
# AP® Computer Science AB
# 2006 Scoring Guidelines

## The College Board: Connecting Students to College Success

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities, and concerns.



**Visit the College Board on the Web: www.collegeboard.com.**
**AP Central is the official online home for the AP Program: apcentral.collegeboard.com.**

# AP® COMPUTER SCIENCE AB
# 2006 SCORING GUIDELINES

## Question 1: Thesaurus

**Part A:** `addSynonym` **4 points**

**+1/2** correctly check if `word` is already stored in `wordMap`

**+2** new word
- **+1/2** correctly create a new set (either `TreeSet` or `HashSet`)
- **+1/2** correctly add `syn` to set
- **+ 1** add new entry to `wordMap`
  - **+1/2** attempt (`wordMap.put(word,syn)` OK)
  - **+1/2** correct

**+1 1/2** existing word
- **+ 1** access the set of synonyms
  - **+1/2** attempt (must access element of collection)
  - **+1/2** correct
- **+1/2** add `syn` to the set of synonyms

**Part B:** `removeSynonym` **5 points**

**+1/2** correctly create a new set (either `TreeSet` or `HashSet`)

**+1 1/2** iterate over all words in `wordMap`
- **+1/2** attempt to iterate over words in map
- **+1/2** get `keySet`
- **+1/2** correctly access each word in `keySet`

**+2 1/2** process words (in context of loop)
- **+ 1** access the set of synonyms
  - **+1/2** attempt (must access element of collection)
  - **+1/2** correct
- **+1/2** check whether the set contains `syn`
- **+1/2** remove `syn` from set of synonyms in correct context of check
- **+1/2** add word to set of affected words in correct context of check

**+1/2** return set of affected words (without destroying the `keySet`)

# AP® COMPUTER SCIENCE AB
# 2006 SCORING GUIDELINES

## Question 2: Packs & Bundles (Design)

**Part A:** Pack **3 1/2 points**

**+1/2** `class Pack implements Product`
**+1/2** declare both private fields (`int` and `Product`)

**+1** constructor
- **+1/2** `public Pack(int ?, Product ?)` (`Item` OK if matches field)
- **+1/2** initialize fields

**+1 1/2** `getPrice`
- **+1/2** `public double getPrice()`
- **+1/2** access product's current price and pack's current number
- **+1/2** calculate and return price

*lose if don't call getPrice on a product*

**Part B:** `Bundle` **5 1/2 points**

**+1/2** `class Bundle implements Product`
**+1/2** private collection field

**+1** constructor (*if collection is initialized when declared, no constructor is needed*)
- **+1/2** `public Bundle()`
- **+1/2** initialize collection

**+1** add
- **+1/2** `public void add(Product ?)` (*No penalty if returns reasonable value*)
- **+1/2** add parameter to collection

**+2 1/2** `getPrice`
- **+1/2** `public double getPrice()`
- **+1/2** declare & initialize sum (must be added to)
- **+1/2** loop over every element in collection
- **+1/2** sum is updated with `getPrice` for each element
- **+1/2** return sum

*lose if accum. price in add()*

**Common Usage:** **-1** for extraneous code with compile-time errors
**-1/2** missing `public` on `getPrice`

## Question 3: Waiting List

**Part A:** `getKthNode` **3 points**

**+1/2** create copy reference (used in loop) *if recursive solution:* returns `front` if $k == 0$

**+1 1/2** iterate through list
- **+1/2** loop (or recursion) referring to the list
- **+1** update list reference *in context of loop/recursion*
  - **+ 1/2** attempt
  - **+ 1/2** correct

**+1** return kth node

**Part B:** `transferNodesFromEnd` **6 points**

**+1 1/2** get sublist from `other`
- **+1/2** attempt to access sublist (manual traversal OK)
- **+1** correctly access sublist w/ `getKthNode` (or previous node if later use `getNext`)

**+2** add sublist to end of `this`
- **+1** get reference to last node (manual traversal OK)
  - **+ 1/2** attempt (must include valid reference to `null` or `size` or `numNodes`)
  - **+ 1/2** correct
- **+1** add nodes from `other` to end (w/o creating any `ListNodes`)
  - **+1/2** attempt
  - **+1/2** correct

**+1 1/2** remove nodes from end of `other`
- **+ 1** remove nodes
  - **+1/2** attempt (manual traversal OK)
  - **+1/2** correct when `other.size()!= num` ( must call `getKthNode`)
- **+1/2** set `other.front` to `null` if all moved (CANNOT define `setFront` public mutator)

**+1** update counts
- **+1/2** add `num` to `numNodes`
- **+1/2** subtract `num` from `other.numNodes` (CANNOT define public `setNumNodes` mutator)

**Common Usage:** Moving `front` is **-1** for destruction of data structure.

## Question 4: Path Finder (MBS)

**Part A:** `possibleEnds` **2 1/2 points**

**+1** check ends to see if empty
- **+1/2** attempt to check if one end is empty
- **+1/2** correctly check both ends using `isEmpty` or `objectAt`

**+1** check ends for direction
- **+1/2** attempt to compare # rows or cols of `start & end`
- **+1/2** correctly compare # rows & cols

**+1/2** return correct value

**Part B:** `findNEPath` **6 1/2 points**

**+1** check endpoints
- **+1/2** call `possibleEnds(start, end)`
- **+1/2** return `null` if fails

**+2** base case (when `start` equals `end`)
- **+1/2** compare to see if `start.equals(end)` (both must be empty)
- **+1/2** create `List`
- **+1/2** add `start/end` to list
- **+1/2** return list

**+1 1/2** recursive cases
- **+1/2** attempt recursive call from neighbor of `start` (N or E) or `end` (S or W)
- **+1/2** attempt other recursive call whenever first case fails
- **+1/2** correct calls for both directions

**+2** recursive application
- **+1** add `start/end` to recursive path
  - **+1/2** attempt (must have `start/end` and recursive path)
  - **+1/2** correct
- **+1/2** return updated path
- **+1/2** return `null` if and only if no path exists

# AP® COMPUTER SCIENCE A/AB
# 2006 GENERAL USAGE

Most common usage errors are addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem, even if it occurs on different parts of a problem.

## Nonpenalized Errors

spelling/case discrepancies*

local variable not declared when any other variables are declared in some part

default constructor called without parens; for example, `new Fish;`

use keyword as identifier

`[r,c]`, `(r)(c)` or `(r,c)` instead of `[r][c]`

= instead of == (and vice versa)

`length`/`size` confusion for array, `String`, and `ArrayList`, with or without `()`

`private` qualifier on local variable

extraneous code with no side-effect, for example a check for precondition

common mathematical symbols for operators (x • ÷ ≤ ≥ < > ≠)

missing { } where indentation clearly conveys intent

missing `( )` on method call or around `if/while` conditions

missing `;`s

missing "new" for constructor call once, when others are present in some part

missing downcast from collection

missing `int` cast when needed

missing `public` on class or constructor header

## Minor Errors (1/2 point)

confused identifier (e.g., `len` for `length` or `left()` for `getLeft()` )

no local variables declared

`new` never used for constructor calls

`void` method or constructor returns a value

modifying a constant (`final`)

use `equals` or `compareTo` method on primitives, for example
`int x; …x.equals(val)`

`[]` – `get` confusion if access not tested in rubric

assignment dyslexia, for example, `x + 3 = y;` for `y = x + 3;`

`super(method())` instead of `super.method()`

formal parameter syntax (with type) in method call, e.g., `a = method(int x)`

missing `public` from method header when required

"false"/"true" or 0/1 for boolean values

"null" for `null`

## Major Errors (1 point)

extraneous code which causes side-effect, for example, information written to output

use interface or class name instead of variable identifier, for example `Simulation.step()` instead of `sim.step()`

`aMethod(obj)` instead of `obj.aMethod()`

use of object reference that is incorrect, for example, use of `f.move()` inside method of `Fish` class

use private data or method when not accessible

destruction of data structure (e.g., by using root reference to a `TreeNode` for traversal of the tree)

use class name in place of `super` either in constructor or in method call

**Note: Spelling and case discrepancies for identifiers fall under the "nonpenalized" category as long as the correction can be unambiguously inferred from context. For example, "Queu" instead of "Queue". Likewise, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the context allows for the reader to assume the object instead of the class.*

# AP® COMPUTER SCIENCE AB
# 2006 CANONICAL SOLUTIONS

## Question 1: Thesaurus

**PART A:**

```
public void addSynonym(String word, String syn)
{
  if (!wordMap.containsKey(word))
  {
    Set synonyms = new TreeSet();
    synonyms.add(syn);
    wordMap.put(word, synonyms);
  }
  else
  {
    Set synonyms = (Set)wordMap.get(word);
    synonyms.add(syn);
  }
}
```

**PART B:**

```
public Set removeSynonym(String syn)
{
  Set affectedWords = new TreeSet();

  Set allWords = wordMap.keySet();
  Iterator iter = allWords.iterator();
  while (iter.hasNext())
  {
    String nextWord = (String)iter.next();
    Set synonyms = (Set)wordMap.get(nextWord);
    if (synonyms.remove(syn))
    {
      affectedWords.add(nextWord);
    }
  }

  return affectedWords;
}
```

# AP® COMPUTER SCIENCE AB
# 2006 CANONICAL SOLUTIONS

## Question 2: Packs & Bundles (Design)

**PART A:**

```
public class Pack implements Product
{
  private int numProducts;
  private Product prod;

  public Pack(int num, Product p)
  {
    numProducts = num;
    prod = p;
  }

  public double getPrice()
  {
    return prod.getPrice() * numProducts;
  }
}
```

**PART B:**

```
public class Bundle implements Product
{
  private ArrayList productList;

  public Bundle()
  {
    productList = new ArrayList();
  }

  public void add(Product newProd)
  {
    productList.add(newProd);
  }

  public double getPrice()
  {
    double totalCost = 0.0;
    for (int i = 0; i < productList.size(); i++)
    {
      totalCost += ((Product)productList.get(i)).getPrice();
    }

    return totalCost;
  }
}
```

# AP® COMPUTER SCIENCE AB
# 2006 CANONICAL SOLUTIONS

## Question 3: Waiting List

**PART A:**

```
public ListNode getKthNode(int k)
{
  ListNode step = front;
  for (int i = 0; i < k; i++)
  {
    step = step.getNext();
  }
  return step;
}
```

**ALTERNATE SOLUTION**

```
public ListNode getKthNode(int k)
{
  if (k == 0)
  {
    return front;
  }
  return getKthNode(k-1).getNext();
}
```

**PART B:**

```
public void transferNodesFromEnd(WaitingList other, int num)
{
  ListNode lastNode = getKthNode(size()-1);
  lastNode.setNext(other.getKthNode(other.size()-num));

  if (other.numNodes == num)
  {
    other.front = null;
  }
  else
  {
    other.getKthNode(other.size()-num -1).setNext(null);
  }

  numNodes += num;
  other.numNodes -= num;
}
```

# AP® COMPUTER SCIENCE AB
# 2006 CANONICAL SOLUTIONS

## Question 4: Path Finder (MBS)

**PART A:**

```
private boolean possibleEnds(Location start, Location end)
{
  return (theEnv.isEmpty(start) && theEnv.isEmpty(end) &&
          start.row() >= end.row() && start.col() <= end.col());
}
```

**PART B:**

*Note: Commented code represents a more common approach but utilizes a* `List` *method not in the APCS Quick Reference. Each commented line replaces the line above it in the alternate solution.*

```
public List findNEPath(Location start, Location end)
{
  if (!possibleEnds(start, end))
  {
    return null;
  }

  List path;
  if (start.equals(end))
  {
    path = new LinkedList();
    path.add(end);
    // path.add(0, start);
    return path;
  }

  path = findNEPath(start, theEnv.getNeighbor(end, Direction.SOUTH));
  // path = findNEPath(theEnv.getNeighbor(start, Direction.NORTH), end);


  if (path == null)
  {
    path = findNEPath(start, theEnv.getNeighbor(end, Direction.WEST));
    // path = findNEPath(theEnv.getNeighbor(start, Direction.EAST), end);
  }

  if (path != null)
  {
    path.add(end);
    // path.add(0, start);
  }

  return path;
}
```