**MIDTERM DSA**

# Theory

### Array

An ARRAY is a collection of variables all of the same TYPE.

### Linear search – Binary search

**Binary search** requires the input **array** to **be** **sorted** in ascending or descending order. If the array is not sorted, binary search may not work correctly.

When **searching** an **unordered** **array**, one common approach is to use **linear search**, which **checks** every element of the array **one by one** until the target element is found or the end of the array is reached.

#### Linear search

**DEFINITION:** Linear search is a simple algorithmic approach for finding a specific element within an unsorted or sorted collection of elements. The algorithm starts by comparing the target value to the first element of the collection and continues searching sequentially through the collection until either the target value is found or the end of the collection is reached.

#### Binary search

**DEFINITION:** Binary search is an algorithmic approach for finding a specific element within a sorted collection of elements. The algorithm starts by comparing the target value to the middle element of the sorted collection. If the target value matches the middle element, the algorithm stops and returns the index of the middle element. Otherwise, if the target value is less than the middle element, the algorithm continues searching only in the left half of the collection, and if the target value is greater than the middle element, the algorithm continues searching only in the right half of the collection. The process repeats until the target value is found or until the algorithm determines that the target value is not in the collection.

**CODE:**

public static int binarySearch(int[] arr, int target) {

int left = 0;

int right = arr.length - 1;

while (left <= right) {

System.out.println("left value: " + left);

System.out.println("right value " + right);

int mid = left + (right - left) / 2;

System.out.println("mid value: " + mid);

System.out.println("the value in mid index:" + arr[mid]);

if (arr[mid] == target) {

return mid;

} else if (arr[mid] < target) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1; // target not found in array

}

public static void main(String[] args) {

int[] arr = {1, 3, 5, 7, 9, 10,11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21};

int target = 11;

int result = binarySearch(arr, target);

if (result == -1) {

System.out.println("Target not found in array");

} else {

System.out.println("Target found at index " + result);

}

}

**Results:**

left value: 0

right value 16

mid value: 8

the value in mid index: 13

left value: 0

right value 7

mid value: 3

the value in mid index: 7

left value: 4

right value 7

mid value: 5

the value in mid index: 10

left value: 6

right value 7

mid value: 6

the value in mid index: 11

Target found at index 6

### Big O notation

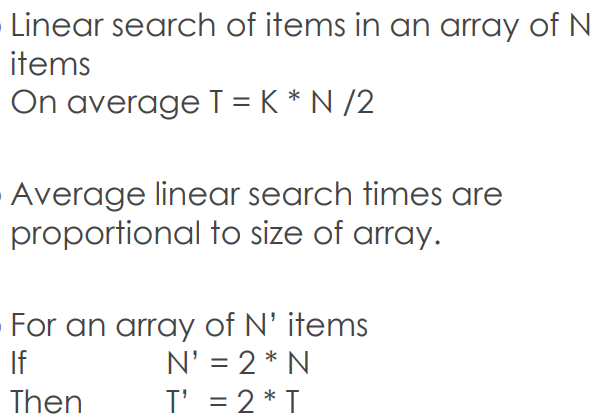
* To measure the EFFICIENCY of algorithms.
* Step: O(N^2), O(N), O(logN), O(1)

#### Constant – O(1)

The time needed by the algorithm is not depend in size of items.

#### Proportional to N – O(N)

* Linear search has a time complexity of O(n), where n is the size of the collection. This means that in the worst case, the algorithm may need to examine every element of the collection before finding the target value.



#### Proportional to log(N) – O(log N)

* Binary search is a very efficient algorithm for searching sorted collections, with a time complexity of O(log n), where n is the size of the collection.

Text

Description automatically generated

For (1->m)

For(1->n) -> m\*n

For (1->m)

For(1->n)

If (condition) -> 1

-> m + n + 1 -> m

### Simple Sorting

* The algorithm with the smallest number of comparisons is Insertion Sort.
* The algorithm with the smallest number of swaps is Selection Sort.
* The algorithm with the smallest number of copies is Insertion Sort.
* The speed of the three types of sort from fastest to slowest is: Insertion Sort, Selection Sort, Bubble Sort

#### Bubble Sort

* Compare the first item in the first two positions (e.g. the left most)
* If the first one is larger, swap with the second
* Move one position right.
* Stop comparing at (n-2).
* Thus number of comparisons is computed as:

(n-1)+(n-2)+… + 1 = n(n-1)/2.

* A swap will occur half the time: For /2 comparisons, we have /4 swaps.
* Both swaps and compares are proportional to (Ignore the 2 and 4)
* Complexity of Bubble Sort is O()
* Space complexity: O(1)

**CODE EXAMPLE:**

##### BubbleSort algorithm.

public static void BubbleSort(int[] arr){

int nElems = arr.length;

for(int out = nElems - 1; out>1; out--){

for(int in = 0; in < out; in++) {

if(arr[in] > arr[in+1]){

swap(arr, in, in+1);

}

}

}

}

##### To sort from Right to Left

public static void bubbleSortRightToLeft(int[] arr){

int nElems = arr.length;

for(int out = 0; out < nElems - 1; out++){

for(int in = nElems - 1; in > out; in--) {

if(arr[in] < arr[in-1]){

swap(arr, in, in-1);

}

}

}

}

##### To sort array descending (Đổi dấu > thành < ở if)

##### To use 2 forward loops

public static void bubbleSortTwoLoops(int[] arr){

int nElems = arr.length;

boolean swapped;

do {

swapped = false;

for(int in = 0; in < nElems - 1; in++) {

if(arr[in] > arr[in+1]){

swap(arr, in, in+1);

swapped = true;

}

}

} while (swapped);

}

##### Testing

public static void main(String[] args) {

int[] arr = {4,1,6,9,2,5,8,0,1,4};

**displayArray(arr);**

BubbleSort(arr);

**displayArray(arr);**

bubbleSortRightToLeft(arr);

**displayArray(arr);**

bubbleSortDescending(arr);

**displayArray(arr);**

bubbleSortTwoLoops(arr);

**displayArray(arr);**

}

**RESULT:**

4 1 6 9 2 5 8 0 1 4

0 1 1 2 4 4 5 6 8 9

0 1 1 2 4 4 5 6 8 9

8 9 6 5 4 4 2 1 1 0

0 1 1 2 4 4 5 6 8 9

#### Selection Sort (Fewer swaps – O(n), same comparisons.)

* Start from the first element (e.g. the left end).
* Scan all elements to selecting the smallest (largest) item.
* Swap with the first element.
* Next pass, move one position right.
* Repeat until all are sorted.
* Time complexity: O()
* Space complexity: O(1)

So, in one pass, you have made n comparisons but possibly ONLY ONE Swap!

* With each succeeding pass:
  + One more item is sorted and in place
  + One fewer item needs to be considered.

**CODE EXAMPLE:**

##### SelectionSort algorithm

public static void SelectionSort(int[] arr){

int nElems = arr.length;

for(int out = 0; out<nElems - 1; out++){

int min = out;

for(int in = out+1; in<nElems; in++){

if(arr[in] < arr[min]){

min = in;

}

}

swap(arr, out, min);

}

}

##### To sort from Right to Left

public static void selectionSortRightToLeft(int[] arr) {

int nElems = arr.length;

for (int out = nElems - 1; out > 0; out--) {

int max = out;

for (int in = out - 1; in >= 0; in--) {

if (arr[in] > arr[max]) {

max = in;

}

}

swap(arr, out, max);

}

}

##### To sort array descending (Đổi min thành max, dấu < ở lệnh if thành >)

#### Insertion Sort

* a[out] is the marked item, and it is moved into temp. a[out] is the leftmost unsorted item.
* Algorithm is an O() sort (worst case), O(n) (best case) (time complexity).
* Space complexity: O(1)
* If data is in very unsorted order (nearly backward) → No faster than the bubble sort

**CODE EXAMPLE:**

public static void InsertionSort(int[] arr) {

int nElems = arr.length;

for (int i = 1; i < nElems; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

### Stack

* Stack is a data structure that follows the Last-In-First-Out (LIFO) principle, meaning the last element added to the stack is the first one to be removed. In a stack, elements are added and removed from the **top** (accessible item) of the stack.
* Operations: Push, Pop, Peek. (All complexity of them are O(1))
* Properties: Stack Overflow (Full) – Stack size (is full?), Stack Underflow (Empty) – Number of element (is empty?)

**CODE EXAMPLE:**

Stack<Integer> myStack = new Stack<>();

// pushing elements to the stack

myStack.push(5);

myStack.push(10);

myStack.push(15);

System.out.println(myStack);

// removing top element from stack

int popped = myStack.pop();

System.out.println("Popped element: " + popped);

System.out.println(myStack);

// accessing top element without removing

int peeked = myStack.peek();

System.out.println("Peeked element: " + peeked);

System.out.println(myStack);

**RESULT:**

[5, 10, 15]

Popped element: 15

[5, 10]

Peeked element: 10

[5, 10]

### Queue

* Queue is a data structure that follows the FIFO (First In First Out) principle. It is an abstract data type that can be implemented using an array or a linked list.
* A queue can be used to store a collection of elements and allow them to be accessed in the order they were added. Elements are added to the back of the queue and removed from the front of the queue.
* Operations: Insert/ Enqueue, Remove/ Dequeue. (Time complexity - O(1))
* Properties: Full – Queue size (is full?), Empty – Number of element (is empty?)
* Accessible item: Head – 1st element added, Tail – last element added.

**CODE EXAMPLE:**

// Create a queue of integers

Queue<Integer> queue = new LinkedList<>();

// Enqueue elements to the queue

queue.add(10);

queue.add(20);

queue.add(30);

queue.add(40);

queue.add(50);

// Dequeue and print the first element in the queue

int removedElement = queue.remove();

System.out.println("Removed Element: " + removedElement);

// Print the remaining elements in the queue

System.out.println("Elements in Queue: " + queue);

// Enqueue another element to the queue

queue.add(60);

// Dequeue and print the first element in the queue

removedElement = queue.remove();

System.out.println("Removed Element: " + removedElement);

// Print the remaining elements in the queue

System.out.println("Elements in Queue: " + queue);

**RESULT:**

Removed Element: 10

Elements in Queue: [20, 30, 40, 50]

Removed Element: 20

Elements in Queue: [30, 40, 50, 60]

### Priority Queues

* Efficiency of priority queue: Insertion (if use ARRAY: O(N)), Deletion O(1).
* One difference between a priority queue and an ordered array is that the highest priority item can be extracted easily from the priority queue but not from the array.

**CODE EXAMPLE:**

// Create a priority queue

PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

// Add elements to the priority queue

pq.add(5);

pq.add(2);

pq.add(8);

pq.add(1);

pq.add(6);

// Print the top element of the priority queue

System.out.println("Top element: " + pq.peek());

// Remove the top element of the priority queue

pq.poll();

// Print all elements of the priority queue

while (!pq.isEmpty()) {

System.out.print(pq.poll() + " ");

}

**RESULT:**

Top element: 1

2 5 6 8

### Parsing Arithmetic Expressions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Character read from Infix | Infix parsed so far | Postfix written so far | Stack contents | Rule |
| A | A | A |  | Write operand to output |
| + | A+ | A | + | If stack empty, push opThis |
| B | A+B | AB | + | Write operand to output |
| - | A+B- | AB |  | Stack not empty, so pop item |
|  | A+B- | AB+ |  | opThis is -, opTop is +, opTop>=opThis, so output opTop |
|  | A+B- | AB+ | - | Then push onThis |
| C | A+B-C | AB+C | - | Write operand to output |
| End | A+B-C | AB+C- |  | Pop leftover item, output it |

***Translation Rules Applied to A+B-C***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Character read from Infix | Infix parsed so far | Postfix written so far | Stack contents | Rule |
| A | A | A |  | Write operand to postfix |
| + | A+ | A | + | If stack empty, push onThis |
| B | A+B | AB | + | Write operand to output |
| \* | A+B\* | AB | + | Stack not empty, so pop opTop |
|  | A+B\* | AB | + | opThis is \*, opTop is +, opTop<opThis, so push opTop |
|  | A+B\* | AB | +\* | Then push onThis |
| C | A+B\*C | ABC | +\* | Write operand to output |
| End | A+B\*C | ABC\* | + | Pop leftover item, output it |
|  | A+B\*C | ABC\*+ |  | Pop leftover item, output it |

***Translation Rules Applied to A+B\*C***

### Simple linked list

### Double-ended list

### Sorted list

### Doubly linked list

### List with iterators

### Triangular Numbers

### Factorials

### A Recursive Binary Search

# Code

### Simple code

#### Code function to display every element in an Array.

public static void displayArray(int[] arr) {

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[i] + " ");

}

System.out.println();

}

#### Code function to swap the position of element in an Array

public static void swap(int[] arr, int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

### Lab

# Midterm exam 2022

### Sort (20 marks)

Given array B

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Data | 72 | 4 | 12 | 53 | 15 | 2 | 8 | 16 | 40 | 21 | 7 | 87 | 35 |

Use the merge sort algorithm to sort array B in ascending order by filling the table below:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Action | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Consider values in the range from 0 to 6 | 72 | 4 | 12 | 53 | 15 | 2 | 8 |  |  |  |  |  |  |
| … | 72 | 4 | 12 | 53 |  |  |  |  |  |  |  |  |  |
| Sort value in the rage | 72 | 4 |  |  |  |  |  |  |  |  |  |  |  |
| … | 72 |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 4 |  |  |  |  |  |  |  |  |  |  |  |
|  | 4 | 72 |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 12 | 53 |  |  |  |  |  |  |  |  |  |
|  |  |  | 12 |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 53 |  |  |  |  |  |  |  |  |  |
|  |  |  | 12 | 53 |  |  |  |  |  |  |  |  |  |
|  | 4 | 12 | 53 | 72 |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | 15 | 2 | 8 |  |  |  |  |  |  |
|  |  |  |  |  | 15 | 2 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |

### Queue and Stack (20 marks)

#### (10 marks) What values are returned after each dequeue() during the following sequence of queue operations, if executed on an initially empty queue?

#### enqueue(2), enqueue(0), dequeue(), enqueue(2), enqueue(2), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue()

The sequence of queue operations is:

* enqueue(2) - The queue becomes [2] - enqueue(0) - The queue becomes [2, 0]
* **dequeue() - The first element 2 is removed and returned. The queue becomes [0]**
* enqueue(2) - The queue becomes [0, 2] - enqueue(2) - The queue becomes [0, 2, 2]
* **dequeue() - The first element 0 is removed and returned. The queue becomes [2, 2]**
* **dequeue() - The next element 2 is removed and returned. The queue becomes [2]**
* enqueue(9) - The queue becomes [2, 9]- enqueue(1) - The queue becomes [2, 9, 1]
* **dequeue() - The first element 2 is removed and returned. The queue becomes [9, 1]**
* enqueue(7) - The queue becomes [9, 1, 7] - enqueue(6) - The queue becomes [9, 1, 7, 6]
* **dequeue() - The first element 9 is removed and returned. The queue becomes [1, 7, 6]**
* **dequeue() - The next element 1 is removed and returned. The queue becomes [7, 6]**
* enqueue(4) - The queue becomes [7, 6, 4]
* **dequeue() - The first element 7 is removed and returned. The queue becomes [6, 4]**
* **dequeue() - The next element 6 is removed and returned. The queue becomes [4]**

Therefore, the values returned after each dequeue() operation are: 2, 0, 2, 2, 9, 2, 1, 9, 7, 6. The final state of the queue is [4].

#### What values are returned after each pop() during the following series of stack operations, if executed upon an initially empty stack?

#### push(5), push(3), pop(), push(2), push(0), pop(), pop(), push(2), push(2), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop().

The sequence of stack operations is:

* push(5) - The stack becomes [5] - push(3) - The stack becomes [5, 3]
* **pop() - The last element 3 is removed and returned. The stack becomes [5]**
* push(2) - The stack becomes [5, 2] - push(0) - The stack becomes [5, 2, 0]
* **pop() - The last element 0 is removed and returned. The stack becomes [5, 2]**
* **pop() - The next element 2 is removed and returned. The stack becomes [5]**
* push(2) - The stack becomes [5, 2] - push(2) - The stack becomes [5, 2, 2]
* **pop() - The last element 2 is removed and returned. The stack becomes [5, 2]**
* push(7) - The stack becomes [5, 2, 7] - push(6) - The stack becomes [5, 2, 7, 6]
* **pop() - The last element 6 is removed and returned. The stack becomes [5, 2, 7]**
* **pop() - The next element 7 is removed and returned. The stack becomes [5, 2]**
* push(4) - The stack becomes [5, 2, 4]
* **pop() - The last element 4 is removed and returned. The stack becomes [5, 2]**
* **pop() - The next element 2 is removed and returned. The stack becomes [5]**

Therefore, the values returned after each pop() operation are: 3, 0, 2, 2, 6, 7, 4, 2. The final state of the stack is [5].

### Linked list (20 marks)

Given 2 classes Node and DoublyLinkList

|  |  |
| --- | --- |
| public class DoublyLinkList{  private Node first;  private Node last;  public void selectionSort() {//**TODO**}  } | public class Node {  private int key;  private Node next;  private Node previous;  } |

Implement the selection sort algorithm for a doubly-linked list.

To implement selection sort algorithm for a doubly-linked list, we can follow the steps below:

* Define a method **selectionSort**() in the **DoublyLinkList** class that will perform the selection sort algorithm on the doubly-linked list.
* Initialize two pointers **i** and **j** to point to the first node of the doubly-linked list.
* Traverse the doubly-linked list using the outer loop with pointer **i**, starting from the first node.
* Within the outer loop, initialize a variable **min** to the current node **i.**
* Traverse the doubly-linked list using the inner loop with pointer **j**, starting from the next node of **i**.
* Within the inner loop, compare the **key** value of each node with the **key** value of **min**.
* If a smaller **key** value is found, set **min** to the current node **j**.
* After the inner loop, swap the **key** value of **i** with the **key** value of **min**.
* Move the pointer **i** to the next node of the doubly-linked list.
* Repeat steps 4 to 9 until the end of the doubly-linked list is reached.

Here is the implementation of **selectionSort**() method:

public void selectionSort() {

if (first == null || first.next == null) {

return; // list is empty or has only one node

}

Node i = first;

while (i != last) {

Node min = i;

Node j = i.next;

while (j != null) {

if (j.key < min.key) {

min = j;

}

j = j.next;

}

if (min != i) {

int temp = i.key;

i.key = min.key;

min.key = temp;

}

i = i.next;

}

}

In this implementation, we have used two pointers, **i** and **j**, to traverse the doubly-linked list. The **min** variable is used to keep track of the node with the smallest **key** value within the inner loop. We have also used the key value to perform the comparison and swap operations.

The time complexity of the **selectionSort**() method is O(n^2) in the worst case, since we are using two nested loops to traverse the doubly-linked list. The space complexity is O(1), since we are only using a constant amount of extra space to perform the sorting.

### Queue (30 marks)

Propose the BEST ways to implement a PRIORITY QUEUE with the given data structures. Fill the table below by.

#### Mapping the operation of a queue to the operation on the data structures in use.

#### Showing the complexity of each operation of the queue in the corresponding data structure.

|  |  |  |  |
| --- | --- | --- | --- |
| Data structure | Array  (10 mark) | Double Ended Linked list  (10 marks) | Doubly Linked list  (10 marks) |
| Enqueue |  |  |  |
| Dequeue |  |  |  |
| IsFull |  |  |  |
| IsEmpty | Store and check the number of items in the array. O(1) | Check whether the first element is null. O(1) | Check whether the first element is null. O(1) |

### Stack (10 marks)

#### (5 marks) Implement a stack using two queues and analyze the complexity of PUSH and POP operations.

To implement a stack using two queues, we can use the following algorithm:

* We start with two empty queues, let's call them **queue1** and **queue2**.
* To push an element onto the stack, we enqueue the element into **queue1**.
* To pop an element from the stack, we move all the elements except the last one from **queue1** to **queue2**, and then dequeue the last element from **queue1**. We then swap the names of **queue1** and **queue2**, so that **queue2** becomes empty and **queue1** becomes the new stack.

Here is the Java code for implementing a stack using two queues:

import java.util.Queue;

import java.util.LinkedList;

public class StackUsingQueues<T> {

private Queue<T> queue1 = new LinkedList<>();

private Queue<T> queue2 = new LinkedList<>();

public void push(T element) {

queue1.offer(element);

}

public T pop() {

if (queue1.isEmpty()) {

throw new IllegalStateException("Stack is empty");

}

while (queue1.size() > 1) {

queue2.offer(queue1.poll());

}

T element = queue1.poll();

Queue<T> temp = queue1;

queue1 = queue2;

queue2 = temp;

return element;

}

}

Now, let's analyze the time complexity of the **push()** and **pop()** operations in this implementation:

* **push()**: Enqueuing an element into a queue takes constant time, so the time complexity of the **push()** operation is O(1).
* **pop()**: Dequeuing an element from a non-empty queue takes constant time, so the time complexity of moving all the elements except the last one from **queue1** to **queue2** is O(n), where n is the number of elements in **queue1**. The time complexity of dequeuing the last element from **queue1** and swapping the names of **queue1** and **queue2** is also O(n), but it is dominated by the first O(n) operation. Therefore, the time complexity of the **pop()** operation is O(n).

In the worst case, when the stack is full and we need to move all the elements from **queue1** to **queue2** to perform a pop operation, the time complexity of the **pop()** operation is O(n). However, on average, the time complexity of the **pop()** operation is much lower, since we only need to move a small number of elements from **queue1** to **queue2**. Overall, this implementation provides a reasonable trade-off between time and space complexity, since it uses two queues but only one queue is non-empty at any given time.

#### (5 marks) Implement two stacks using one array of size N in such a way that:

* PUSH and POP operations run in O(1) in time.
* Neither stack overflows unless the total number of elements in both stacks together is N.

To implement two stacks using one array of size N in such a way that PUSH and POP operations run in O(1) time and neither stack overflows unless the total number of elements in both stacks together is N, we can use the following approach:

* Divide the array into two equal halves, one for each stack.
* Define two pointers, one for each stack, to keep track of the top of the stack.
* When pushing an element onto a stack, increment the corresponding pointer and set the element at the index of the pointer.
* When popping an element from a stack, return the element at the index of the pointer and decrement the corresponding pointer.
* To check if a stack is empty, check if the corresponding pointer is pointing to the first element of the stack.
* To check if a stack is full, check if both pointers are pointing to the same index in the array.

Here is the Java code for implementing two stacks using one array:

public class TwoStacks {

private int[] array;

private int pointer1; // pointer for stack 1

private int pointer2; // pointer for stack 2

public **TwoStacks**(int size) {

if (size < 2) {

throw new IllegalArgumentException("Size should be at least 2");

}

array = new int[size];

pointer1 = -1;

pointer2 = size;

}

public void **push1**(int element) {

if (pointer1 + 1 == pointer2) {

throw new IllegalStateException("Stack 1 is full");

}

pointer1++;

array[pointer1] = element;

}

public int **pop1**() {

if (pointer1 == -1) {

throw new IllegalStateException("Stack 1 is empty");

}

int element = array[pointer1];

pointer1--;

return element;

}

public boolean **isEmpty1**() {

return pointer1 == -1;

}

public boolean **isFull1**() {

return pointer1 + 1 == pointer2;

}

public void **push2**(int element) {

if (pointer2 - 1 == pointer1) {

throw new IllegalStateException("Stack 2 is full");

}

pointer2--;

array[pointer2] = element;

}

public int **pop2**() {

if (pointer2 == array.length) {

throw new IllegalStateException("Stack 2 is empty");

}

int element = array[pointer2];

pointer2++;

return element;

}

public boolean **isEmpty2**() {

return pointer2 == array.length;

}

public boolean **isFull2**() {

return pointer1 + 1 == pointer2;

}

}

In this implementation, we have used two pointers, pointer1 and pointer2, to keep track of the top of each stack. The push1() and pop1() methods are used for the first stack, while the push2() and pop2() methods are used for the second stack. The isEmpty1(), isFull1(), isEmpty2(), and isFull2() methods are used to check if a stack is empty or full.

The time complexity of the push1(), pop1(), push2(), and pop2() methods is O(1), since they only involve incrementing or decrementing the corresponding pointer and accessing or setting an element in the array. The space complexity of this implementation is O(N), since we are using an array of size N to store the elements of both stacks.

# Midterm exam 2021

Câu 3, 5 giống 2022.

### Binary search (25 pts)

Given Array A

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Data | 172 | 98 | 76 | 62 | 40 | 29 | 18 | 10 | 1 | 0 |

Search for the value 97. Fill the following table with corresponding steps in the binary search

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step | Lower bound | Upper bound | Considering Index | Considering data | Conclusion |
| 1 | 0 | 9 | 5 | 29 | Update UB |
| 2 | 0 | 4 | 2 | 76 | Update UB |
| 3 | 0 | 1 | 1 | 98 | Update LB |
| 4 | 2 | 1 |  |  | Stop program |

### Sort (25 pts)

Given array B

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Data | 80 | 4 | 3 | 50 | 30 | 1 | 50 | 88 | 2 | 0 | 15 | 43 | 11 |

Using shell sort algorithm to sort array B in ascending order by filling the table below

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Action/Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| Data | 80 | 4 | 3 | 50 | 30 | 1 | 50 | 88 | 2 | 0 | 15 | 43 | 11 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |

### Linked list (20 pts) – Same 2022

### Queue (20 pts)

What is the best complexity of the enqueue, dequeue, isEmpty, and isFull functions if a priority queue is implemented with different data structures? Fill in the following table with suitable big O values and explanation if needed.

|  |  |  |  |
| --- | --- | --- | --- |
| Data structure | Array | Double Ended Linked list | Doubly Linked List |
| Enqueue |  |  |  |
| Dequeue |  |  |  |
| IsEmpty |  |  |  |
| IsFull |  |  |  |

#### Array

If a priority queue is implemented with an **array**, the **best** **complexity** of the **enqueue** function is **O(n),** where n is the number of elements in the array. This is because we need to find the correct position to insert the new element based on its priority, and this may require shifting other elements in the array.

The **best** **complexity** of the **dequeue** function is **O(1)**, as we can simply remove the first element in the array, which is the one with the highest priority.

The **best complexity** of the **isEmpty** function is **O(1),** as we can simply check if the number of elements in the array is zero.

The **best complexity** of the **isFull** function **is O(1),** as we can simply check if the number of elements in the array is equal to the maximum capacity of the array.

#### Double Ended Linked list

If a priority queue is implemented with a **Double Ended Linked List**, the **best complexity** of the **enqueue** function is **O(1),** as we can simply add a new element to the end of the list without needing to shift any existing elements.

The **best complexity** of the **dequeue** function is also **O(1),** as we can simply remove the first element in the list, which is the one with the highest priority.

The **best complexity** of the **isEmpty** function is **O(1),** as we can simply check if the head pointer of the list is null, indicating that the list is empty.

The **best complexity** of the **isFull** function is **not applicable**, as there is no fixed maximum capacity for a linked list.

#### Doubly Linked List

If a priority queue is implemented with a **Doubly Linked List**, the **best complexity** of the **enqueue** function is **O(1)**, as we can simply add a new element to the end of the list without needing to shift any existing elements.

The **best complexity** of the **dequeue** function is also **O(1),** as we can simply remove the first element in the list, which is the one with the highest priority.

The **best complexity** of the **isEmpty** function is **O(1),** as we can simply check if the head pointer of the list is null, indicating that the list is empty.

The **best complexity** of the **isFull** function is **not applicable**, as there is no fixed maximum capacity for a linked list.

### Stack (10pts) – Same 2022

# Midterm exam 2018

### Binary search (20 pts)

Given array A

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Data | 120 | 90 | 75 | 70 | 29 | 25 | 21 | 11 | 2 | 0 |

Fill the following table with corresponding steps in the binary search algorithm until 90 is found in A

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Step | Lower bound (LB) | Upper bound (UB) | Considering Index | Considering data | Conclusion |
| 1 | 0 | 9 | 5 | 25 | Update UB |
| 2 |  |  |  |  |  |

### Sort (20 pts)

Given array B

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Data | 100 | 52 | 3 | 22 | 31 | 67 | 42 | 75 | 23 | 0 |

Sort array B in ascending order using the quicksort algorithm – where the rightmost item is used as a pivot, by filling the following table

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Action/Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| ……………. |  |  |  |  |  |  |  |  |  |  |
| ……………. |  |  |  |  |  |  |  |  |  |  |

### Stack (25 pts)

Given an IStack interface, and an IDoubleEndList interface

|  |  |
| --- | --- |
| public interface IStack{  void push(int data);  int pop();  int peek();  boolean isEmpty();  boolean isFull();  } | public interface IDoubleEndList {  void insertFirst(int data);  void insertLast(int data);  int getFirst();  int getLast();  int deleteFirst();  int deleteLast();  boolean isEmpty();  } |

Implement a stack using a double-end linked list in such a way that the complexity of the PUSH and POP functions are O(1).

**CODE:**

public class StackUsingDoubleEndedLinkedList implements IStack {

private IDoubleEndList list;

public StackUsingDoubleEndedLinkedList() {

list = new DoubleEndedLinkedList();

}

@Override

public void push(int data) {

list.insertFirst(data);

}

@Override

public int pop() {

if (isEmpty()) {

throw new RuntimeException("Stack is empty");

}

return list.deleteFirst();

}

@Override

public int peek() {

if (isEmpty()) {

throw new RuntimeException("Stack is empty");

}

return list.getFirst();

}

@Override

public boolean isEmpty() {

return list.isEmpty();

}

@Override

public boolean isFull() {

return false; // since we are using a linked list, the stack can never be full

}

}

In this implementation, we use a double-ended linked list to represent the stack. The push operation simply inserts the new element at the beginning of the list using the insertFirst method, which takes O(1) time. The pop operation deletes the first element of the list using the deleteFirst method, which also takes O(1) time. The peek operation simply returns the value of the first element using the getFirst method, which also takes O(1) time. The isEmpty operation checks if the list is empty using the isEmpty method, which also takes O(1) time. Since we are using a linked list, the isFull operation always returns false.

Note that we are assuming that the IDoubleEndList interface provides a double-ended linked list implementation with the above methods that have O(1) time complexity.

### Linked list (25 pts)

What is the best complexity of the push, pop, peek, isEmpty and isFull functions if one implements the IStack inter face with different data structure? Fill in following table with suitable big O values and explanation if needed.

|  |  |  |  |
| --- | --- | --- | --- |
| Data structure | Array | Double Ended Linked list | Doubly Linked List |
| Push |  |  |  |
| Pop |  |  |  |
| Peek |  |  |  |
| IsEmpty |  |  |  |
| IsFull |  |  |  |

public class QueueUsingTwoStacks<T> {

private Stack<T> enqueueStack; // stack for enqueue operations

private Stack<T> dequeueStack; // stack for dequeue operations

public QueueUsingTwoStacks() {

enqueueStack = new Stack<>();

dequeueStack = new Stack<>();

}

public void enqueue(T element) {

enqueueStack.push(element); // simply push the element onto the enqueue stack

}

public T dequeue() {

if (dequeueStack.isEmpty()) { // transfer elements from enqueue stack to dequeue stack if the dequeue stack is empty

while (!enqueueStack.isEmpty()) {

dequeueStack.push(enqueueStack.pop());

}

}

return dequeueStack.pop(); // pop the top element from the dequeue stack

}

public boolean isEmpty() {

return enqueueStack.isEmpty() && dequeueStack.isEmpty(); // queue is empty if both stacks are empty

}

public int size() {

return enqueueStack.size() + dequeueStack.size(); // size of queue is the sum of sizes of both stacks

}

}

* Enqueue operation: Since we are simply pushing the element onto the enqueue stack, the time complexity is O(1).
* Dequeue operation: If the dequeue stack is empty, we transfer all elements from the enqueue stack to the dequeue stack, reversing their order. This takes O(n) time, where n is the number of elements in the enqueue stack. However, this operation only happens once for every n elements, so the amortized time complexity is O(1) per dequeue operation. After transferring the elements, we simply pop the top element from the dequeue stack, which takes O(1) time. Therefore, the worst case time complexity of dequeue operation is O(n), but the amortized time complexity is O(1).
* IsEmpty operation: Checking if both stacks are empty takes O(1) time.
* Size operation: Adding the sizes of both stacks takes O(1) time.

Overall, the time complexity of each operation is very efficient, with enqueue and isEmpty operations taking O(1) time and dequeue and size operations taking O(1) amortized time.

