Data Structures 2022 Assignment 2

Publish date: 12/04/2022

Due date: 10/05/2022, 23:59

Senior faculty referent: Prof. Paz Carmi

Junior faculty referents: Matan Goren, Ilan Naiman

Assignment Structure

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

Important Implementation Notes

- It is strongly recommended to read the entire assignment and FAQ section (in the moodle) before you start writing your solutions.
- The assignment may be submitted either by pairs of students or by a sole student.
- 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.
- 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 the VPL environment, and therefore you must make sure that it compiles and runs in that environment. Code that will not compile **will receive a grade of 0**. We provide some basic sanity checks for you to make sure that your code compiles.
- Don't forget to sign 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**.
- The indices throughout this assignment are zero based.

Section 0: Integrity Statement

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.

To sign and submit the integrity statement, fill your name/s in the method “signature” in the class IntegrityStatement. If you submit the assignment alone, write your full name, and if you submit the assignment in pairs, write your full names separated by “and”. See the comment in the method “signature”.

Section 1: Introduction

An **algorithm** is a series of pre-determined steps that when performed on certain objects (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. (Given some data structure *S* and object *x*, 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*.)

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**.

Some well known examples where backtracking is used:

- Enabling users to hit "Ctrl+z" on their PC and undo the last action. This is possible due to a stack operating in the background, storing the actions performed by the users in FILO (**F**irst **I**n **L**ast **O**ut) order.
- Backtracking is needed while using a GPS application, such as Waze. Let's assume that you want to ride to the Hermon mountain, and you want to have a refreshing break at the Kinneret lake. When you start your journey, Waze finds a route according to your input. During the ride, the algorithm receives updates in real time. For example, say that a road is blocked due to an accident on the route from the Kinneret the Hermon, the application will find a new route. However, it should not find a new whole route, only the second part of it. Therefore, in the worst case, the algorithm should backtrack the route until the Kinneret, and replan only from there. This lets the application to deliver a new route faster.

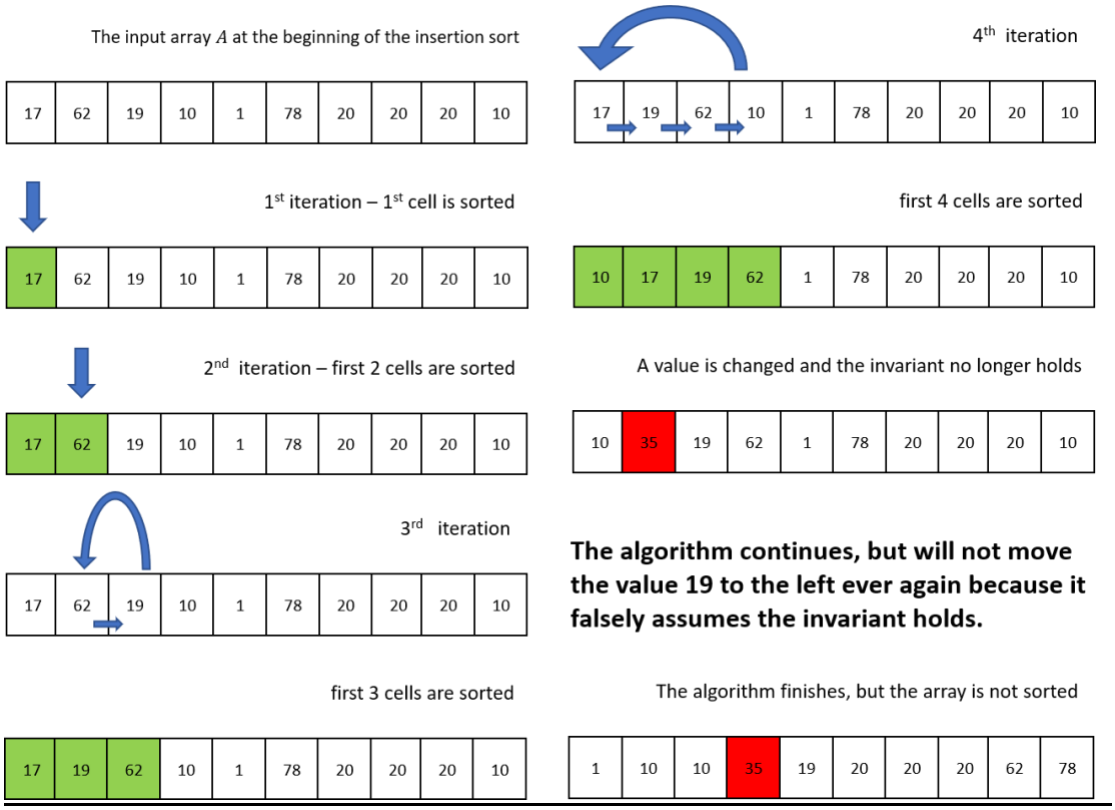
A backtracking dynamic abstract data type (ADT) *S* is an ADT with the additional method:

- *backtrack(S): cancels the last data manipulating action performed by the data structure.*

Section 2: Consistent and Backtracking Algorithms

We say that an algorithm is **consistent** if all its defined properties and invariants hold. An invariant is a property which remains unchanged after any modifications/operations are applied.

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.)



In such cases where the input given to the algorithm might change during the algorithm's execution, a **backtracking algorithm** is needed.

Section 3: Warm up Exercises

Recall the **Stack ADT**:

Stack ADT

In your code you may use the class Stack that implements the Stack ADT, but you may not submit the code for the class. The code that will be used by your submitted functions during the grading process will be provided by the course staff. The class that we will provide you will implement the following (standard stack ADT) interface:

- *stack create()*
- *bool isEmpty()*
- *void push(x)*
- *Object pop()*

In this section you will implement several basic algorithms with the support of backtracking.

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.

Notice: These exercises are mandatory and will be graded.

Implement the following methods:

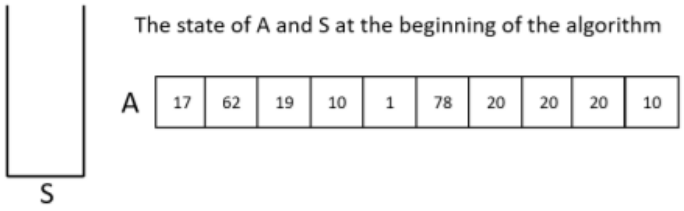
a. **public int backtrackingSearch(int[] arr, int x, int forward, int back, Stack myStack).**
The method receives an **unsorted** array of integers **arr** and searches for the index of the first occurrence of the value **x** with the added property that after every **forward** search steps (steps are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

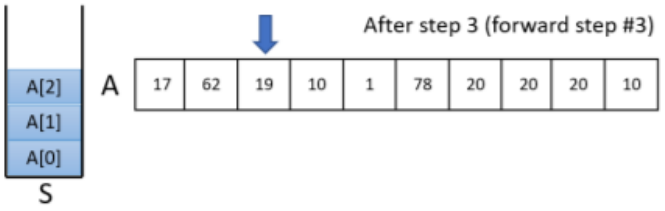
Step:
In this algorithm, a step of the search is an iteration of the main loop if the algorithm is implemented iteratively, or a recursive call if it is implemented recursively.
See Appendix A for an example.

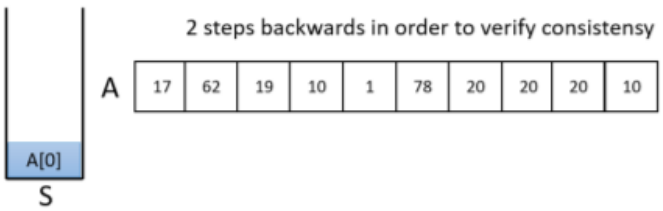
Invariant:
Every cell of **arr** that was accessed by the algorithm in previous steps does not contain the value **x**.

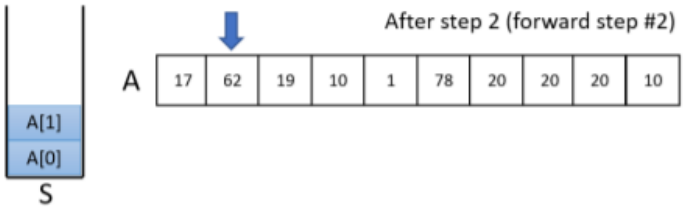
Example:
An illustration of the call backtrackingSearch*(A, 10, 3, 2)* with the array *A={17, 62, 19, 10, 1, 78, 20, 20, 20, 10}* appears next. The stack S depicted in the illustration is holding the steps performed by the algorithm.

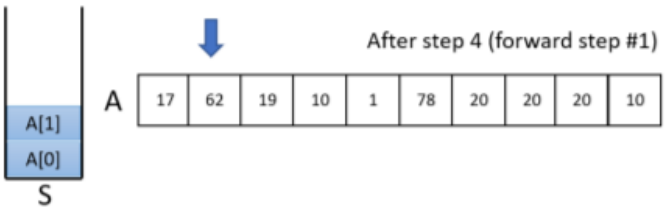
Disclaimer: the example below only illustrates the idea of the search, but the implementation ***most probably*** work in a different way.

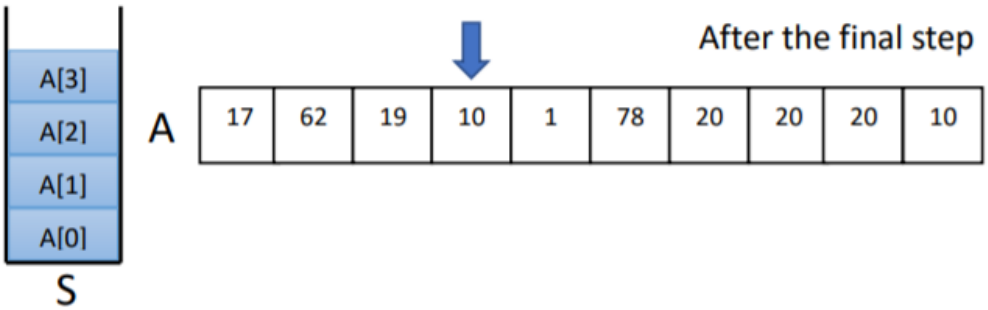












Notice: You may assume that forward > back

b. **public int consistentBinSearch(int[] arr, int x, Stack myStack).**
The method 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).
The function 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 you test your code with the following function (explanation below):

```
// Don't remove this function, but you may change its implementation
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 x, then for every $j \leq i$, $\boldsymbol{arr}[j] < x$.
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 x, then for every $j \geq i$, $\boldsymbol{arr}[j] > x$.

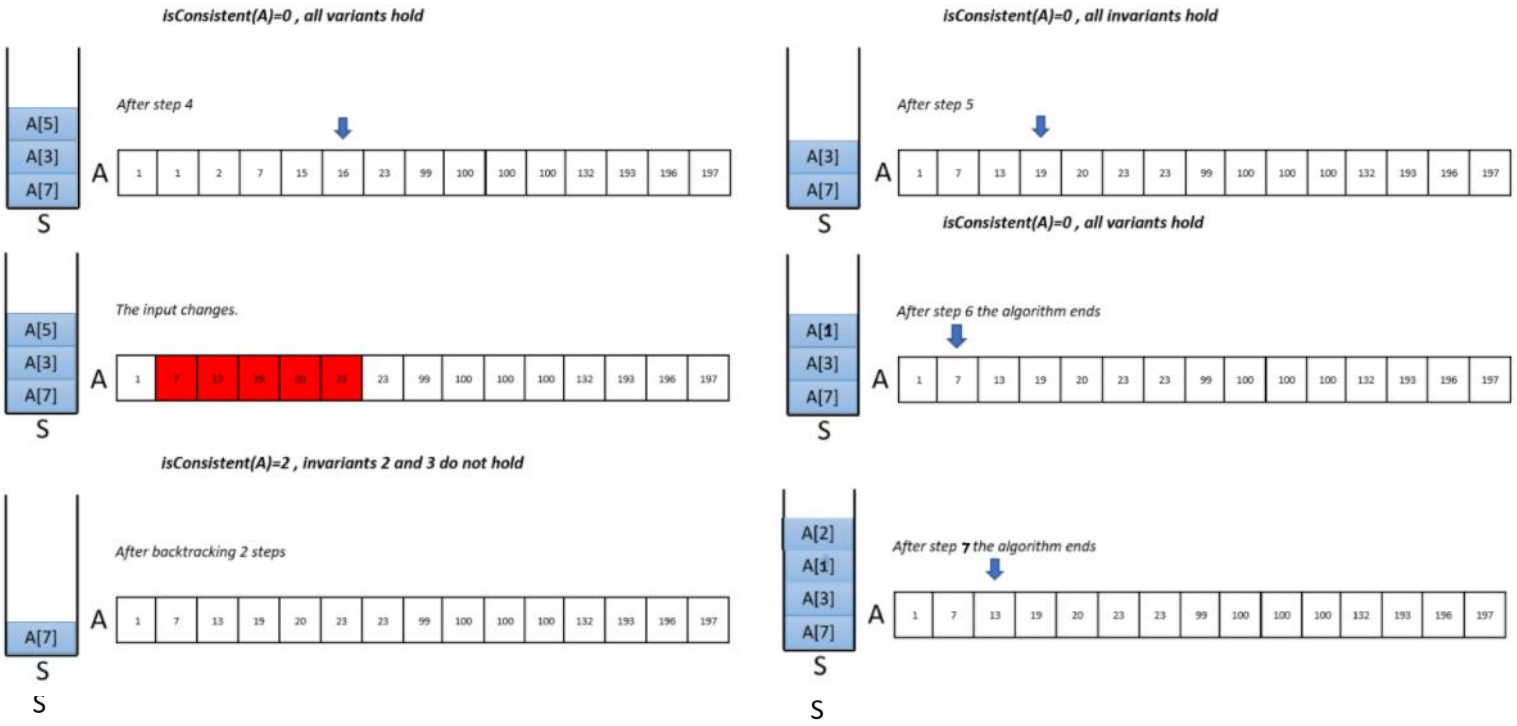
Intuitively, invariants 2 and 3 mean that all of the comparisons made by the algorithm so far are still valid.

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.

Disclaimer: the example below only illustrates the idea of the search, but the implementation ***most probably*** work in a different way.

Note that you may assume that the array remains sorted.



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.
- A search that did not find the desired index should return -1.
- The methods are described as if they take the data structure as an input. This is done for generality. Since you implement the assignment in Java, when we write something in the form of operation(S,x), you should write *public void operation(int x)* and use the standard *S.operation(x)* form.

Section 4: Backtracking Dynamic Set ADT

In this section you will implement backtracking **dynamic set ADT** using different underlying data structures.

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

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

In addition your implementation must include the following methods:

print(S) – Prints **only** 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.

In order to avoid grade reduction, make sure that your function prints exactly as described. We strongly recommend that you use the tests we provide to verify the correctness of the printing format. (see below printing examples)

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. 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.

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

In exercises (i)-(iii) you are required to implement a backtracking dynamic set ADT using different underlying data structures. In your implementation 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. There is an exception in method **print(S)** (see above).

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)* canceled 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.

Mandatory Remarks!

To simplify the requirements, assume that every element has a unique key, i.e. duplicated keys will never be inserted to the set.

Only for array-based implementations:

get(S, i) – returns the key 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.

The methods *search, minimum, maximum, predecessor and successor* **should return (if the requested value exists in the set):**

a. **a tree node if the implementation is BST-based;**
b. **an index if the implementation is array-based.**

If the requested value doesn't exist in the set:

c. **for unsuccessful** *search***, return null for BST-based implementation, and -1 for array-based implementation;**
d. **for** *minimum, maximum, predecessor and successor* **throw an appropriate exception** (Notice!! it should be of **type Exception).**

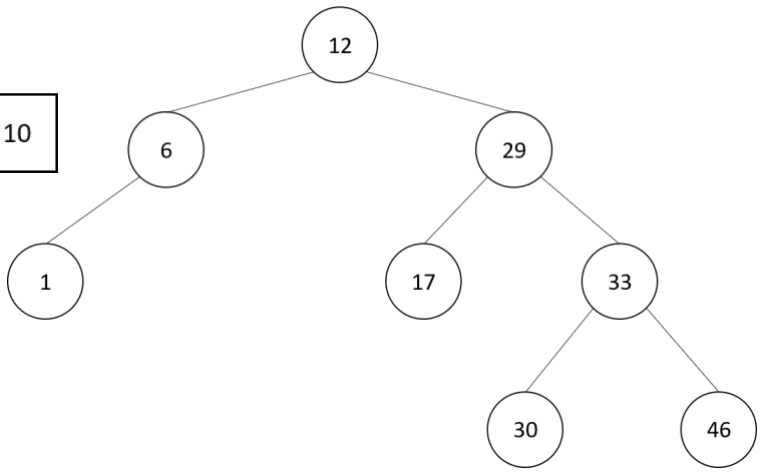
Recall that the methods are described as if they take the data structure as an input. This is done for generality. Since you implement the assignment in Java, when we write something in the form of operation(S,x), you should write *public void operation(int x)* and use the standard *S.operation(x)* form.

Printing examples:

print(S) output: "17 62 19 10 1 78 20 35 12 10"

17	62	19	10	1	78	20	35	12	10

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



Example for a series of steps on an unsorted array based dynamic set ADT:

Action 1: maximum(A) Value returned: 14

1	2	3	7	15	16	23	99	100	105	109	132	193	196	197

Action 2: delete(A,9) Value returned: none

1	2	3	7	15	16	23	99	100	109	132	193	196	197	

Action 3: delete(A,12) Value returned: none

1	2	3	7	15	16	23	99	100	109	132	193	196		

Action 4: maximum(A) Value returned: 12

1	2	3	7	15	16	23	99	100	109	132	193	196		

Action 5: backrack(A) Value returned: none

1	2	3	7	15	16	23	99	100	109	132	193	196	197	

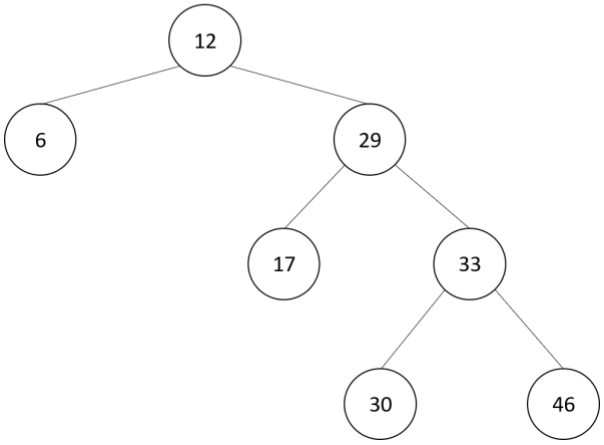
Action 6: backtrack(A) Value returned: none

1	2	3	7	15	16	23	99	100	105	109	132	193	196	197

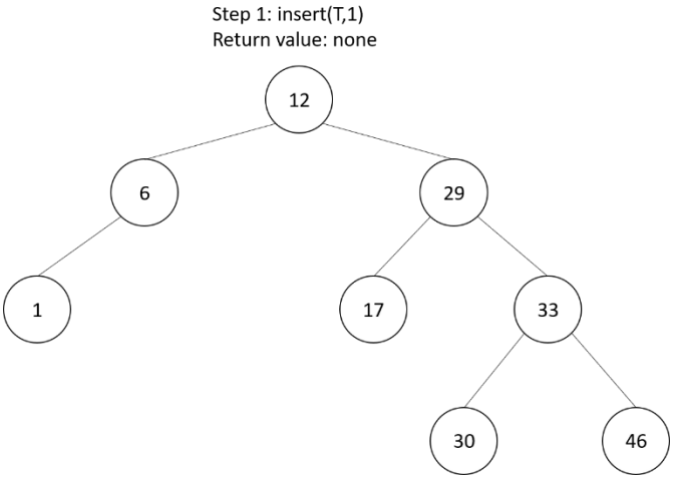
NOTICE: In the following BST based examples, when the function call is written with the key value, e.g., insert(T,1) and delete(T,20), we mean that we call the function with the reference/pointer to the element itself (the node in case of a tree) with the specified key value. This is, the Node with a key value of 1 (or 20) and not 1 (or 20) as the integer 1 (or 20)!

Example for a series of steps on a BST based dynamic set ADT:

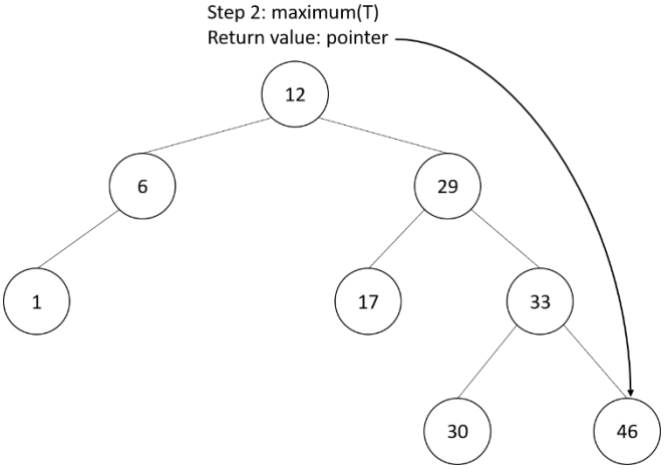
Step 0: starting state of T
Return value: none



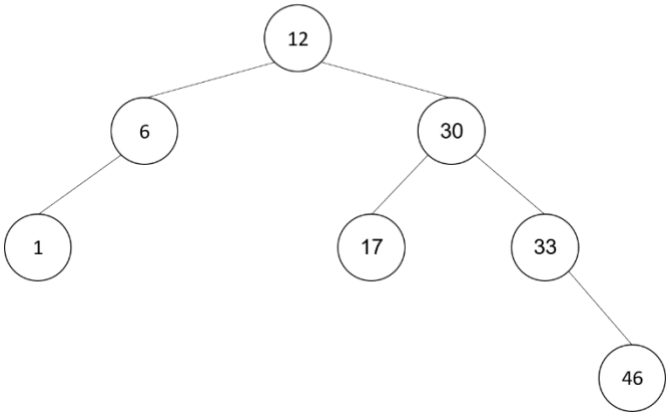
Step 1: insert(T,1)
Return value: none



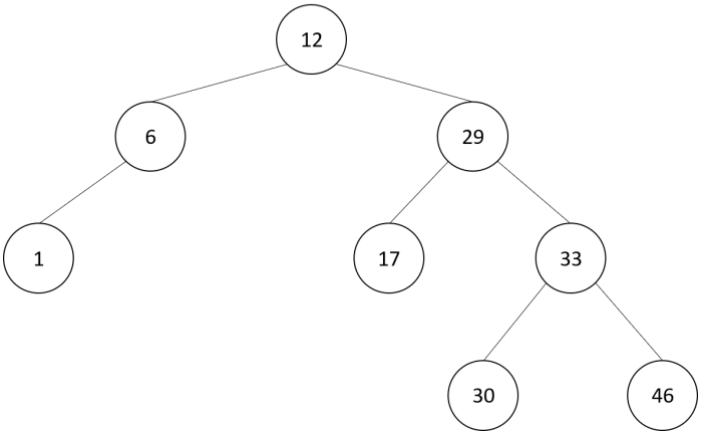
Step 2: maximum(T)
Return value: pointer



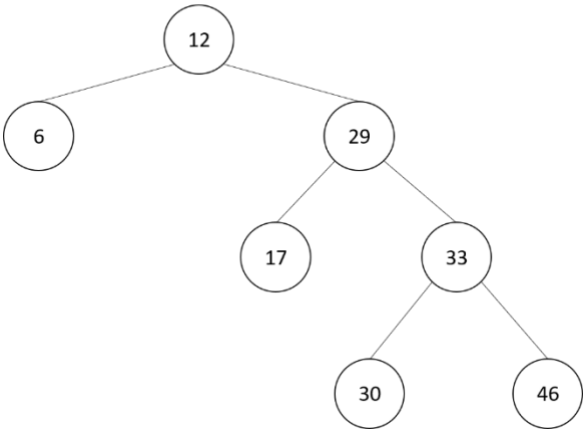
Step 3: delete(T,29)
Return value: none



Step 4: backtrack(T)
Return value: none

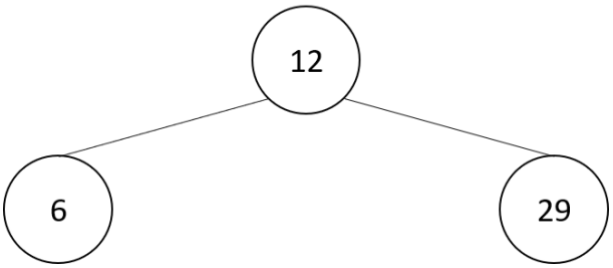


Step 5: backtrack(T)
Return value: none

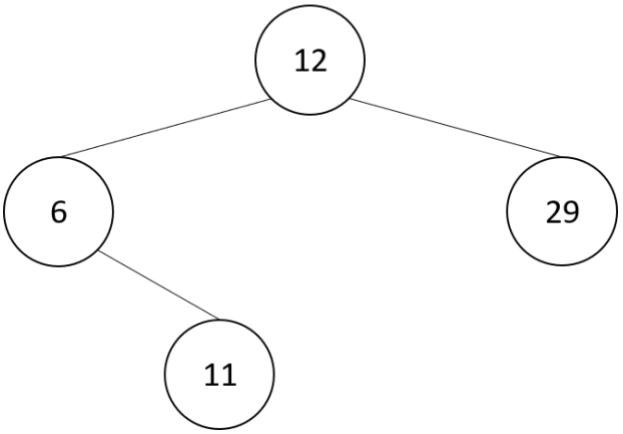


Example for a series of steps on a BST based dynamic set ADT:

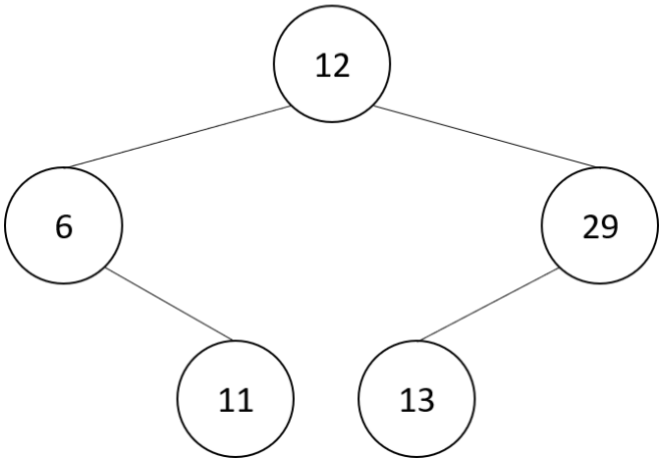
Step 0: starting state of T
Return value: none



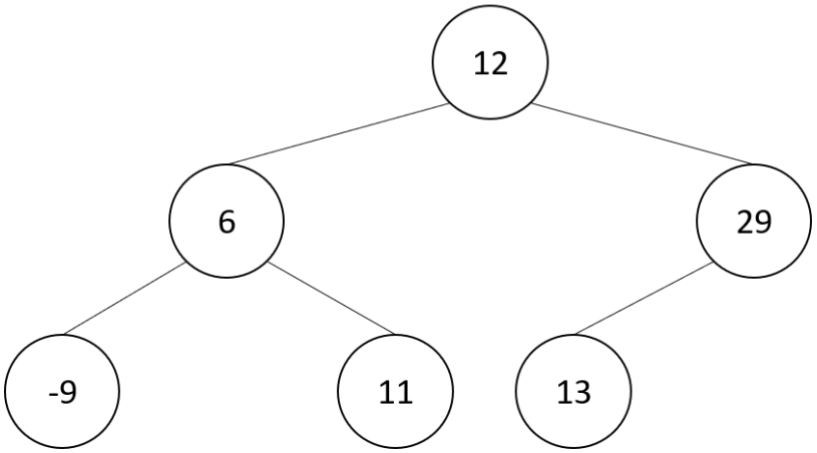
Step 1: *insert(T,11)*
Return value: none



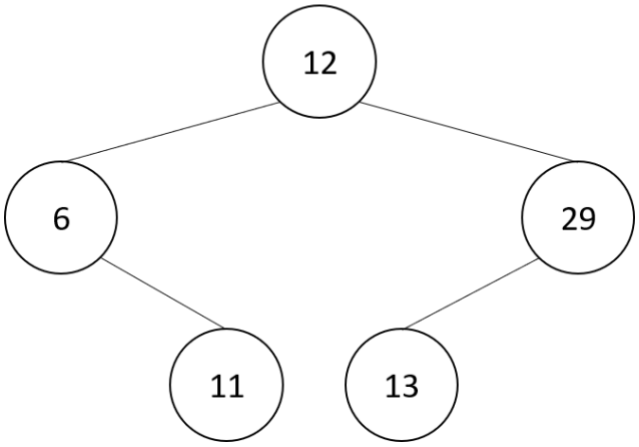
Step 2: *insert(T,13)*
Return value: none



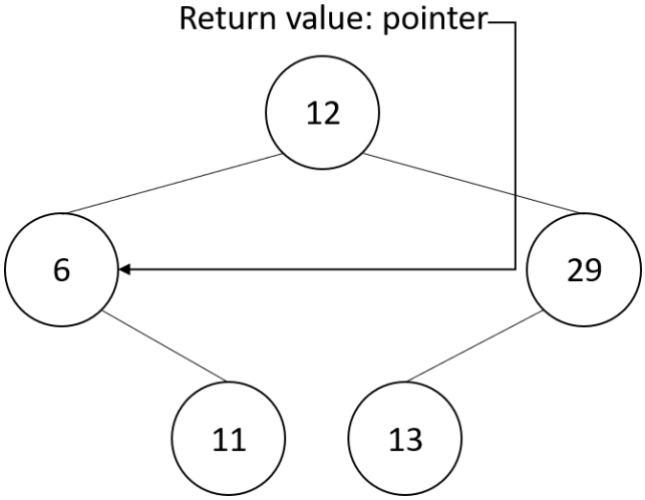
Step 3: *insert(T,-9)*
Return value: none



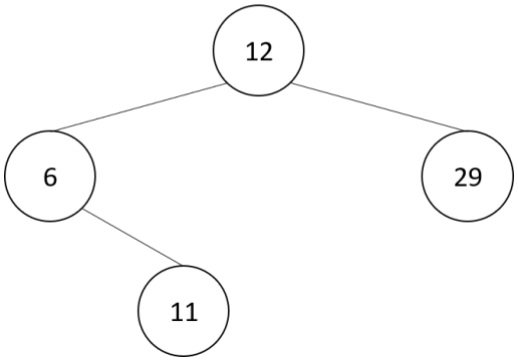
Step 4: *backtrack(T)*
Return value: none



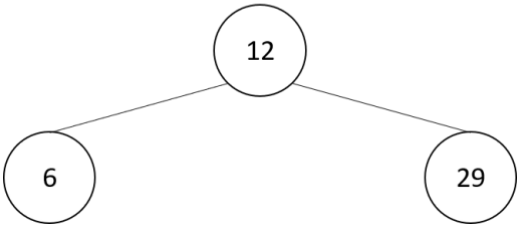
Step 5: *minimum(T)*
Return value: pointer



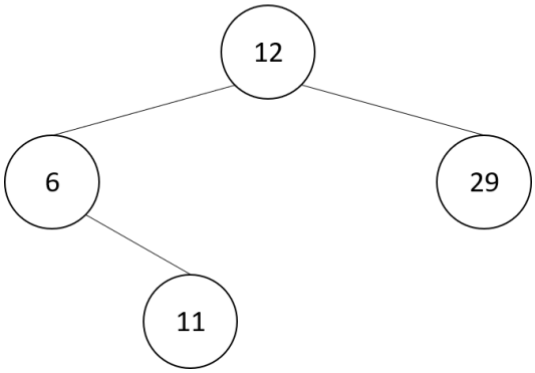
Step 6: *backtrack(T)*
Return value: none



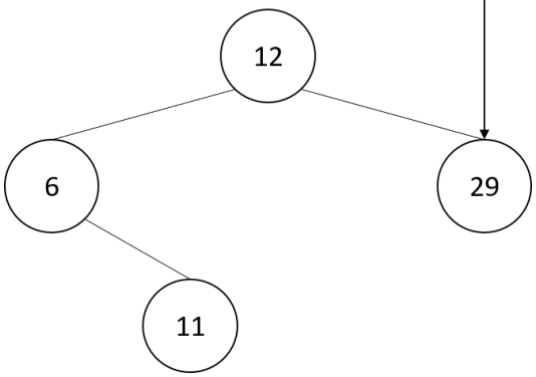
Step 7: *backtrack(T)*
Return value: none



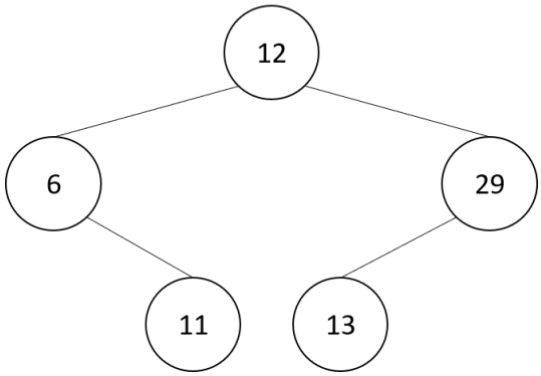
Step 8: *retrack(T)*
Return value: none



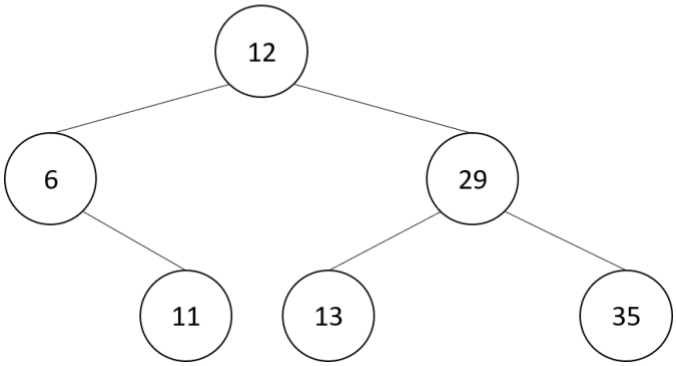
Step 9: *maximum(T)*
Return value: pointer



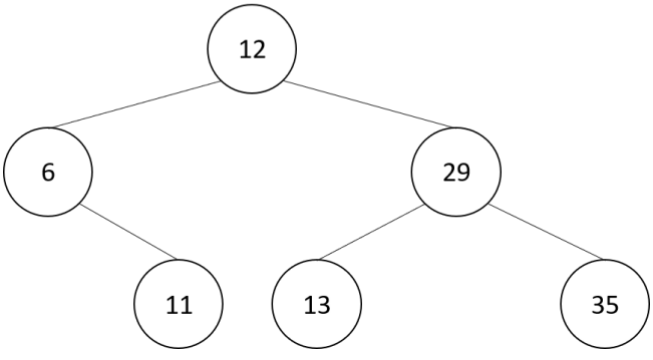
Step 10: *retrack(T)*
Return value: none



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



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



Section 4: Analysis of Backtracking Data Structures and Algorithms

For each of the implementations mentioned in sections (i)-(iv), given that the backtracking dynamic set ADT is _____-based, analyze the runtime of the ***backtrack(S)*** method when the backtracked action is *delete(S,x)*.

i. Unsorted array based.
ii. Sorted array based.
iii. BST based.
iv. AVL-tree based.

For sections (v)-(vi), analyze the minimal **space** complexity needed for printing a BST in pre-order when the implementation is done by _____.

v. Recursion.
vi. Iteration.

Hint: the recursive function which calculates factorial demands $O(n)$ space complexity, because whenever a recursive call is committed, a new variable for the numerical parameter is kept in the memory, and there are $n-1$ recursive calls.

You are supplied a class in the templates with 6 methods, each for any of the section of this question. All the answers are marked as comments. Uncomment the correct answers in each section. For example, if the answer for section ii is $\Theta\big(f(n)\big)$, then you should return the following (note which line is uncomment):

```
public static String section_ii() {
        String answer = null;
        // answer = "Theta(t(n))";
        answer = "Theta(f(n))";
        // answer = "Theta(g(n))";
        // answer = "Theta(h(n))";
        return answer;
}
```

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.

[The break command means to abort the execution of the loop (either for loop or while loop) and continue from the line after the loop.]

```
search(arr, x):
    n ← size(arr)
    for i = 0 → (n − 1) do:
        if (arr[i] = x): break
    end //for
    if (arr[i] = x): return i
    else : return − 1
end //function
```

```
search(arr, x):
    n ← size(arr)
    i = 0
    while i < n do:
        curr ← arr[i]
        if (curr = x): break
        i ← i + 1
    end //while
    if (arr[curr] = x): return curr
    else : return − 1
end //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 ≥ min do:

        $curr \leftarrow \lfloor \frac{max+min}{2} \rfloor$

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

        $else$:

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

        $else$:

            $min \leftarrow curr + 1$

    $end\ //while$

    if (max ≥ min): $return\ curr$

    $else$: $return - 1$

$end\ //function$