# Insertion Sort: Average-Case Analysis

For a given iteration  $i$ , there are  $i$  possible cases as follows:

Case	$A[i]$ is	#Comparisons
1	largest	1
2	second largest	2
3	third largest	3
...	...	...
$i - 1$	second smallest	$i - 1$
$i$	smallest	$i - 1$

# Insertion Sort: Average-Case Analysis

Let  $N_i$  be a random variable representing the number of comparisons during iteration  $i$ .

$$E(N_i) = \sum_{k=1}^i n_k \Pr\{\text{Case } k \text{ happens}\}$$

$$E(N_i) = 1 \cdot \frac{1}{i} + 2 \cdot \frac{1}{i} + \cdots + (i-1) \cdot \frac{1}{i} + (i-1) \cdot \frac{1}{i}$$

$$E(N_i) = \frac{i-1}{2} + \frac{i-1}{i}$$



# Insertion Sort: Average-Case Analysis

The *expected* total number  $N$  of comparisons, where  $N = \sum_{i=2}^n N_i$ , is

$$\begin{aligned} E(N) &= \sum_{i=2}^n \left\{ \frac{i-1}{2} + \frac{i-1}{i} \right\} \leq \sum_{i=2}^n \frac{i-1}{2} + (n-1) \quad \left[ \lim_{i \rightarrow \infty} \frac{i-1}{i} = 1 \right] \\ &= \frac{n^2 + 3n - 4}{4} \end{aligned}$$

Therefore,  $E(N) = \Theta(n^2)$  ■

# Insertion Sort: in-place algorithm

It is worth noting that insertion sort is an **in-place** algorithm in which  $O(1)$  **auxiliary storage** is required to store **intermediate results** during the execution of the sorting algorithm.

**Line 3** indicates that the variable *key* is the auxiliary space used by insertion sort.

---

```
1: procedure INSERTION-SORT(A)
2:   for  $j = 2 \rightarrow A.length$  do
3:      $key = A[j]$ 
4:      $i = j - 1$ 
5:     while  $i > 0 \wedge A[i] > key$  do
6:        $A[i + 1] = A[i]$ 
7:        $i = i - 1$ 
8:      $A[i + 1] = key$ 
```

---

# Merge Sort

**Idea:** Merge sort is a **divide-and-conquer** algorithm

- Divide the array into **two subarrays** of (approximately the same size) if the array size is larger than one → **Divide**
- Sort each subarray (**recursively**) → **Conquer**
- Merge the **sorted** two subarrays → **Combine**

# Merge Sort

Merge sort has several interesting properties as follows:

- The **divide** and **conquer** parts are very simple
- The **combine** part is (computationally) complex

---

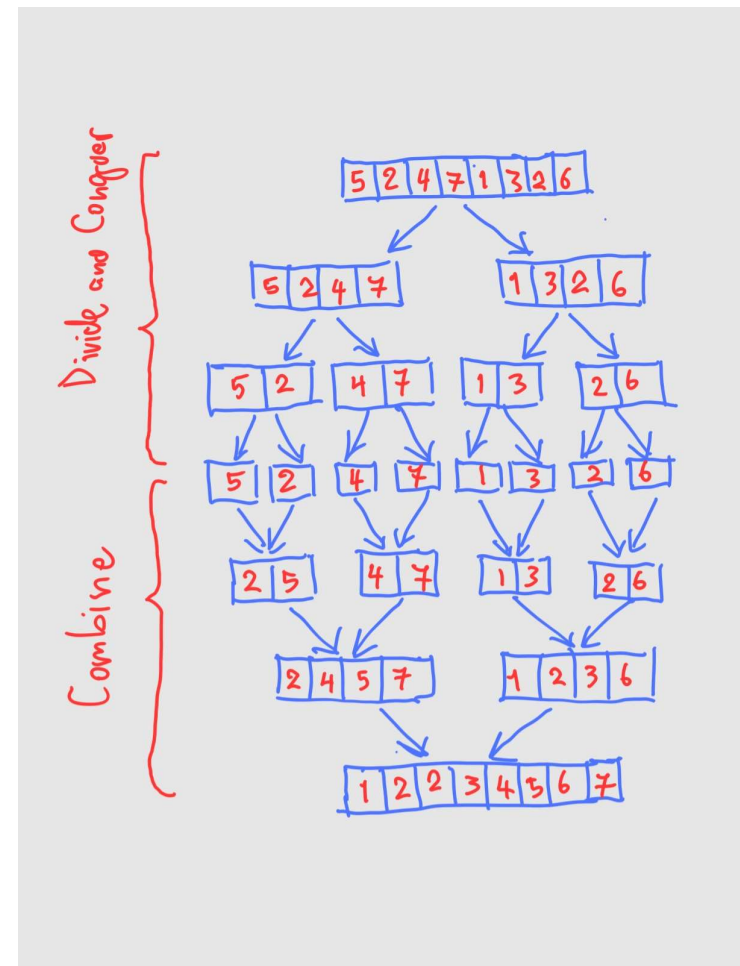
```
1: procedure MERGE-SORT( $A, p, r$ )
2:   if  $p < r$  then
3:      $q = \lfloor \frac{p+r}{2} \rfloor$ 
4:     MERGE-SORT( $A, p, q$ )
5:     MERGE-SORT( $A, q + 1, r$ )
6:     MERGE( $A, p, q, r$ )
```

---

# Merge Sort: Example

The **divide** and **conquer** parts are very simple

The **combine** part is (computationally) complex



# Merge Sort: Merge

The *MERGE* routine is the **core part** of the **merge sort** algorithm as it is where most of the computational power is spent.

The *MERGE* assumes that the two subarrays  $A[p \dots q]$  and  $A[q + 1 \dots r]$  are **already sorted**.

Let  $n_1$  and  $n_2$  denote the number of elements in the **left-half** and the **right-half** subarrays of the array  $A$ , respectively. (**Lines 2-3**)

---

```
1: procedure MERGE( $A, p, q, r$ )
2:    $n_1 = q - p + 1$ 
3:    $n_2 = r - q$ 
4:   CREATE  $L[1 \dots n_1 + 1]$ 
5:   CREATE  $R[1 \dots n_2 + 1]$ 
6:   for  $i = 1 \rightarrow n_1$  do
7:      $L[i] = A[p + i - 1]$ 
8:   for  $j = 1 \rightarrow n_2$  do
9:      $R[j] = A[q + j]$ 
10:   $L[n_1 + 1] = \infty$ 
11:   $R[n_2 + 1] = \infty$ 
12:   $i = 1$ 
13:   $j = 1$ 
14:  for  $k = p \rightarrow r$  do
15:    if  $L[i] \leq R[j]$  then
16:       $A[k] = L[i]$ 
17:       $i = i + 1$ 
18:    else
19:       $A[k] = R[j]$ 
20:       $j = j + 1$ 
```

---

# Merge Sort: Merge

**Auxiliary arrays**  $L$  and  $R$  are created to hold the left-half and the right-half subarrays, respectively. (**Lines 4-9**)

The so called **two-finger** algorithm is used to compare and merge the two subarrays. (**Lines 14-20**)

- The **left finger** points to the smallest element in the **left subarray**  $L$  that has not been copied back to  $A$
- The **right finger** points to the smallest element in the **right subarray**  $R$  that has not been copied back to  $A$

---

```
1: procedure MERGE( $A, p, q, r$ )
2:    $n_1 = q - p + 1$ 
3:    $n_2 = r - q$ 
4:   CREATE  $L[1..n_1 + 1]$ 
5:   CREATE  $R[1..n_2 + 1]$ 
6:   for  $i = 1 \rightarrow n_1$  do
7:      $L[i] = A[p + i - 1]$ 
8:   for  $j = 1 \rightarrow n_2$  do
9:      $R[j] = A[q + j]$ 
10:   $L[n_1 + 1] = \infty$ 
11:   $R[n_2 + 1] = \infty$ 
12:   $i = 1$ 
13:   $j = 1$ 
14:  for  $k = p \rightarrow r$  do
15:    if  $L[i] \leq R[j]$  then
16:       $A[k] = L[i]$ 
17:       $i = i + 1$ 
18:    else
19:       $A[k] = R[j]$ 
20:       $j = j + 1$ 
```

---

# Merge Sort: Proof of Correctness

Since *MERGE* is the core part, we will focus on the correctness of the *MERGE* routine by considering its **loop invariant**.

Once we show the correctness of *MERGE*, it is easy to see the correctness of *MergeSort*.

## **Loop Invariant:**

- (1) At the start of each iteration of the for loop of **Lines 14-20**, the subarray  $A[p \dots k - 1]$  contains the  $k - p$  smallest elements of  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$  in sorted order.
- (2)  $L[i]$  and  $R[j]$  are the smallest elements of the two arrays that have not been copied back to  $A$ .



# Merge Sort: Proof of Correctness

## Initialization:

Prior to the first iteration,  $k = p$ .

Therefore, the subarray  $A[p \dots p - 1]$  is **empty** and contains  $k - p = k - k = 0$  smallest elements.

Moreover,  $i = j = 1$ .  $L[1]$  and  $R[1]$  contain the smallest elements of the two subarrays that have not been copied back into  $A$ .

# Merge Sort: Proof of Correctness

## Maintenance:

Consider the case  $L[i] \leq R[j]$ .

By the loop invariant and the fact that  $L[i] \leq R[j]$ ,  $L[i]$  is the smallest element not copied back to  $A$ .

By the loop invariant,  $A[p \dots k-1]$  contains the smallest  $k-p$  elements.

By inspection,  $A[k] = L[i]$  (**Line 16**).

(1) Therefore,  $A[p \dots k]$  **now** contains the smallest  $k-p+1$  elements.

(2) Moreover,  $L[i+1]$  ( $L[i]$  has just been copied back to  $A$  by **Line 16**) and  $R[j]$  contain the smallest elements of the two subarrays that have not been copied back to  $A$ .

# Merge Sort: Proof of Correctness

## Maintenance: (Continued)

After **Line 17**,  $i$  is incremented,  $j$  remains the same and  $k$  is incremented:

$$i' = i + 1, j' = j \text{ and } k' = k + 1$$

(1) Therefore,  $A[p \dots k' - 1]$  contains the smallest  $k' - p$  elements.

(2)  $L[i']$  and  $R[j']$  now contains the smallest elements of the two arrays that have not been copied back to  $A$ .

This reestablishes the loop invariant at the start of the next iteration.

\*\*\*The maintenance property for the other case  $L[i] > R[j]$  can be shown in a similar way.

# Merge Sort: Proof of Correctness

## Termination:

When the for loop terminates,  $k = r + 1$ .

By the definition of the loop invariant,

$A[p \dots k - 1] = A[p \dots (r + 1) - 1] = A[p \dots r]$  contains the smallest  $k - p = (r + 1) - p = r - p + 1$  elements in sorted order.

Therefore, we have just shown that *MERGE* works correctly by producing a sorted array  $A[p \dots r]$  ■

# Merge Sort: Time Complexity

The running time  $T(n)$  for the base case ( $n = 1$ ) is  $O(1)$ .

The running time  $T(n)$  for the recursive case ( $n > 1$ ) is split into the following components:

the **divide** part costs  $O(1)$

the **conquer** part costs  $2T\left(\frac{n}{2}\right)$

the **combine** part costs  $O(n)$

Therefore,  $T(n) = O(1) + 2T\left(\frac{n}{2}\right) + O(n)$

Allowing the asymptotically smaller term  $O(1)$  to be absorbed into the asymptotically larger term  $O(n)$ , we have

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \text{ for } n > 1 \blacksquare$$

# Merge Sort: Time Complexity

Solving the recurrence relation, we get

$$T(n) = \Theta(n \log n)$$

**NB:** you solved this using the *recursion tree method* in PS2.4 as homework.

## Merge Sort: out-of-place

Merge sort needs *auxiliary space*  $\Theta(n)$  to store *intermediate results* during the execution of the sorting algorithm.

Therefore, it is *not* an in-place algorithm.

# Summary

We have covered the following topics:

- Binary Search Tree and its basic operations
- Insertion Sort
- Merge Sort
- Correctness Proof using Loop Invariants

In the next lecture, we will cover *divide and conquer*.