The Neglected Middle Novice Programmer: Reading and Writing without Abstracting

Raymond Lister
Faculty of Information Technology
University of Technology, Sydney
Australia
raymond@it.uts.edu.au

Abstract

Many teachers of novice programmers have lamented that students either seem to have a natural gift for programming, or have no gift for it at all. In this paper, we discuss a third group of students, the middle novice programmer. At the completion of their first semester of programming, these students can manifest a strong concrete grasp of the semantics of basic programming language constructs, by hand executing code, but they cannot reason about code at a higher goal/plan level. The research evidence presented in this paper for the existence of these middle novice programmers is from the analysis of twelve multiple choice questions, which students attempted as part of an end-of-first-semester exam.

Keywords: novice programmers, CS1, schema, plans.

1 Introduction

For academics who compartmentalize their teaching and research, an exam paper is an instrument for assigning grades to students. For a scholar of teaching, however, an exam paper is also an opportunity to study the learning of the students. Whilst the scholar of teaching does so with the aim of improving his/her's own teaching, just as important to the scholar is the aim of communicating the findings to other scholars within the discipline, in the hope of improving teaching across the discipline. In this paper, the author reports his research findings from an analysis of one of his exam papers.

The exam was sat by over 300 students, who had just completed their first semester of programming. In the original exam paper, there were 26 multiple choice questions, all focused upon aspects of programming in Java. Approximately half of those questions examined object-oriented concepts, and questions of that type have been analyzed in an earlier publication (Lister, 2005).

This paper analyses the student performance on 12 multiple choice questions, which focus on the classic imperative programming concepts of selection, iteration, and arrays. These 12 questions in turn break into two categories of questions: "Type A" and "Type B".

This quality assured paper appeared at the *20th Annual Conference of the National Advisory Committee on Computing Qualifications* (NACCQ 2007), Nelson, New Zealand. Samuel Mann and Noel Bridgeman (Eds).

2 The Four "Type A" Questions

Of the twelve multiple choice questions examined in this paper, four were Type A questions. These questions specified a short piece of code, and students were asked to determine the value in a particular variable, after the code had finished executing.

While it is not strictly necessary, these short pieces of code in the author's exams tend to be "nonsense" code. That is, the code does not usually perform a function that an experienced programmer would recognise. Nonsense code is the most expedient code to use for Type A questions (particularly after several semesters of setting these exams) as it is essential that students have not seen these pieces of code before – otherwise there is a danger that a student might happen to remember the answer (from a tutorial) without being able to compute the answer.

These questions test the students in two ways:

- The Type A questions test whether students understand basic programming constructs, particularly selection, iteration and arrays.
- The Type A questions test whether a student has the determination and focus required to manually execute code (i.e. "trace", or "desk check" code). The author of this paper explicitly teaches his students a methodical approach to walking through code, using tables like those shown in the subsections below. The tables shown below were provided in the exam paper given to the students.

The following four subsections provide a complete description of each of the four Type A questions. These four subsections contain nothing but the actual exam questions (except that the tables shown have been abbreviated, to save space).

2.1 Type A Question 1

Consider the following code:

```
int[] x = {1, 2, 3, 4, 5};
int a = 3;
int b = 0;
int c = 0;
```

```
while ( (c<a) && (b<x.length) )
{
   ++b;
   c += x[b];
}
System.out.println(b);
```

What value will be outputted by this code? You may use the table below to help you calculate the answer to the question.

a) 0
b) 1
c) 2
d) 3

Code	Comment	b	c
		0	0
… approx. 30 rows provided in the full table …			

2.2 Type A Question 2

Consider the following code fragment:

```
int[] x1 = {1, 2, 4, 7};
int[] x2 = {1, 2, 5, 7};
int i1 = x1.length-1;
int i2 = x2.length-1;
int count = 0;

while ((i1 > 0 ) && (i2 > 0 ))
{
      if ( x1[i1] == x2[i2] )
      {
            --i1;
            --i2;
      }
      else if (x1[i1] < x2[i2])
      {
            ++count;
            --i2;
      }
      else
      {     // x1[i1] > x2[i2]
            --i1;
      }
}
```

After the above while loop finishes, "count" contains what value? You may use the table below to help you calculate the answer to the question.

a) 3
b) 2
c) 1
d) 0

Code	Comment	i1	i2	count
				0
… approx. 30 rows provided in the full table …				

2.3 Type A Question 3

Consider the following code fragment.

```
int[] x = {0, 1, 2, 3};
int temp;
int i = 1;
int j = x.length-1;

while (i < j)
{
      temp = 2*x[i];
      x[i] = x[j];
      x[j] = temp;
      i++;
      j--;
}
```

After this code is executed , array x contains what values? You may use the table below to help you calculate the answer to the question.

a) {3, 2, 1, 0}
b) {0, 3, 2, 2}
c) {0, 4, 2, 2}
d) {0, 6, 2, 1}

Code	Comment	i	j	temp				
		1						
… approx. 30 rows provided in the full table …								

2.4 Type A Question 4

Consider the following code fragment.

```
int[] x = {1, 2, 2, 2, 1, 1};
int count = 0;
int i = 0;
int j = x.length/2;

while  ( j < x.length )
{
   if ( x[i] == x[j] ) ++count;
   else
   if ( x[i]  < x[j] ) --i;
   else
   if ( x[i]  > x[j] ) --j;

   ++i;
   ++j;
}
```

After this code is executed, the variable "count" contains what value? You may use the table below to help you calculate the answer to the question.

a) 4
b) 3
c) 2
d) 1

Code	Comment	i	j	count
		0		0
… approx. 30 rows were provided in the full table …				

2.5 Discussion of Type A Questions

For these Type A questions, 208 students (62%) scored a perfect 4. Hand executing code is tedious and error prone at the best of times, so the 208 students who did so perfectly, under exam conditions, demonstrated not just their knowledge of the programming concepts, but also their considerable commitment and capacity to attend to detail – excellent qualities in an aspiring programmer.

The correct answers to the four Type A questions are options c, c, b and c respectively. While option c is over represented in these four questions, across the full 26 questions in the entire exam, the correct answers were distributed almost evenly among the options. Furthermore, student responses were distributed almost evenly among the options across all 26 questions. Therefore, it is unlikely that the percentage of students who answered these four questions correctly is distorted by students using common multiple choice guessing heuristics, such as "when in doubt, choose option C".

3 "Type B" Questions; three in detail

The eight multiple choice Type B questions are intended to test students on their capacity to reason at a higher level than merely hand executing code. These Type B questions differ from Type A questions in three ways:

- Students are not required to specify the value in a variable, as they are in Type A questions. Instead, students are required to correctly identify lines of code that have been omitted from the complete code.
- The code in Type B questions perform functions that an experienced programmer would recognise. Algorithms represented in the Type B questions include basic sorting and searching algorithms, such as bubble sort, some other quadratic sorts, linear search, and binary search.
- The code in these questions has been seen by the students prior to the exam. All these algorithms were taught in class. Furthermore, the eight Type B questions in the exam were drawn from a pool of 30 questions, which were used in tutorial exercises as part of the learning process. Students were told that eight questions from this pool would appear – unaltered – in the exam.

Note that students were not required to rote-learn the algorithms taught. In fact, to discourage rote-learning, the students were provided in the exam with all the diagrams from lecture notes that explained the algorithms (and students were aware of this prior to the exam). A sample of these diagrams is given in the appendix. In total, there were 101 such diagrams provided in the exam. Any non-novice programmer who had never encountered these algorithms before would have been able to deduce the answers to the multiple-choice questions from these detailed diagrams.

Space limitations do not allow for a full exposition of all eight Type B questions. Instead, the following subsections describe three of the questions.

3.1 Adding an Element to a Set

A number of algorithms studied during semester related to storing the elements of a set in an array. The elements, all positive integers, are stored in ascending order in an array, with the end of the set indicated by the "sentinel" value of minus one (declared as a constant). For example, an array declared and initialized as:

```
int s[] = {2, 4, 6, -1, 1, 7};
```

contains the set {2, 4, 6}, with the last two positions in the array not forming part of the set.

The following "skeleton code" was studied during semester, and was supplied in the exam. The skeleton code of the method "`AddElementToSet`" code partially describes the addition of an element "e" to the set "s", returning false if there is not room for the new element. Before studying the three Type B multiple choice questions that follow, readers might attempt to deduce the missing lines for themselves, using the slides in the appendix.

```
class Sets {
/* This is a skeleton program for
   various set operations, where sets
   are implemented as a sorted array
   (ascending order) terminated by a
   sentinel value.
 */

// The "sentinel" terminates the
// sorted values in a set.
static final int sentinel = -1;

   ...

public static boolean AddElementToSet(
           int e,      // to be added
           int s[]) {  // to this set

/* Like the name says, this function
 * adds "e" to set "s". Element "e" is
 * added in its correct position, so
 * that the sorted order of the array
 * is maintained.  This implies those
 * elements larger than "e" are pushed
 * up one place to make room for "e".
 * If "e" is not already in the set
 * "s", and "s" is full, then the
 * function returns false; otherwise
 * it returns true.  Remember that all
 * elements in the array are unique.
 */

// Can't have the sentinel in the set
if ( e == sentinel ) return false;

// First try to find "e" in the set,
```

```
// or find where it belongs
int pos = 0;
while ( (xxx1xxx) && (xxx1xxx) )
     xxx2xxx;

if ( xxx3xxx )  // if "e" is in set
   return true; // success by default.

// Find the "last" position
// i.e. the position of the sentinel
int last = pos;
while ( xxx4xxx ) xxx5xxx;

// At this point in the code, the
// array element pointed by "last"
// contains the sentinel.

// Now check if there is room for the
// new element
if ( xxx6xxx ) return false;

// The remaining code adds "e"

// First, push up one place all
// elements bigger than "e"
for (int i=last ; xxx7xxx ; xxx7xxx)
   xxx7xxx;

// Finally, put the new element into
// its correct place in the array
xxx8xxx;
return true;

} /* AddElementToSet */
```

The final three questions in the exam required students to identify some of the above missing code. These questions are described in each of the next three subsections.

3.2 Type B Question 10

Skeleton code is provided for a method "`AddElementToSet`" in the "Sets" class. The skeleton code contains:

```
while ( xxx4xxx ) xxx5xxx;
```

The correct completion of this line is:

(a) while `(s[last] != s[pos]) ++last;`
(b) `while (last != sentinel) ++last;`
(c) `while (last != pos) ++last;`
(d) `while (s[last] != sentinel) ++last;`

3.3 Type B Question 11

This questions follows on from the previous question, on "`AddElementToSet`". The skeleton code contains:

```
if ( xxx6xxx ) return false;
```

The correct completion of this line is:

(a) `if ( s[last] == s[s.length-1] )`
(b) `if ( last == s.length-1 )`
(c) `if ( last == s.length )`
(d) `if ( s[last] == s[s.length] )`

3.4 Type B Question 12

This question follows on from the previous two questions, on "`AddElementToSet`". The skeleton code contains:

```
for (int i=last; xxx7xxx; xxx7xxx)
   xxx7xxx;
```

The correct completion of this line is:

(a)
```
for (int i=last; i<=pos; ++i)
   s[i+1] = s[i];
```

(b)
```
for (int i=last; i>=pos; --i)
   s[i] = s[i+1];
```

(c)
```
for (int i=last; i<=pos; ++i)
   s[i] = s[i-1];
```

(d)
```
for (int i=last; i>=pos; --i)
   s[i+1] = s[i];
```

4 Results: The Middle Students

Some students did very well on the exam. In fact, 29% of the class achieved a perfect score on both Type A and Type B questions. There was also a group of students who did poorly on the exam, in both Type A and Type B questions.

The data, however, shows that there is also a middle group of students. These students are among the 208 students who scored a perfect 4 on Type A questions, but were not particularly successful at answering the Type B questions. On each of the three Type B questions describe above, between 10-20% of those 208 students answered incorrectly. Figure 1 shows the performance on each Type B question for those 208 students. Table 1 provides a breakdown, for the three Type B questions described above, of the incorrect options chosen by this 10-20% of the 208 students.

Table 2 shows the cumulative percentage of Type B questions correct among the students who scored a perfect 4 on Type A questions. That table shows that approximately half of these students scored 7 or less out of 8 on the Type B questions. Of course, we all make simple mistakes, especially under exam conditions (although recall that these these students did not make any mistakes in answering the Type A questions) so we may regard 7 questions correct out of 8 as a good score; perhaps even 6 out of 8 is a good score. However, the 19% of these 208 students (i.e. 40 students) who score 5 or less on these Type B questions are manifesting an unambiguous weakness at answering these types of questions, despite (as discussed earlier) the benefit of having:

- Seen these 8 questions in a pool of 30 prior to the exam.
- Access in the exam to diagrams illustrating the algorithms.

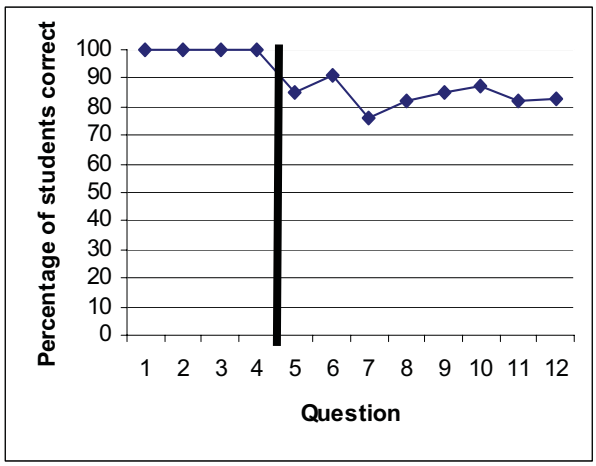


Figure 1: The percentage of Type B questions correct among the students who scored a perfect 4 on Type A questions. (N=208). The questions to the left of the vertical bar are the four Type A questions. The questions to the right of the vertical bar are the eight Type B questions.

Table 1: The number of incorrect responses to the three Type B questions among the 208 students who scored a perfect 4 on Type A questions. Asterisks indicate the correct responses for those questions.

Option	Q10	Q11	Q12
a	8	21	16
b	15	***	12
c	4	15	5
d	***	2	***
Total	27	38	33

Table 2: The cumulative percentage of Type B questions correct among the students who scored a perfect 4 on Type A questions (N=208).

0	1	2	3	4	5	6	7	8
0	1	4	9	13	19	30	53	100

5 Discussion: The Middle Students

Given that students had been warned that eight Type B questions from the pool of 30 would appear in the exam, and students were provided in the exam with the diagrams like those shown in the appendix, one might expect that students would do very well on the eight Type B questions – especially the 208 students who scored a perfect 4 on Type A questions. How do we explain why they did not answer these questions correctly?

First, we can discard some of the common "staff room" explanations of student behaviour, at least for the 208 students who answered Type A questions correctly. Those students cannot be described as lazy, sloppy, or lacking in commitment.

5.1 An Explanation from the Literature

There have been studies across many disciplines into the differences between novices and experts (Chi, Glaser & Farr, 1988; Ericsson & Smith, 1991). That research indicates experts organize their knowledge into more abstract forms than novices. This is apparent in the classic studies of chess players (Chase & Simon, 1973). When asked to memorize board positions of several chess pieces, novices tended to remember the position of each piece in isolation, whereas experts organized the information at a more abstract level, the attacking and defensive combinations.

In a study of programming that reflected the earlier chess studies, Adelson (1984) showed that, when given typical tasks on well-written code, experts outperformed novices, but when faced with unnatural tasks, novices sometimes outperformed the experts – the explanation being that when code is well-written and the tasks are natural, experts are able to reason about code at a more abstract level than novices.

During the 1980s, Elliot Soloway led a movement to focus programming pedagogy upon "plans" or "schema" (Soloway, 1986; Rist, 2004), and not on programming language constructs. Soloway wrote:

> *...language constructs do not pose major stumbling blocks for novices... rather, the real problems novices have lie in "putting the pieces together," composing and coordinating components of a program.* (Soloway, 1986, p. 850)

Results from the BRACElet project, which has analyzed data collected in exams sat by New Zealand students, is consistent with the above literature. In one BRACElet study, students were asked to "explain in plain English" what a short piece of code did. Most students provided relatively concrete explanations (Whalley *et al*., 2006; Lister *et al*., 2006). In another BRACElet study, students were asked to identify similarities in four code segments, which all found either the minimum or maximum in an array of numbers. Many students identified syntactic similarities, but failed to identify the more abstract functional similarities (Thompson, *et al*., 2006).

5.2 Bizarre Moments in Teaching Explained

In the light of the above literature, it appears then that the middle students identified in this study of an exam paper are accomplished concrete reasoners about code (as evidenced by the perfect performance on the Type A problems) but have not (at least at this stage) developed a capacity to reason about code at any higher level of abstraction.

The identification of these students explains some of the more seemingly bizarre moments in the teaching of programming to novices. Every teacher of novice programmers, including the author of this paper, has stories from the pedagogic "twilight zone", such as:

- Students who attempt to debug code, sometimes for hours on end, by "random mutation".

- Students who introduce new bugs as they attempt a superficial and incorrect fix to an existing bug.
- Students who come to the teacher for help with a bug, saying that they have worked for hours to find it, when the bug is glaringly obvious to the teacher.
- Students who cannot explain their own code (in cases where the teacher discounts the possibility of cheating).

6 Conclusion

Most computing academics took to programming "like ducks to water". They happened to have the mental orientation that allowed them to reason about programs at an abstract level without having to be explicitly taught to do so. Academics barely notice the minutia of code, but instead read and understand code at a more abstract level. We see the intent of the code, the plan, or "schema".

Academics who took to programming "like ducks to water" fail to appreciate the high cognitive load in reading and understanding code for many novice programmers. Such academics routinely despair at the many students who cannot even reproduce, in an exam, short pieces of code that were taught during semester. Teachers often dismiss such students as either lacking the "knack" for programming, or lacking commitment. Undoubtedly, there are students who lack these attributes, but the results in this paper suggest that we should not dismiss all struggling students in these ways. A majority of students who sat this exam scored perfectly on the Type A questions. While those Type A questions only require a concrete understanding of code, consistently answering those questions correctly requires commitment and attention to detail, especially when we consider that the questions were being answered under exam conditions. However, among the students who consistently answered Type A questions correctly, approximately 20% demonstrated a weakness on the Type B questions. These Type B questions require the student to reason about code at a higher level of abstraction. It is this 20% of students who are strong at concrete reasoning but weak at abstract reasoning that we refer to as the middle novice programmer. These novices do not see intent, plans, or schema − they just see code.

Although these middle novice programmers may not automatically learn to reason about programs, perhaps they can learn from explicit instruction. For example, de Raadt, Toleman & Watson (2004) have described their approach to explicitly teaching schemas. Another approach is to explicitly teach students the roles of variables (Kuittinen & Sajaniemi, 2004).

While we do advocate a greater emphasis on explicitly teaching students to see the "ghost in the program", we also advocate that students be taught and rigorously assessed on the low-level skill of tracing through code. Very recent experimental results indicate that students cannot learn to reason about code abstractly unless they also acquire "*complete mastery of the code tracing task*" (Philpott, Robbins and Whalley, 2007).

Computing academics have been teaching programming, in ways largely unchanged, for decades. Teaching techniques are unlikely to change when our teaching lives are compartmentalized from our research lives. This paper demonstrates how the scholarly approach to analysing something as mundane as an exam paper can lead to fresh perspectives into teaching.

7 References

Adelson, B. (1984) When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, **10**(3): 483-495.

Chase, W. C., & Simon, H. A. (1973) Perception in chess. *Cognitive Psychology*, **4**: 55-81.

Chi, M. T. H., Glaser, R. & Farr, M. J. (Eds.) *The nature of expertise*. Hillsdale, NJ, Lawrence Erlbaum Associates, 1988.

de Raadt, M., Toleman, M., and Watson, R. (2004): Training strategic problem solvers (2004): *SIGCSE Bulletin*, **36**(2, June): 48-51.

Ericsson K, and Smith, J. (Eds) *Toward a General Theory of Expertise : Prospects and Limits*. Cambridge University Press,England, 1991.

Kuittinen, M, and Sajaniemi, J. (2004): Teaching Roles of Variables in Elementary Programming Courses. *Proc. of the ACM ITiCSE International Conference on Innovation and Technology in Computer Science Education*, Leeds, UK, 57–61.

Lister, R. (2005) One Small Step Toward a Culture of Peer Review and Multi-Institutional Sharing of Educational Resources: A Multiple Choice Exam for First Semester Programming Students. *Seventh Australasian Computing Education Conference* (*ACE2005*). Newcastle. Jan 31 – Feb 3. pp. 155-164. http://crpit.com/confpapers/CRPITV42Lister.pdf [Accessed June 2007]

Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on innovation and Technology in Computer Science Education.* (Bologna, Italy, June 26 - 28, 2006). ITICSE '06. ACM Press, New York, NY, 118-122.

Philpott, A., Robbins, P., and Whalley, J. (2007) Assessing the Steps on the Road to Relational Thinking. . In *Proceedings of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications*, NACCQ, Nelson, New Zealand, July 8-11.

Rist, R.(2004): Learning to Program: Schema Creation, Application, and Evaluation. In *Computer Science Education Research*. Fincher, S and Petre, M. (eds). Routledge Falmer.

Soloway, E. (1986): Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM.* **29**(9): 850 - 858.

Thompson, E., Whalley, J., Lister, R., Simon, B. (2006) Code Classification as a Learning and Assessment Exercise for Novice Programmers. In *Proceedings of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications*, NACCQ, Wellington, New Zealand, July 7-10. pp. 291-298.

Whalley, J, Lister, R, Thompson, E, Clear, T, Robbins, P, Prasad, C (2006) An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Australian Computer Science Communications* 52: 243-252.

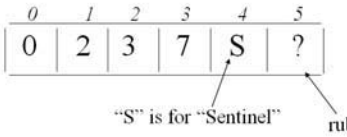
8 Acknowledgements

The author thanks his collaborators on the BRACElet project, including Jacqueline Whalley, Errol Thompson, Beth Simon, and Tony Clear.

Appendix

The following slides were used during lectures to teach the algorithms. These slides were also provided to students in the exam.

Our way of representing sets in arrays

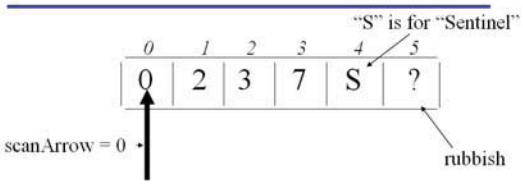


Note:

1. The elements of the array are in ascending order.
2. The use of a sentinel value. The final position of the array is not in use. This is indicated by the Sentinel value (a special **integer** value, not char) in the previous position of the array. "S" is NOT part of the set! "S" indicates the end of the set.

05/18/99

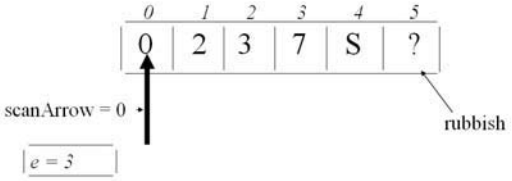
Found/Linear Search: The movie (1)



Here we show the initial conditions for a search. The scanning arrow is initialized to "point" to the first element in the array.

05/18/99

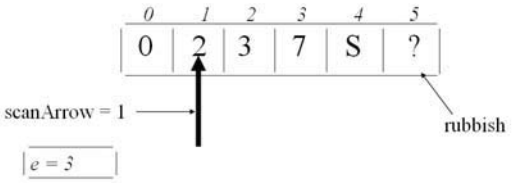
Found/Linear Search: The movie (2)



Suppose we are looking for the number 3. Position 0 of the array does not contain 3. Furthermore, 0 is less than 3, so the search should continue to scan along the array ... *next slide ...*

05/18/99

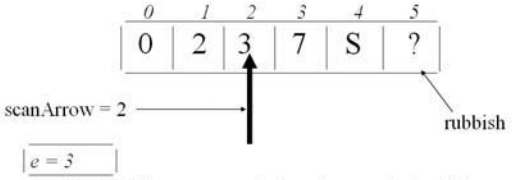
Found/Linear Search: The movie (3)



Position 1 of the array is less than 3, so the search should continue to scan along the array ... *next slide ...*

05/18/99

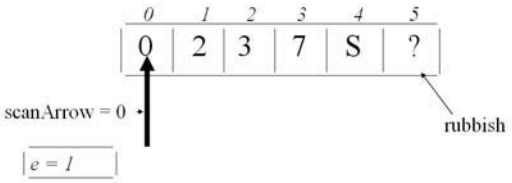
Found/Linear Search: The movie (4)



Position 2 of the array equals 3, so the search algorithm has found the target element. **Our method returns 2, the position of the target element in the array.** Linear search is a little more complicated if the target is not in the array ... *next slide ...*

05/18/99

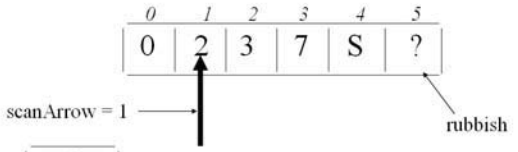
Found/Linear Search: The movie (5)



This slide is the commencement of a **brand new search, for a target that is not in the array.** The search begins the same as in the previous example. Position 0 of the array is less than 1, so the search should continue to scan along the array ... *next slide ...*

05/18/99

Found/Linear Search: The movie (6)

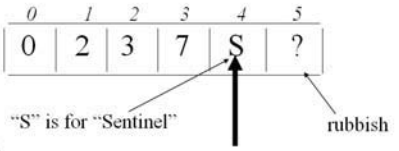


e = 1

Position 1 of the array is greater than 1. **Since the elements of the array are in ascending order**, **the search can stop immediately**. **Our method returns -1** when it cannot find the target in the array. This is not the only situation that can arise when searching for a target that is not in the array ... *next slide ...*

05/18/99

Found/Linear Search: The movie (7)

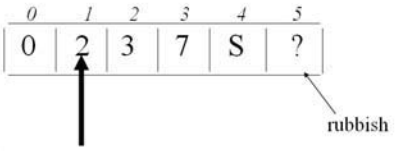


e = 8

In this scenario, the target is not only not in the array, but it is bigger than all elements in the array. **When the scanArrow "points" to the sentinel value, the search should stop.** As before, the method returns -1.

05/18/99

"AddElementToSet": The Movie (1)



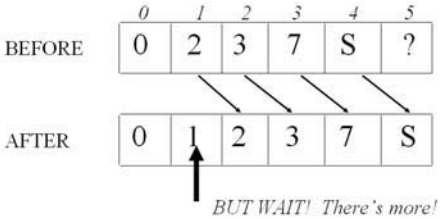
e = 1

Suppose we want to add e=1. First we have to find where "e" belongs in the array. That's easy ... we use the same kind of Linear Search code as we used in "Found".

05/18/99

"AddElementToSet": The Movie (2)

Having found where "e" belongs, we now push up one place all elements bigger than "e" (and also push up the sentinel).

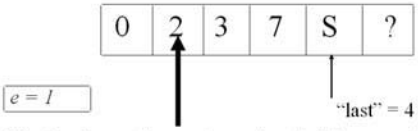


BUT WAIT! There's more! Next slide ...

05/18/99

"AddElementToSet": The Movie (3)

*Note: the "push up" loop has to run "**backwards**", from the sentinel back towards the place where "e" belongs. But the only way we can find the sentinel is to scan forward! So there will be two loops, one after the other.*



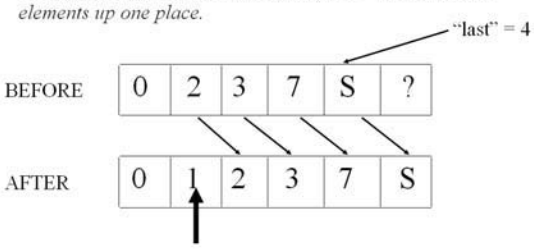
e = 1

The first loop will scan forward to find the sentinel. (We're catching a different rabbit, but its still a rabbit, so we'll use linear search). The second loop will ... see next slide ...

05/18/99

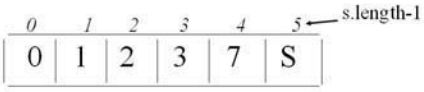
"AddElementToSet": The Movie (4)

The second loop will run backwards from "last", pushing elements up one place.



05/18/99

AddElementToSet: A special case



What if the array "s" is full?

That's why "AddElementToSet" returns a boolean ... If "s" is full, the set is left unchanged and "false" is returned.

Under all other circumstances, "AddElementToSet" returns "true".

05/18/99