# Data Structures 2020 Assignment 2

Publish date: 02/04/2020

Due date: 30/04/2020

Senior faculty referent: Prof. Paz Carmi

Junior faculty referents: Stav Ashur, Elad Sulami

## Assignment Structure

0. Integrity statement
1. Backtracking data structures and algorithms – **Explanatory section**
2. Backtracking algorithms – **Programming "warm-up" section**
3. Backtracking Dynamic Set ADT – **Programming task section**
    i. Unsorted array implementation
    ii. Sorted array implementation
    iii. BST implementation
    iv. Undo-Redo BST implementation
4. Runtime complexity analysis of chapters 2-3 – **Theoretical task section**

## Important Implementation Notes

- It is strongly recommended to read the entire assignment at least twice before you start writing your solutions (code **and** proofs).
- The assignment may be submitted either by pairs of students or by a sole student. However, you are encouraged to submit in pairs.
- You may **not** use generic data structures implemented by others (the developers of Java, Git projects and so on).
- Your code should be neat and well documented.
- When testing your code, you may use whatever tools you want, including classes and data structures created by others. The restriction above applies only to the code you submit to us.
- Your implementation should be as efficient as possible. Inefficient implementations will receive a partial score depending on the magnitude of the complexity.
- You may assume that your code will only be tested with proper inputs.
- Your code must not print any output that was not specifically requested in the exercises
- As you have learned, in this course in general and specifically in this assignment, the analysis of runtime complexity is always a worst-case analysis.
- Your code will be tested in in computers using Java 8 and without any external packages. You must make sure that your code compiles and runs in such an environment. Code that will not compile or run **will receive a grade of 0**.
- Don't forget to sign and submit the statement in section 0. Your code will be checked for plagiarism using automated tools and manually. The course faculty, CS department and the university regard plagiarism with all seriousness, and severe actions will be taken against anyone that was found to have plagiarized. A submitted assignment without a signed statement **will receive a grade of 0**.
- You must submit a single zipped folder (.zip) containing all of your Java files, integrity statement, and answers to the theoretical exercises. The zipped folder should not contain any subfolders. Additional folders or inside the zipped file may cause the automated test to give you a grade of 0 which will **not** be changed by appeals.

## Section 0: Integrity Statement

Sign this statement and submit it as exercise 0.

I assert that the work I submitted is entirely my own.

I have not received any part from any other student in the class, nor did I give parts of it for others to use.

I realize that if my work is found to contain code that is not originally my own, a formal case will be opened against me with the BGU disciplinary committee.

| | |
|---|---|
| X _____ | X _____ |
| Name and signature | ID number |
| X _____ | X _____ |
| Name and signature | ID number |

# Section 1: Backtracking Data-Structures and Algorithms

An algorithm is a series of pre-determined steps that when preformed on certain objects (the input) achieve a desired result (output). A data structure is a method of organizing data in the memory of a computer in order to efficiently use it. The algorithms that you have seen so far (insertion sort, bubble sort, binary search etc.) always "progress" in a certain direction and never "regret" a decision that they made. For example, after the binary search algorithm decides to search in a certain half of the input, it will never "regret" and re-evaluate that decision. In the same way, the data structures you have seen can only delete existing data, and insert new data. There is no way to regret adding or deleting an object. (Notice that the outcome applying pairs of *insert(S,x)* and *delete(S,x)* or vice versa on a data structure, is not necessarily the same as not inserting *x*, meaning that, for example, deleting an object is not the same as cancelling its insertion.)

However, in more advanced algorithms that can receive dynamic data structures as an input (a data structure that might change during the course of the algorithm), the ability to undo several steps is sometimes needed, and that ability is made possible by backtracking data structures. Also, many data bases exist for which backtracking capabilities are crucial in order to correct faulty operations or cancel actions that a user regretted. Such data-bases are used by banks, back-up services and more.

The most well-known backtracking action is the one enabling users to hit "Ctrl+z" on their PC and undo the last action. This is made possible by a stack operating in the background, storing the actions performed by the user in LIFO (**L**ast **I**n **F**irst **O**ut) order.

**Definitions**:

A *modifying operation* is a method of an ADT which changes the data stored within the data structure in any way. For example, the *push(S)* method of the stack ADT is a modifying operation, and the method *minimum(S)* of the dynamic set ADT is **not** a modifying operation.

A backtracking dynamic data structure ADT *S* is a data structure ADT with the additional method:

*backtrack(S): cancels the last modifying operation performed by the data structure*

For example, a backtracking stack ADT has the methods *create(S), push(S,x), pop(S), isEmpty(S), and backtrack(S).* The only modifying operations in this list of methods are *push()*, and *pop()* since they are the only actions that change the data.

## Stack Class

In your code you **must** use the class Stack found in the source files of the assignment. Stack is implements the standard stack ADT interface:

*bool isEmpty()*

*void push(x)*

*Object pop()*

And another method that empties the stack:

*void clear()*

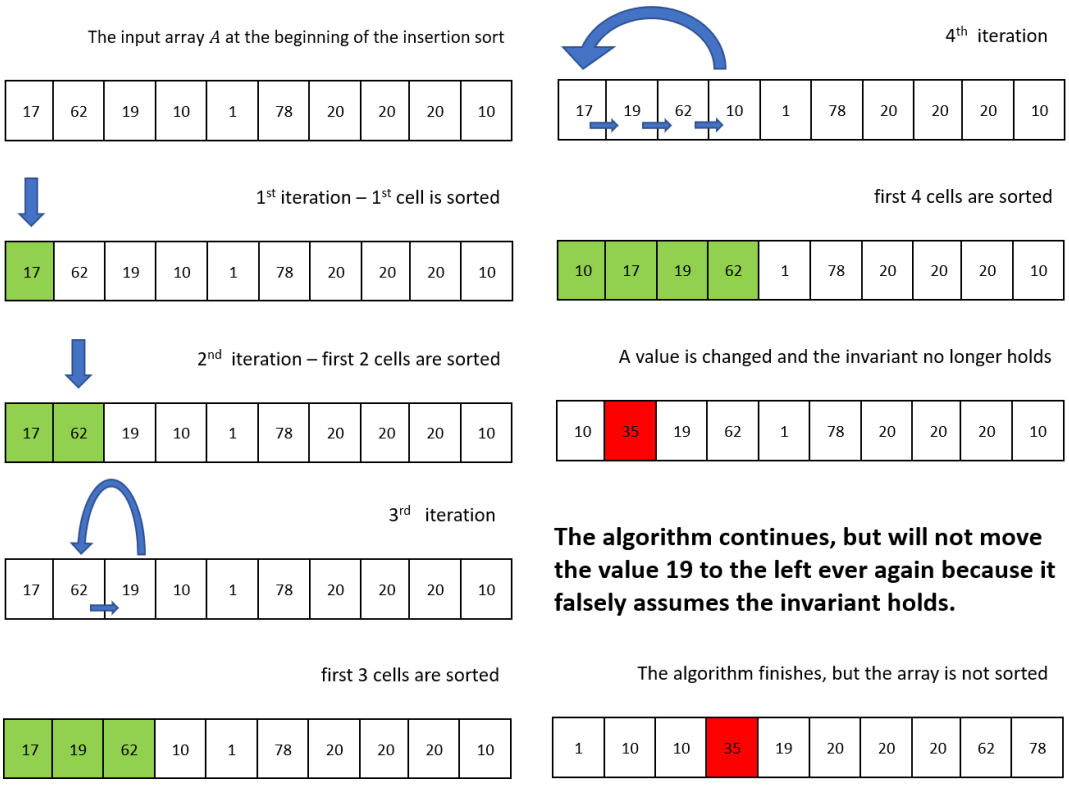
All methods are implemented efficiently and have a runtime bounded by $O(1)$.

# Section 2: Backtracking Algorithms

**Consistent Algorithms**

As mentioned in Section 1, backtracking algorithms are used in cases where the input given to the algorithm might change during the algorithm's execution. These scenarios are problematic since many algorithms depend on a specific property or invariant (טענה נשמרת) that must hold at all times, and a change in the input might cause the algorithm to lose one or more of its properties or make the invariant false.

For example, the insertion sort algorithm maintains the invariant that at the $i^{th}$ iteration, the first $i$ cells in the array are sorted. If at some point during the execution of the algorithm, one of the considered cells will receive a new value, this invariant might not hold anymore, and the result of the algorithm will therefore be fallacious. (See an illustration in the figure below.)



**Definition:**

We say that an algorithm is **consistent** if it has all of its defined properties, and all of its invariants hold. The insertion sort algorithm depicted above was consistent at the start of iterations 1-4, but inconsistent starting at the start of the $5^{th}$ iteration and until its last step.

**Notice:**

In each of the implementations of the functions in this section you may use only one instance of the Stack class that is given to the function as an input, and $O(1)$ additional space. Moreover, a search that did not find the desired index should return -1.

**Warm up exercises:** (These are mandatory and will be graded)

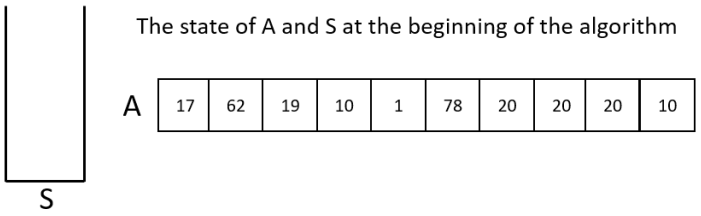
These exercises are given in order to allow you to gain some experience with implementing backtracking algorithms using the stack ADT. So, even though it is possible to implement the functions without a stack, you are required to use a stack for backtracking.

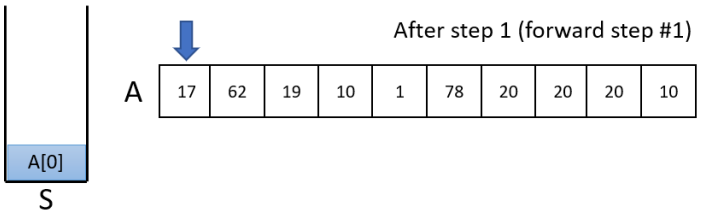
a. Implement the function
***public int backtrackingSearch(int[] arr, int x, int fd, int bk, Stack myStack)***
That receives an **unsorted** array of integers **arr** and searches for the index of the first occurrence of the value ***x*** with the added property that after every ***fd*** many search steps (steps are defined below), backtracks (undoes) ***bk*** many steps back. The algorithm stops if it finds the needed index or reaches the and of the array.

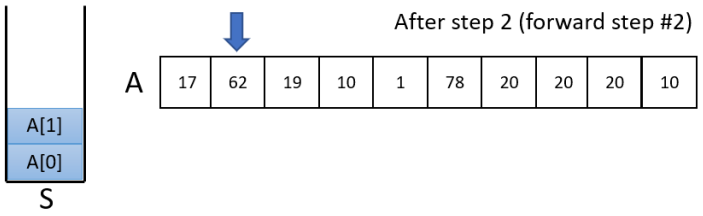
Step:
In this algorithm, a step of the search is an iteration of the main loop if the algorithm is implemented iteratively, or a recursive call if it is implemented recursively. See Appendix A for an example.

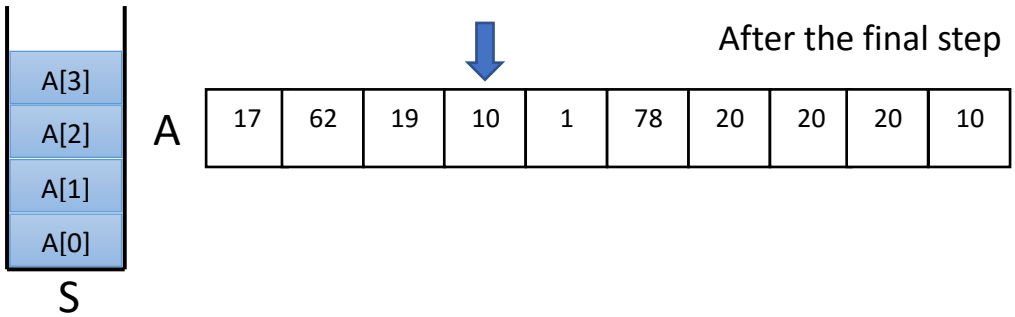
Invariant:
Every cell of ***arr*** that was accessed by the algorithm in previous steps does not contain the value ***x***.

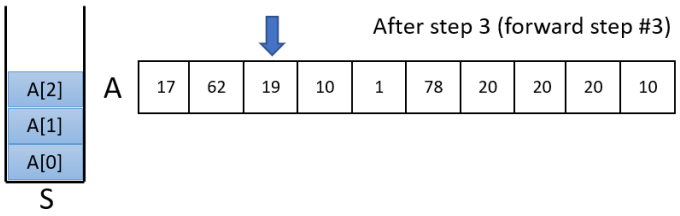
Example:
An illustration of the call *backtrackingSearch(A, 10, 3, 2)* with the array *A={17, 62, 19, 10, 1, 78, 20, 20, 20, 10}*. The stack *S* depicted in the illustration is holding the steps performed by the algorithm.

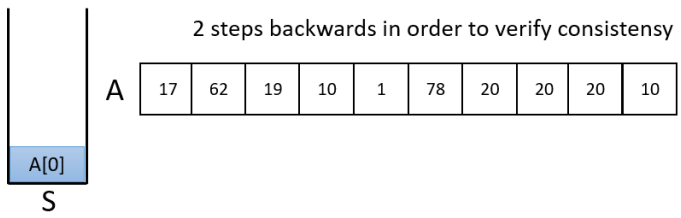

The state of A and S at the beginning of the algorithm
A
17 62 19 10 1 78 20 20 20 10
S

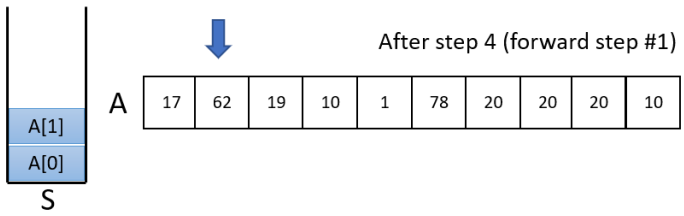

After step 1 (forward step #1)
A
17 62 19 10 1 78 20 20 20 10
A[0]
S


After step 2 (forward step #2)
A
17 62 19 10 1 78 20 20 20 10
A[1]
A[0]
S


After the final step
A
17 62 19 10 1 78 20 20 20 10
A[3]
A[2]
A[1]
A[0]
S


After step 3 (forward step #3)
A
17 62 19 10 1 78 20 20 20 10
A[2]
A[1]
A[0]
S


2 steps backwards in order to verify consistensy
A
17 62 19 10 1 78 20 20 20 10
A[0]
S


After step 4 (forward step #1)
A
17 62 19 10 1 78 20 20 20 10
A[1]
A[0]
S

b. Implement the function

***public int consistentBinSearch(int[] arr, int x, Stack myStack)***

That receives a **sorted** array of integers and searches for the index of the occurrence of the value *x* by using the binary search algorithm with the added property that before each step (a step is defined below), the algorithm checks the array for inconsistencies by calling the function:

***public static int isConsistent(int[] arr)***

That receives an array and returns the minimal number of steps which the algorithm must undo in order to become consistent. If the function returns 0, then the algorithm is currently consistent and may continue performing the next steps.

After checking for inconsistencies, your function will act accordingly by either backtracking, or moving on to the next step.

The function *isConsistent()* will be a part of the environment in which your code will be tested in, and you do not need to implement it or worry about its operation.

We recommend testing your code with the following function:

```
  public static int isConsistent(int[] arr) {
 double res = Math.random() * 100 - 75;

 if (res > 0) {
    return (int)Math.round(res / 10);
 } else {
    return 0;
 }
}
```

This function is a simple random function that operates in the following manner:

- Returns 0 ~80% of the time
- Returns 1 ~10% of the time
- Returns 2 ~10% of the time

The function (*isConsistent*()) written above **does not really checks for inconsistencies**. It only returns a random number. You do not need to worry about the operation of the real function your code will use during grading, and may assume it will return a valid output which is a non-negative integer that is not bigger than the number of steps performed by your algorithm.

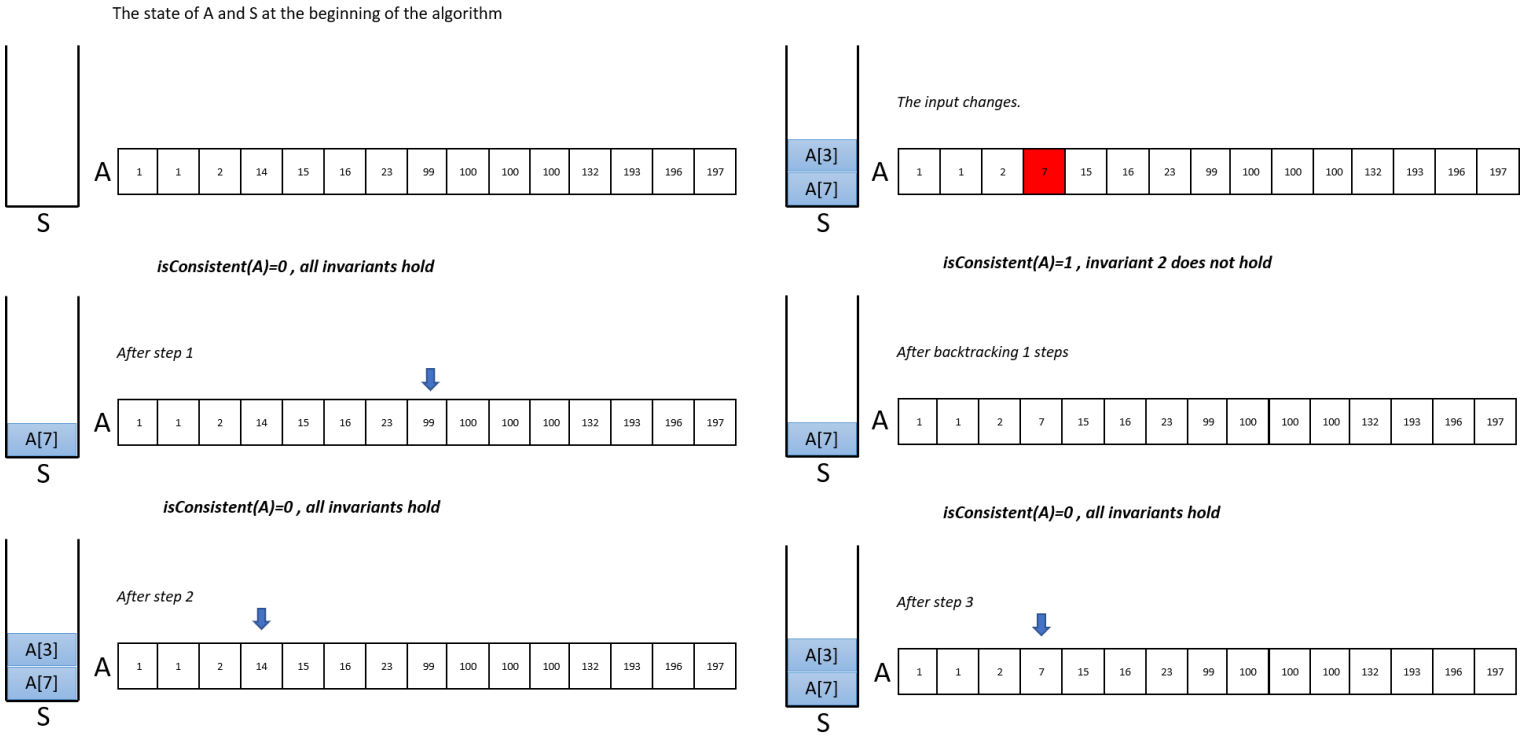
Step: (This is exactly the same as 1.a)
In this algorithm, a step of the search is an iteration of the main loop if the algorithm is implemented iteratively, or a recursive call if it is implemented recursively. See Appendix **B** for an example.

Invariants:

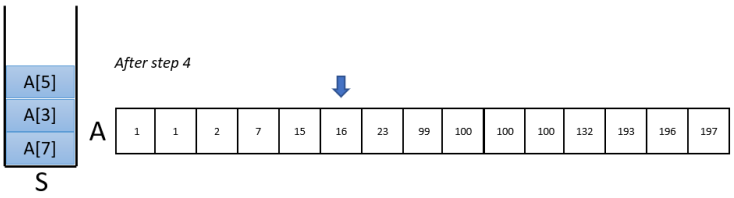
1. The array **arr** is sorted.
2. If at some step of the algorithm a cell $\boldsymbol{arr}[i]$ was accessed and compared with **x** and the comparison gave that $\boldsymbol{arr}[i]$ was smaller than $\boldsymbol{x}$, then for every $j \leq i$, $\boldsymbol{arr}[j] < \boldsymbol{x}$.
3. If at some step of the algorithm a cell $\boldsymbol{arr}[i]$ was accessed and compared with **x** and the comparison gave that $\boldsymbol{arr}[i]$ was bigger than $\boldsymbol{x}$, then for every $j \geq i$, $\boldsymbol{arr}[j] > \boldsymbol{x}$.

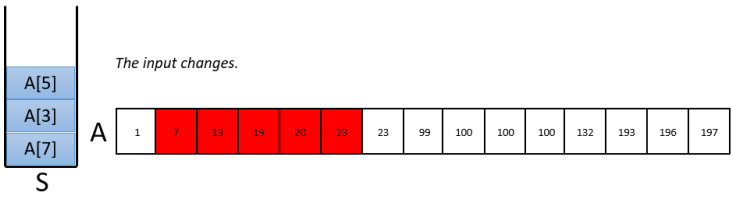
Intuitively, invariants 2 and 3 mean that all of the comparisons made by the algorithm so far are still valid.

Example:
An illustration of the call *consistentBinSearch(A, 13)* with the array *A={1, 1, 2, 14, 15, 16, 23, 99, 100, 100, 100, 132, 193, 196, 197}*. The stack *S* depicted in the illustration is holding the steps performed by the algorithm.

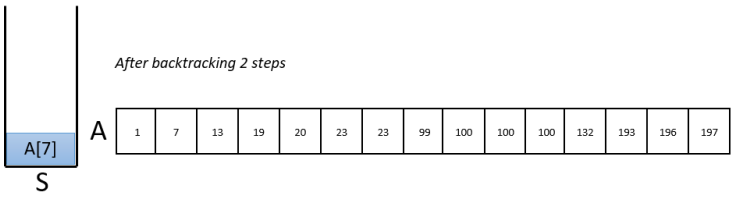


***isConsistent(A)=0 , all variants hold***

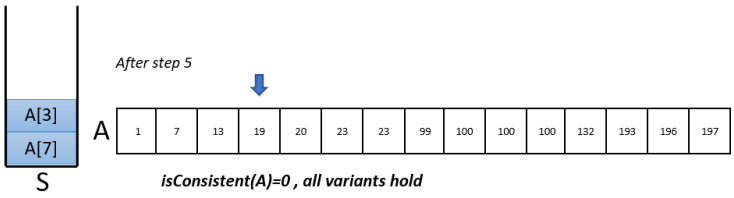


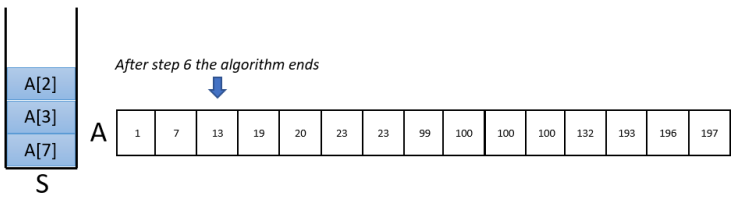

***isConsistent(A)=2 , invariants 2 and 3 do not hold***




***isConsistent(A)=0 , all invariants hold***




***isConsistent(A)=0 , all variants hold***



## Section 3: Backtracking Dynamic Set ADT

In exercises i-iii you are required to implement a backtracking dynamic set ADT using different underlying data structures. In your you may use only one instance of the Stack class and $O(1)$ additional space. In exercise iv you may use 2 instances of the Stack class and $O(1)$ additional space. The function ***print(S)*** is the exception.

Your implementations must include all methods of the dynamic set ADT :

| | | |
|---|---|---|
| ***search(S, k)*** | ***minimum(S)*** | ***predecessor(S, x)*** |
| ***insert(S, x)*** | ***maximum(S)*** | |
| ***delete(S, x)*** | ***successor(S, x)*** | |

And also:

***print(S)*** – Prints the values stored in the dynamic set. In array-based implementation the printed values should be separated by single spaces, and ordered by index from low to high (the value stored in the cell indexed 0 first, the value stored in the cell indexed 1 second, and so on), and in a BST-based implementation the printed values should be in pre-order. Examples follow. You may use additional $O(n)$ space for the implementation of this method. Notice that implementations that only use constant additional space exist and are in the scope of what you have already learned.

**Make sure that your function prints exactly as described in order to avoid grade reduction. We strongly recommend that you use the tests we provide in order to verify the correctness of the printing format.**

***backtrack(S)*** – This method should cancel the last *insert(S,x)* or *delete(S,x)* performed by the data structure and return the data-structure to **exactly the same** state prior to that action, and print "backtracking performed". This means that after backtracking, the data structure should look as if the backtracked action was never performed. If no *insert/delete* actions were performed by the data structure then the method should not change anything in the data structure.

**only for array-based implementations:**

***get(S, i)*** – returns the value stored in the underlying array at index *i*. Notice that this is not a part of the dynamic set ADT, and is required only for testing.

**only for BST-based implementations:**

***getRoot(S)*** – returns the root of the underlying BST. Notice that this is not a part of the dynamic set ADT, and is required only for testing.

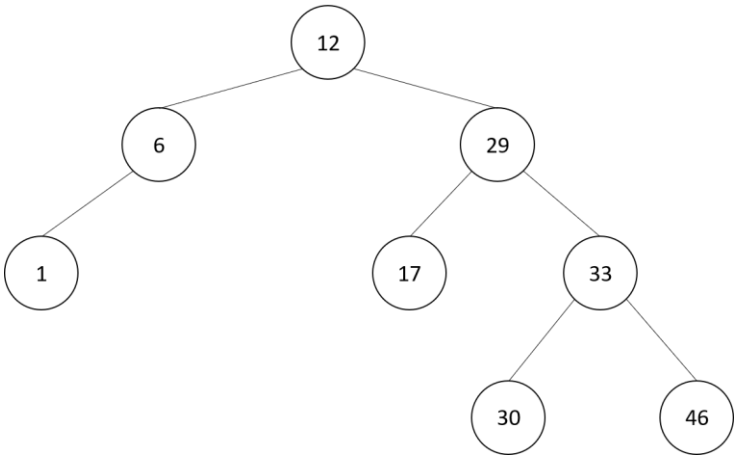
**Notice: The methods are described as if they take the data structure as an input. This is done for generality. In your Java implementation you must not declare that since Java implicitly does so for every method. For example, you must not write** *public void push(Stack S, int x)*. **but instead write** *public void push(int x)* **and use the standard** *S.push(x)* **form. Also, the method** *search(S, x)* **should return a tree node if the implementation is BST-based, and an index if the implementation is array-based.**

Printing examples:

print(S) output: “17 62 19 10 1 78 20 20 20 10”

| 17 | 62 | 19 | 10 | 1 | 78 | 20 | 20 | 20 | 10 |
|---|---|---|---|---|---|---|---|---|---|

print(T) output: “12 6 1 29 17 33 30 46”




**Notice:** The interface of a dynamic set is provided in the assignment files.

**Exercises:**

i. Implement the backtracking dynamic set ADT using an unsorted array as the underlying data structure.

ii. Implement the backtracking dynamic set ADT using a sorted array as the underlying data structure.

iii. Implement the backtracking dynamic set ADT using a BST as the underlying data structure. The nodes of the BST should contain the fields *key* and *value*. Although you will only use the *key* field in this assignment, the implementation should be general, and every node should contain a *null* object in its *value* field.

iv. Implement the double backtracking dynamic set ADT using a BST as the underlying data structure. This ADT has all of the methods of the backtracking dynamic set ADT, and also the method ***retrack(S)***.

Intuitively, you are required to implement the mechanism of the "undo" and "redo" as you know them from programs such as Word and PowerPoint for the dynamic set ADT.

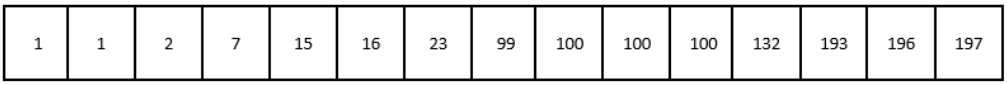
The method *retrack(S)* cancels the cancellation of the last *insert(S,x)* or *delete(S,x)* cancelled by a *backtrack(S)* action and returns the data-structure to its state prior to that backtracking action. This action can only be performed if the last modifying operation was a *backtrack(S)* action, and can only be performed *i* times consecutively if the last *i* modifying operations were *backtrack(S)* actions. After inserting or deleting an item in *S*, no *retrack(S)* action can be performed before an action is backtracked.

If no *backtrack(S)* actions were performed by the data structure after the last *insert*\delete action, then the method *retrack(S)* should not change anything in the data structure.

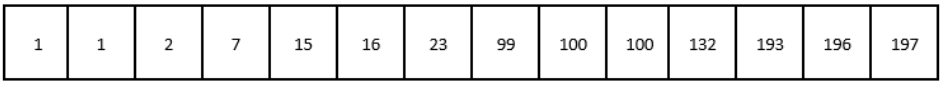
**Notice: for exercise** iv **you may use 2 instances of the stack ADT we provide and $O(1)$ additional space.**

**Example for a series of steps on a sorted array based backtracking dynamic set ADT.**

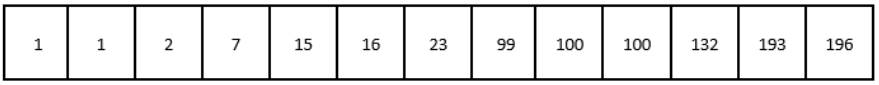
Action 1: *maximum(A)* Value returned: 14



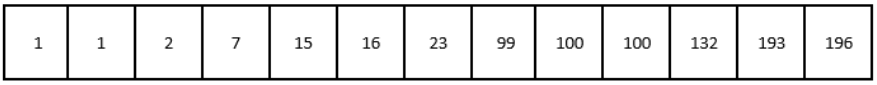

Action 2: *delete(A, 100)* Value returned: none



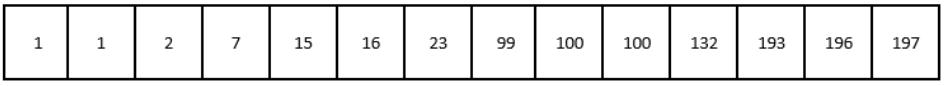

Action 3: *delete(A, 197)* Value returned: none



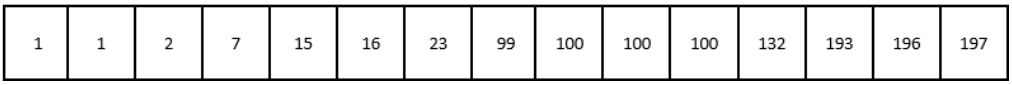

Action 4: *maximum(A)* Value returned: 12




Action 5: *backtrack(A)* Value returned: none

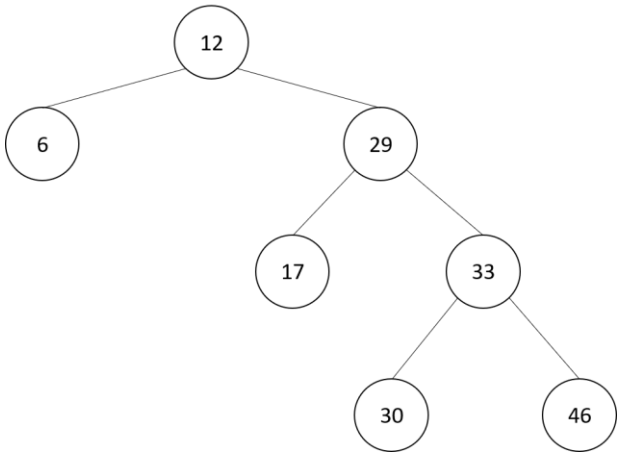



Action 6: *backtrack(A)* Value returned: none

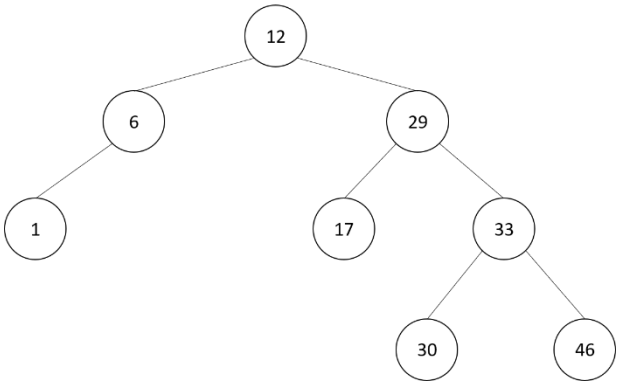


## Example for a series of steps on a BST based backtracking dynamic set ADT.

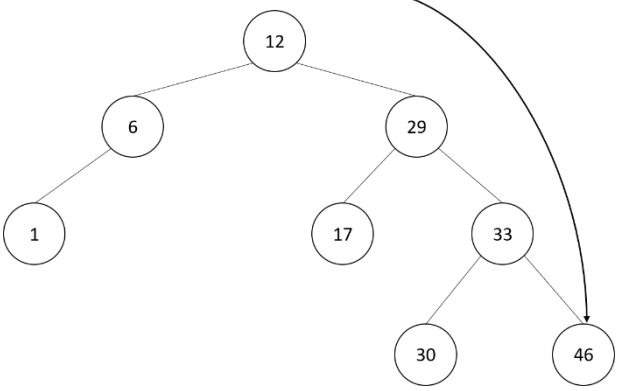
Step 0: starting state of T
Return value: none



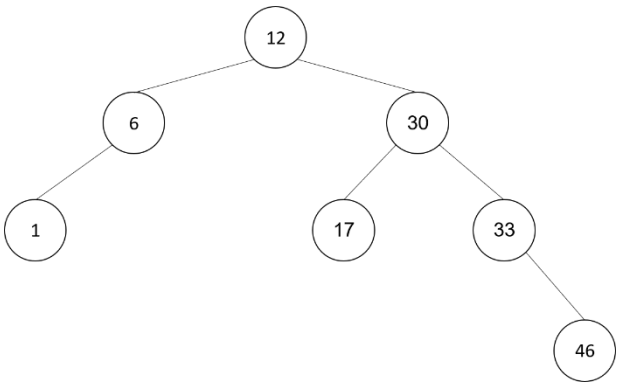

Step 1: insert(T,1)
Return value: none



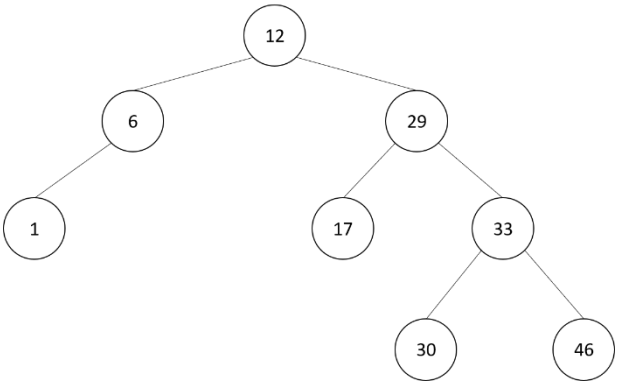

Step 2: maximum(T)
Return value: pointer



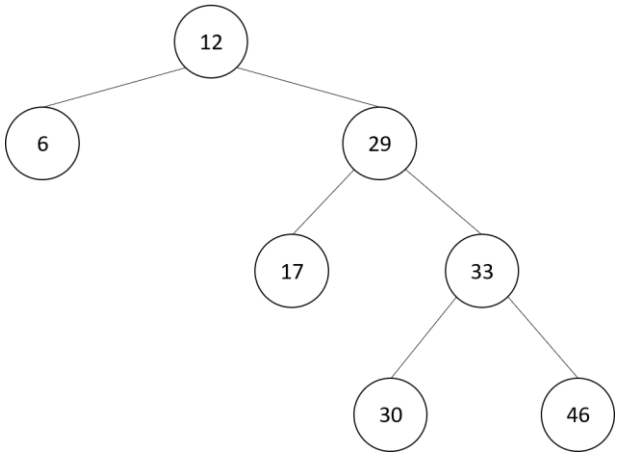

Step 3: delete(T,29)
Return value: none




Step 4: backtrack(T)
Return value: none

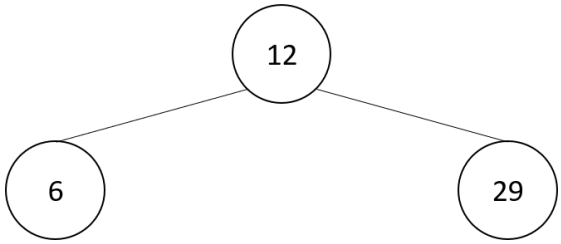



Step 5: backtrack(T)
Return value: none

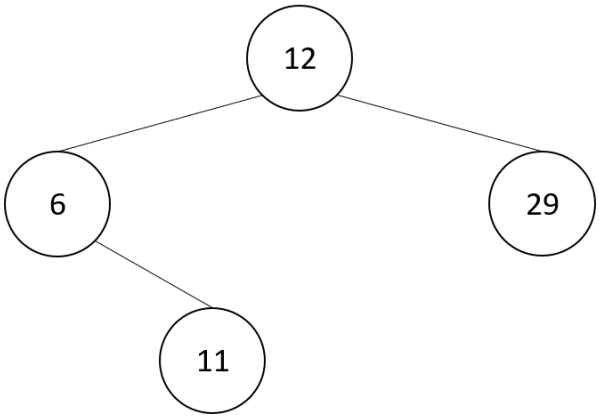


**Example for a series of steps on a BST based double backtracking dynamic set ADT.**

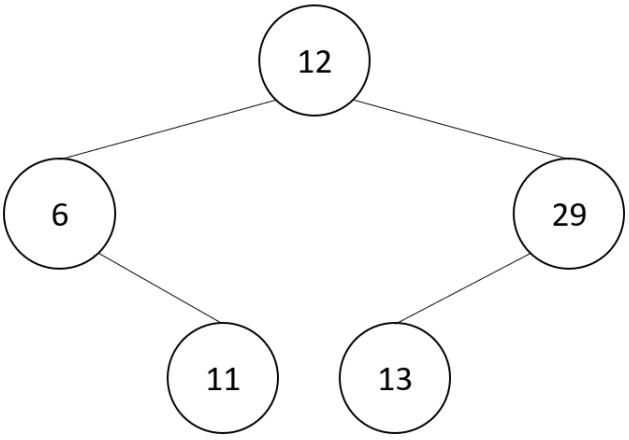
Step 0: starting state of T
Return value: none



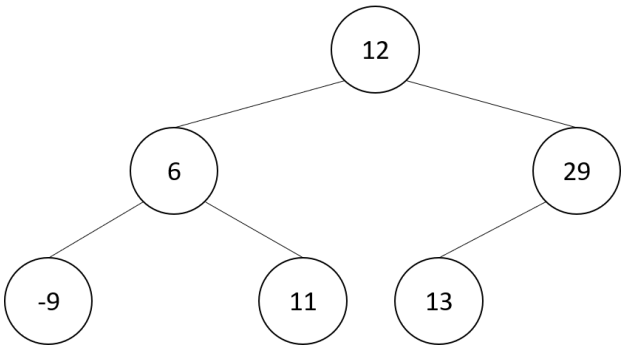

Step 1: *insert(T,11)*
Return value: none



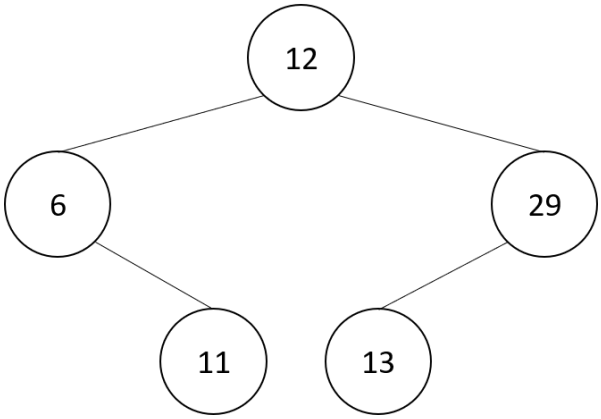

Step 2: *insert(T,13)*
Return value: none



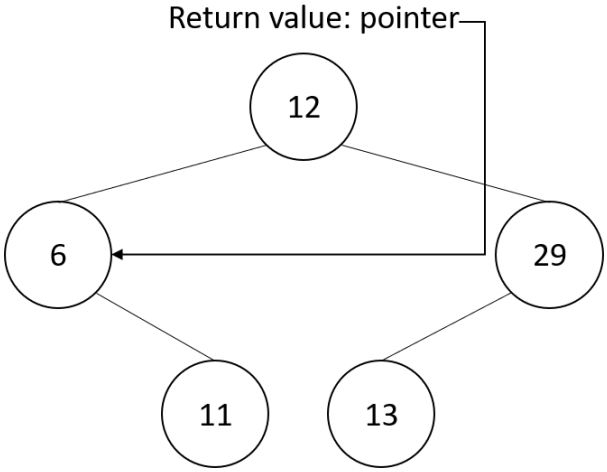

Step 3: *insert(T,-9)*
Return value: none



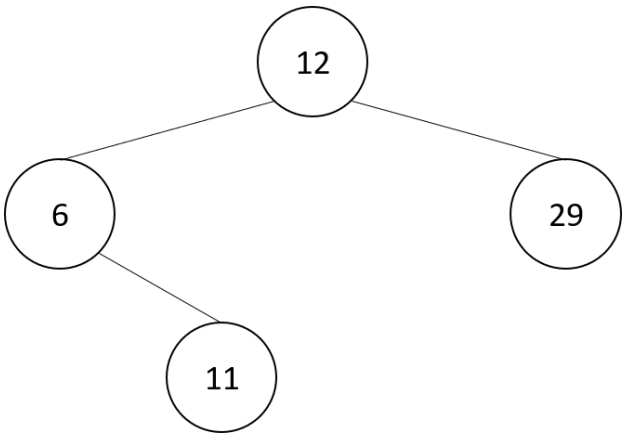

Step 4: *backtrack(T)*
Return value: none



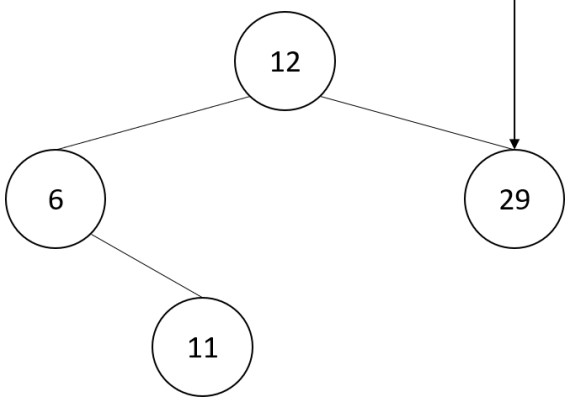

Step 5: *minimum(T)*
Return value: pointer



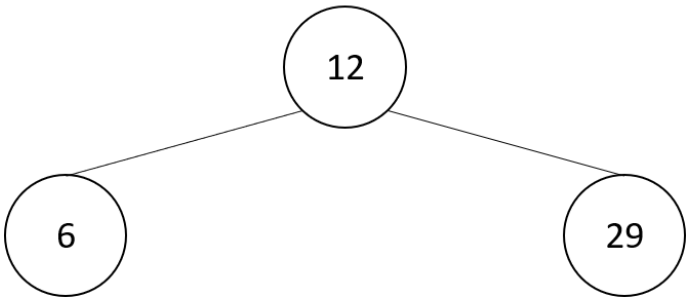
Step 6: *backtrack(T)*
Return value: none



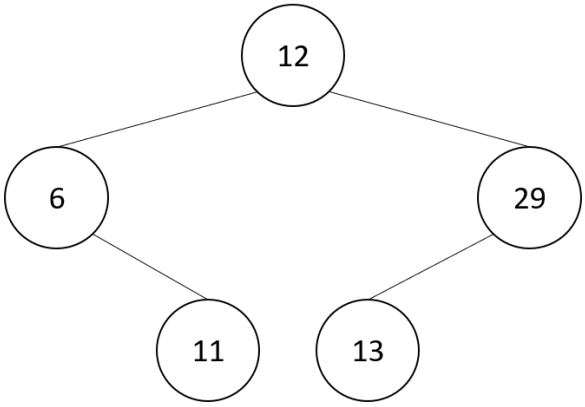

Step 9: *maximum(T)*
Return value: pointer



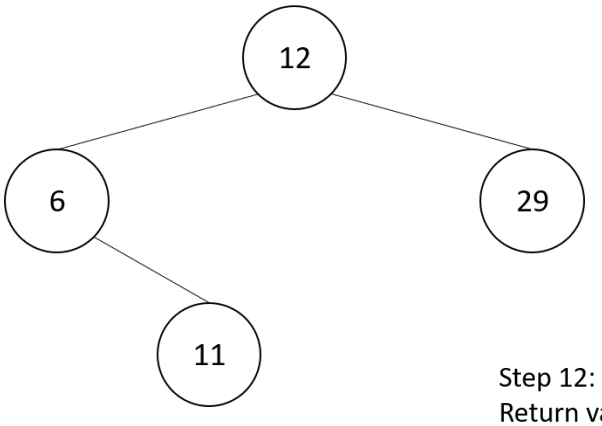

Step 7: *backtrack(T)*
Return value: none



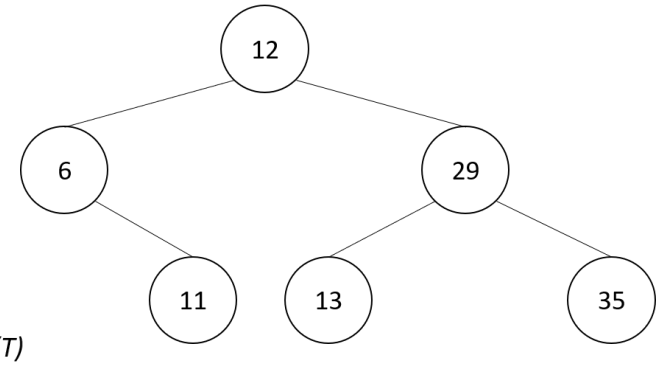

Step 10: *retrack(T)*
Return value: none



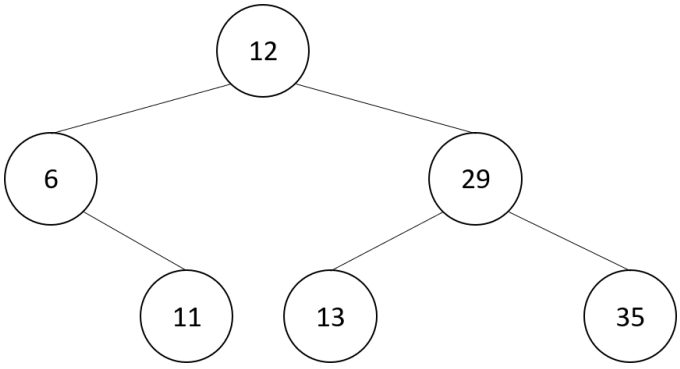

Step 8: *retrack(T)*
Return value: none




Step 11: *insert(T, 35)*
Return value: none




Step 12: *retrack(T)*
Return value: none



## Section 4: Analysis of Backtracking Data Structures and Algorithms

In this section you are required to analyze runtimes in terms of $O(\ )$ and $\Omega(\ )$ [upper and lower bounds]

1. In *i* and *ii*, the functions contain the following pseudo-code:

   $curr \leftarrow A[i]$
   If $(curr = x)$: return $i$
   else: $i = i + 1$

   We call this sequence a step.
   Analyze the runtime of the following functions. Assume an efficient implementation that uses a backtracking stack as in Section 2.

   *i.* *int backtrackingSearch1(Array A, int x).*
   This function iteratively searches for *x* in an unsorted array by performing the following actions in its $i^{th}$ iteration:
   - Perform $i$ search steps
   - Backtrack $i - 1$ steps

   *ii.* *int backtrackingSearch2(Array A, int x).*
   This function iteratively searches for *x* in an unsorted array by performing the following actions in its $i^{th}$ iteration:
   - perform 1 search step
   - Backtrack $i - 2$ steps
   - Perform $i - 2$ search steps
   - Backtrack $i - 3$ steps
   - Perform $i - 3$ search steps
   - .
   - .
   - .
   - Backtrack 1 steps
   - Perform 1 search step
   - Continue to the next iteration [ $(i + 1)^{th}$ $iteration$]

   *iii.* *BacktrackingSearch(A, x, $n_1$, $n_2$)* - for $A.length > n_1 > n_2 \geq 0$. (the function is the same one you have implemented in Section 2.a.)

2. For each of the the following implementations of the backtracking dynamic set ADT, analyze the runtime of the *backtrack(S)* method when the backtracked action is *insert(S,x)*, and when the backtracked action is *delete(S,x)*.
   i. Unsorted array based.
   ii. Sorted array based.
   iii. BST based.
   iv. AVL-tree based.

**Notice:** Assume that the implementations are efficient, and the runtimes are as you have seen in the lectures and practical sessions.

## Appendix A:

Below you can see two pseudo-code "implementations" of a function that iteratively searches an unsorted array *arr* for the value *x*. The highlighted part is a search step. Notice that the first uses a for loop, and the second uses a while loop. This pseudo-code is only intended to give some intuition of what a search step is.

$search(arr, x)$:

$n \leftarrow size(arr)$

$for\ i = 0 \rightarrow (n-1)\ do$:

$if\ (arr[i] = x)$: $break$

$end\ \backslash\backslash for$

$if\ (arr[i] = x)$: $return\ i$

$else$ : $return\ -1$

$end\ \backslash\backslash function$

$search(arr, x)$:

$n \leftarrow size(arr)$

$i \leftarrow 0$

$while\ i < n\ do$:

$curr \leftarrow arr[i]$

$if\ (curr = x)$: $break$

$i \leftarrow i + 1$

$end\ \backslash\backslash while$

$if\ (arr[curr] = x)$: $return\ curr$

$else$ : $return\ -1$

$end\ \backslash\backslash function$

## Appendix B:

In the following pseudo-code of a function that searches a sorted array *arr* for the value *x* using the binary search algorithm. The highlighted part is a search step. This pseudo-code is only intended to give some intuition of what a search step is.

$bin\_search(arr, x)$:

$min \leftarrow 0$

$max \leftarrow size(arr) - 1$

$while \max \geq \min do$:

$curr \leftarrow arr[\frac{max+min}{2}]$

$if\ (arr[curr] = x)$: $break$

$else$ :

$if\ (arr[curr] < x)$: $max \leftarrow curr - 1$

$else$ :

$min \leftarrow curr + 1$

$end\ \backslash\backslash while$

$if\ (\max \geq min)$: $return\ curr$

$else$ : $return - 1$

$end\ \backslash\backslash function$